Bern University
of Applied Sciences

Adding Schnorr's Blind Signature in Taler

An improved Taler Blind Signature System

Course of study	Bachelor of Science in Computer Science
Author	Gian Demarmels, Lucien Heuzeveldt
Advisor	Prof. Dr. Emmanuel Benoist
Expert	Elektronikingenieur HTL Daniel Voisard

Version 1.0 of February 20, 2022

▶ Engineering and Computer Sciences
▶ Institute for Cybersecurity and Engineering ICE

Abstract

GNU Taler is an intuitive, fast and socially responsible digital payment system implemented as free software. While preserving the customers privacy, GNU Taler is still compliant to regulations.

The goal of this thesis is to improve Taler's performance and provide cipher agility by adding support for Schnorr's blind signatures. To achieve this goal, the current state in research for Schnorr signatures needs to be analyzed. After choosing a signature scheme, it has to be integrated into the Taler protocols. Besides implementing the redesigned protocols in Taler, an implementation of the cryptographic routines is needed.

The paper "Blind Schnorr Signatures and Signed ElGamal Encryption in the Algebraic Group Model" [FPS19] from 2019 (updated in 2021) introducing Clause Blind Schnorr Signatures is used as theoretical basis for our improvements. The paper explains why simple Blind Schnorr Signatures are broken and how the Clause Schnorr Blind Signature scheme is secured against this attack.
Compared to the currently used RSA Blind Signatures, the new scheme has an additional request, two blinding factors instead of one and many calculations are done twice to prevent attacks.

The Taler protocols were redesigned to support the Clause Blind Schnorr Signature scheme, including slight alterations to ensure *abort-idempotency*, and then further specified. Before starting with the implementation of the redesigned protocols, the cryptographic routines for Clause Blind Schnorr Signatures were implemented as part of the thesis.
All of the implemented code is tested and benchmarks are added for the cryptographic routines.

Multiple results were achieved during this thesis: The redesigned protocols Taler protocols with support for Clause Blind Schnorr Signatures, the implementation of the cryptographic routines, the implementation of Talers core protocols and a detailed comparison between RSA Blind Signatures and Clause Blind Schnorr Signatures. Overall, the Clause Blind Schnorr Signatures are significantly faster, require less disk space, and bandwidth and provide *cipher agility* for Taler.

Acknowledgement

We would like to kindly thank Christian Grothoff (Bern University of Applied Sciences) for his extensive advice, support and very helpful feedback during our whole thesis.
We also kindly thank Jeffrey Burdges (Web 3, Switzerland) for reviewing the proposal containing the redesigned protocols and giving feedback.
Further, we kindly thank Jacob Appelbaum (Bern University of Applied Sciences, Eindhoven University of Technology) for further results for the performance measurements of our cryptographic routines and the insightful conversations.

Contents

1. Introduction

1.1. Motivation

Public key cryptography based on elliptic curves allows smaller key sizes compared to other cryptographic systems. While still providing equivalent security, the smaller key size leads to huge performance benefits.
Blind Signatures are one of the key components upon which Taler's privacy is built upon. Our thesis adds support for a modern cryptographic scheme called the Clause Blind Schnorr Signature scheme [FPS19].
Additionally to the benefits of ellicptic curve cryptography, adding a second blind signature scheme makes Taler independent of a single cryptographic scheme and thus provides *cipher agility*.

1.2. Goals

The project definition is as follows [Ben21]:
The students will implement the blind Schnorr signature inside Taler. Taler is a system for the management of virtual money. Taler is based on coins that need to be signed by an exchange (for instance a bank). In the actual version of the system, coins are signed by the exchange using Schaum's bind-signature protocol. This allows users to have signed coins, without the exchange knowing what it signed. This step is fundamental for the privacy protection of the system.
The students have to insert the Schnorr blind signature algorithm inside the protocol for the creation of coins. But they also need to change the Taler subsystems where the verification of the signature is done.
The actual Taler system allows people to let an exchange sign a coin for which they do not have the private key. This is a security issue (for misuse of coins on the dark-net for instance). An optional task for the project is to prevent a user to let an exchange sign a public key when the client does not have access to the corresponding private key.
Here is a list of the tasks that the students must do:

- Design a protocol integrating Schnorr blind signature in the creation of Taler coins.
- Implement the protocol inside the exchange application and the wallet app.
- Analyze the different Taler subsystems to find where the blind signature is verified.
- Replace verification of the blind signature everywhere it occurs.

- Compare both blind signature systems (Schaum's and Schnorr's), from the point of view of security, privacy protection, speed, ...
- Write tests for the written software.
- Conduct tests for the written software.
- Transfer the new software the Taler developers team

Here is a list of optional features:

- Design a protocol, such that the exchange can verify that the user knows the private key corresponding to the coin that is to be signed.
- Implement that protocol.

1.3. Scope

In scope are all necessary changes on the protocol(s) and components for the following tasks:

- Research the current state of Blind Schnorr Signature schemes
- Redesign the Taler protocols to support Blind Schnorr signatures
- Add support for a Blind Schnorr Signature Scheme in the exchange, merchant, wallet-core, wallet web-extension and optionally on the android mobile wallet
- design and implement a protocol where the user proves to the exchange the knowledge of the coin that is to be signed (optional)

Out of scope is production readyness of the implementation. This is because changes in the protocos and code need to be thoroughly vetted to ensure that no weaknesses or security vulnerabilities were introduced. Such an audit is out of scope for the thesis and is recommended to be performed in the future. The iOS wallet will not be considered in this work.
It is not unusual that a scope changes when a project develops. Due to different reasons, the scope needed to be shifted. Since there are no libraries supporting Clause Blind Schnorr Signatures, the signature scheme has to be implemented and tested before integrating it into Taler. While this is still reasonable to do in this project, it will affect the scope quite a bit. The analysis of the optional goal showed, that a good solution that aligns with Taler's goals and properties needs more research and is a whole project by itself.

Scope changes during the project:

- **Added:** Implement the cryptographic routines in GNUnet
- **Removed:** design and implement a protocol where the user proves to the exchange the knowledge of the coin that is to be signed (optional)

- **Adjusted:** Focus is on the implementation of the exchange protocols (Withdraw, Spend, Refresh and cryptographic utilities)
- **Adjusted:** Implementation of the refresh protocol and wallet-core are nice-to-have goals
- **Removed:** The Merchant and the android wallet implementations are out of scope

2. Preliminaries

2.1. GNU Taler Overview

This chapter provides an high-level overview of GNU Taler with its core components. The purpose of this chapter is to provide all the necessary details to understand this work and is not a specification nor a documentation of GNU Taler. For more information on GNU Taler refer to [Dol19] or the GNU Taler documentation [SAc].
Generally, GNU Taler is based on Chaumian e-cash [Dav83]. The following parts discuss the different entities seen in the figure 2.1

2.1.1. Components

In this section the different components are described as in [Dol19].

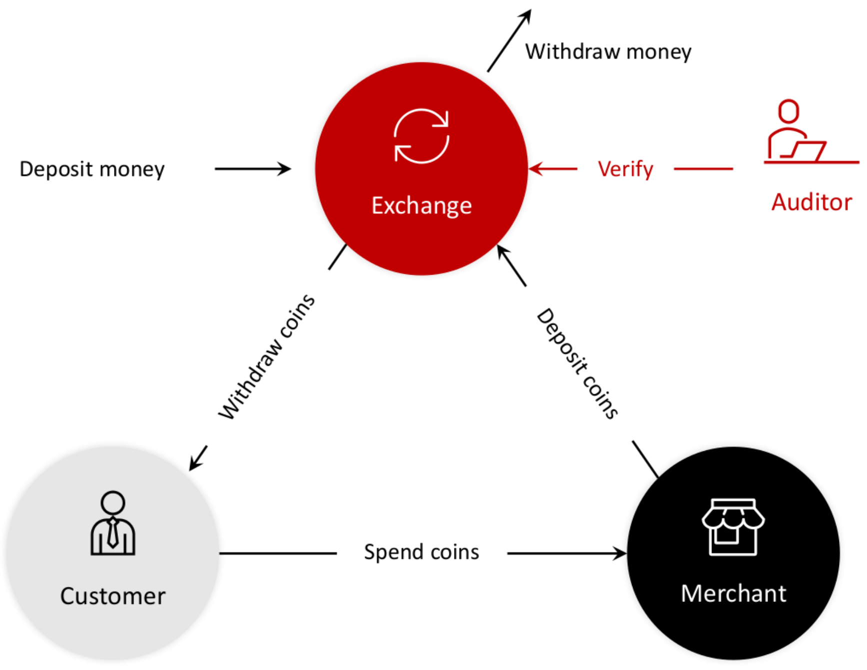

Figure 2.1.: GNU Taler simple overview (source: [Tal21c])

Exchange

The exchange is the payment service provider for financial transactions between a customer and merchant. The exchange holds bank money as reserve for the anonymous digital coins.
Details of the exchange's functionality can be found in section 4.3 from Florian Dold's thesis [Dol19] or in the documentation [SAd].
The code can be found in the exchange's git repository [Repd].

Customer (Wallet)

A customer holds Taler Coins in his electronic wallet. As we see in figure 2.1, a customer can withdraw coins from the exchange. These coins can then be spent with a merchant.
Details of the wallet's functionality can be found in section 4.6 from Florian Dold's thesis [Dol19] or in the documentations [SAi] [SAh].
Git Repositories:

- Main repository [Repj]
  This Repository includes the wallet-core and the implementations for the web extension and CLI.
- Android app [Repg]
- iOS app [Reph]

Merchant

A merchant accepts Taler Coins in exchange for goods and services. The merchant deposits these coins at the exchange and receives bank money in return.
Details of the wallet's functionality can be found in section 4.5 from Florian Dold's thesis [Dol19] or in the documentations:

- Operator manual [SAf]
- Merchant API [SAe]
- Back-Office [SAa]
- Point-of-Sales [SAg]

Git Repositories:

- Backend: [Repe]
- Backoffice: [Repb]
- Point-of-Sales App: [Repg] (part of android repo)

Merchant Frontend Repositories:

- Payments with Django: [Repc]
- Wordpress woocommerce plugin: [Repk]
- Saleor Frontend: [Repf]
- Demo Frontends: [Repi]

Auditor

The auditors, which are typically run by financial regulators, have the purpose to monitor the behavior of the exchanges to assure that exchanges operate correctly.
Details of the auditor's functionality can be found in section 4.4 from Florian Dold's thesis [Dol19] or in the documentation [SAb].
Git Repositories:

- Main repository [Repd] (Part of exchange repository, inside ./src/auditor and ./src/auditordb)
- Auditor's public website [Repa]

Bank

The banks receive wire transfer instructions from customers and exchanges. As long as the banks can make wire transfers to each other, the Taler parties do not have to have the same bank.

2.1.2. Taler Step by Step

This is a high-level overview of what Taler generally does. Many details (like privacy of the buyer, income transparency) are left out and are explained in the following sections. We see in Figure 2.2 how Taler works step by step (at a high-level).

1. The customer decides to withdraw Taler coins. To do this, he goes to his bank and gives the order to pay the exchange.
2. The customers bank receives the customers order and makes a wire transfer to the exchanges Bank.
3. The exchange has received the money and the customer can now withdraw coins to his wallet.
4. The customer can now spend his coins at a merchant or merchants of his choice.
5. The merchant can then deposit the coins at the exchange.
6. The exchanges bank makes a wire transfer to the merchants bank.
7. The merchant has successfully received the money for the goods he sold.

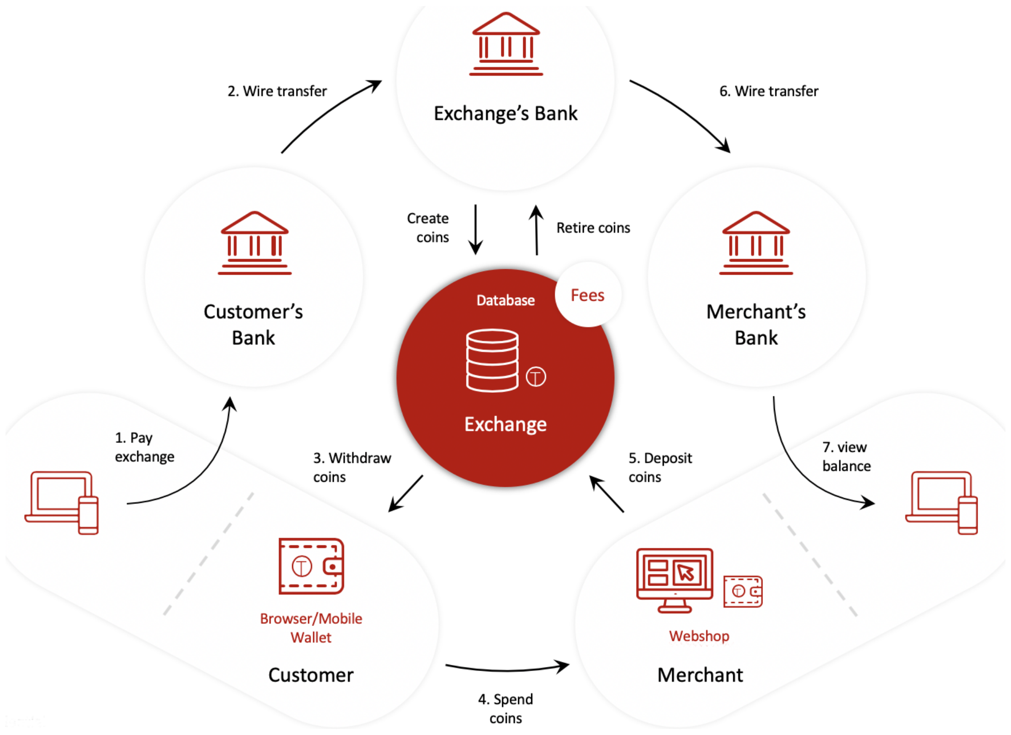

Figure 2.2.: GNU Taler overview (source: [Tal21b])

2.1.3. Protocols Overview

This section provides a high-level overview of the different Taler protocols. The details are here omitted and discussed later.

Refresh Protocol

Taler has a quite interesting protocol to get change. The purpose of the protocol is to give unlinkable change. When a customer buys something from a merchant, in most situations he does not have the exact sum in coins. For this reason, change is needed to provide a convenient payment system. A coin can be partially spent. When this happens, the exchange and the merchant know that this coin is used for that specific contract. If the rest of this coin would be spent in future, one could link these two transactions. Therefore, a mechanism to get unlinkable change while still preventing money laundering or tax evasion is needed.

Refund

Taler has a built-in refund functionality. Merchants can instruct the exchange to refund a transaction before the refund deadline. The customer then refreshes the coin(s) in order for payments to remain unlinkable.

Payment Fees

The exchange can charge fees for withdrawal, refreshing, deposition of coins. These fees can depend on the denomination since different denominations can have different storage requirements. Merchants are able to cover these costs fully or partially.
Exchanges are also able to aggregate wire transfers to merchants, thus reducing wire transfer fees.

Tipping

Merchants can give customers a small tip. This feature can be useful for different use cases, for example a merchant can give a tip when a customer participates in a survey.

Auditing

Financial auditing is built-in to Taler in the form of auditors. Auditors have read access to certain exchange databases. Their task is to verify that exchange work as expected, see chapter 4.4 in Florian Dold's thesis [Dol19] for more details. In future versions, the auditor will provide an interface that can be used by merchants to submit deposit confirmation samples. This can be used to detect compromised signing keys or a malicious exchange.

2.1.4. Properties

This section describes Taler's properties.

Free Software

The core components of GNU Taler are under the following licenses:

- exchange [Repd]: GNU AGPL[1]
- merchant [Repe]:
  - backend: GNU GPL[2]v3+, GNU AGPL
  - library: GNU LGPL[3]
- wallet-core [Repj]: GNU GPL

[1]GNU Affero General Public License
[2]GNU General Public License
[3]GNU Lesser General Public License

Buyer Privacy Protection

Taler protects the privacy of buyers during the different stages in the lifetime of a coin:

1. Reserve: The reserve is identified by a key pair (private and public key). This means that the exchange doesn't know the identity of the reserve account holder. Whoever knows the private key is able to withdraw from the corresponding reserve.
2. Withdrawal: The withdrawal process is encrypted with TLS and uses a blind signature scheme. Therefore the exchange doesn't know which customer holds which coin.
3. Payment: The complete payment process doesn't rely on any information identifying a customer.

Beware that an anonymous bi-directional channel is assumed for communication between the customer and the merchant as well as during the retrieval of denomination key from the exchange and change for partially spent coins (between customer and exchange).

Merchant Taxability

Merchant's incomes are transparent to auditors which makes taxation by the state possible.
A buyer could theoretically transfer the private key and signature of a coin directly to the merchant to bypass the exchange. However, this is suboptimal for the merchant because the knowledge of the coin doesn't grant him the sole ownership. If the customer spends the coin in another transaction before the merchant, the coin is voided before the merchant claims its value, thus rendering this form of payment unusable. The same principle holds for change (refreshed coins) because it is linked to the original coin. Whoever knows the private key and signature of the original coin can obtain the change and use it before the merchant.

Anti Money Laundering and Combating Financing of Terrorism

Every transaction contains the cryptographic hash of the associated contract. This enables the authorities to request the merchant to reveal the transaction details (the contract). If the merchant isn't able to reveal the contract, in other words fails to deliver a contract with the same hash which is included in the transaction, he risks punishment or further investigation.
Another aspect for AML[4] and CFT[5] are KYC[6] checks. Know Your Customer checks require certain institutions to verify certain information about their business partners in order to prevent money laundering and terrorism (see [Wik20]).
GNU Taler implements these KYC checks:

[4] Anti Money Laundering
[5] Combating Financing of Terrorism
[6] Know Your Customer

- Exchanges know the identities of their customers.
- Merchants might need to pre-register with exchanges (depending on the deployment scenario).

Payer Fraud Prevention

The following definition was taken from the BigCommerce website [Big].

"Payment fraud is any type of false or illegal transaction completed by a cybercriminal. The perpetrator deprives the victim of funds, personal property, interest or sensitive information via the internet."

Prevention of payment fraud is a design goal for GNU Taler.

Minimal Information Disclosure

GNU Taler aims to disclose as minimal information as possible. This mostly concerns customers, but merchants also profit by keeping financial details hidden from competitors.

Single Point of Failure Avoidance

SPOF[7]s are fatal because a failure in this component can bring the complete system to a halt.

Offline Payment (unsupported)

GNU Taler doesn't offer offline payments due to the CAP problem (see chapter "Challenges of offline payments" in [Chr21]).

2.2. Cryptographic Preliminaries

In this section we will cover the necessary preliminaries to understand Taler. For this part we took most of the information from Nigel P. Smarts book Cryptography made simple [Sma16] and from the course "Applied Cryptography" at the BFH. The chapter includes preliminaries of the already implemented cryptographic schemes and the ones that are implemented during this work.

[7]Single Point of Failure

2.2.1. Hash Functions

As source for this section, page 271-275 were used in *Cryptography made Simple* [Sma16]. In this paper a hash function is always a cryptographic hash function. Cryptographic hash function are one-way functions $H()$, which are calculating the hash value h from a message m so that $h = H(m)$. With known input one can easily calculate the hash value. The other way around is computationally infeasible.
Cryptographic hash functions have the following properties.

(First) Preimage Resistance

It should be hard to find a message with a given hash value. For a given output y it is impossible to calculate the input x such that $h(x) = y$.
This basically means, a hash function can not be inverted, not even with unlimited computing power. Too much information is destroyed by the hash function and there are many values resulting in the same hash.

Second Preimage Resistance

Given one message, it should be hard to find another message with the same hash value. For a given x_1 and $h(x_1)$ it is hard to find a x_2 such that $h(x_1) = h(x_2)$.

Collision Resistance

It should be hard to find two messages with the same hash value. It is quite obvious that collisions are existent, since there are more possible messages than hash values. This is also known as the pigeonhole principle. Even if there are hash collisions, it should be hard to find $x_1 \neq x_2$ such that $h(x_1) = h(x_2)$. Due to the birthday paradoxon (a detailed description can be found under [Wik21a]) it is easier to cause a collision of two arbitrary messages than of a specific message.

2.2.2. Key Derivation

A KDF[8] derives one or more cryptographically strong secret keys from initial keying material by using a Pseudo Random Function. Therefore, input of a KDF is some sort of keying material (e.g. from a key exchange). Output will be a pseudo-random bit-string, which can be used as new key material.

Pseudo Random Function

A PRF[9] is a deterministic function whose output appears to be random if the input is unknown. The output is computationally indistinguishable from a true random source. Dif-

[8] Key Derivation Function
[9] Pseudo Random Function

ferent PRFs exist, for example AES[10] or HMAC could be used as PRF. In the case of HKDF, HMAC is a suitable choice as PRF.

HMAC

A Message Authentication Code (MAC[11]) provides **unforgeability**, which means, only a person who knows the key k can compute the MAC. Further, a MAC protects the **message integrity**, since unauthorized changes are being detected. Last but not least, **message authenticity** is provided too, since only a person who knows the key can compute the HMAC. However, it does not provide non-repudation because it is a shared secret. MACs take a message and a key as input and give the MAC tag as output.
One way to design such MACs is by using a hash function. The obvious way one would design such a function would most likely be: $t = H(k||m||pad)$ However, this variant would be **completely insecure** with hash functions based on Merkle-Damgard constructions. Because of the structure of such hash functions, it is easy to find $H(M||X)$ for an arbitrary X and a hash value $H(M)$, with that a so called *length-extension* attack is possible.
HMAC prevents this attack by computing the MAC as follows: $t = \text{HMAC}_k(m) = H((k \oplus opad)||H((k \oplus ipad)||m))$
H() could be any standard hash functions, for example SHA-256, SHA-512 or SHA-3. $\oplus$ stands for the XOR operation. HMAC is specified in [RFC2104].

HKDF

HKDF follows the *extract-then-expand* paradigm and therefore has two phases. In the extract phase, the input keying material is taken and a fixed-length pseudorandom key K is *extracted*. This phase is used to generate a high entropy pseudorandom key from potentially weaker input keying material. This key K is used in the *expand* phase to output a variable-length, pseudorandom key.

The HKDF makes use of HMAC (2.2.2) instantiated with a hashfunction 2.2.1. It takes the input keying material, a salt and the length of output keying material as arguments. HKDF is specified in [RFC5869].

2.2.3. Digital Signatures

As source for this section, page 216-218 were used in *Cryptography made Simple*[Sma16]. A digital signature is a cryptographic function used to verify the origin and integrity of a message. It provides the following properties:

- Sender authenticity: The origin/sender of a message can not be forged.
- Message integrity: No unauthorized change to the message can be made, the message is tamperproof.

[10] Advanced Encryption Standard
[11] Message Authentication Code

- Non-repudiation: After a message is signed, one can not dispute that a message was signed.

If verification is successful, only Alice knows her private key and Bob uses Alice's public key to verify, then Bob knows that this message is really from Alice and the message has not been tampered or further modified. A digital signature scheme has a message space M, a signature space S and three algorithms:

- Key generation: $(pk, sk) \leftarrow keyGen()$
- Signatue generation: $s \leftarrow$sign$_s k(m)$
- Verification: $v \leftarrow$verify$_p k(m, s)$ where $v \in 0, 1$

If the result of the verification algorithm equals 1, a signature for m is called valid.
Digital signatures are publicly verifiable, which means anyone can verify that (m, s) is legitimate.

Adversary Models & Provable Security

Digital Signature schemes are believed to be secure when they are EUF-CMA secure. Existentially Unforgeability (EUF[12]) means that given a public key pk the adversary cannot construct a message with a valid signature, except with a negligible probability. Choosen-Message Attack (CMA[13]) means that an adversary can ask a signing oracle to produce valid signatures $s' = sign_{sk}(m')$ for arbitrary messages $m' \neq m$.
EUF-CMA is therefore existentially unforgeability under chosen message attack and is a standard security model for digital signatures. More details can be found in page 217-218 in *Cryptography made Simple* [Sma16].

RSA-FDH Signature Scheme

As source for this section, pages 300-301 and 333-335 were used in *Cryptography made Simple* [Sma16].

RSA-FDH is a deterministic digital signature scheme which provides authenticity, message integrity and non-repudation. The RSA signature scheme (without the full domain hash) is NOT EUF secure and is vulnerable to existential forgery attacks. RSA-FDH is one possible solution for a EUF-CMA secure scheme. EUF-CMA and its adversary model is further discussed in section 2.2.3. RSA-FDH is EUF-CMA secure under factoring and RSA assumptions. More details on the hardness assumptions can be found on page 32-49 in *Cryptography made Simple* [Sma16].

[12]Existentially Unforgeability
[13]Choosen-Message Attack

Full-Domain Hash A Full-Domain Hash is a hash function with an **image size equal to the size of the RSA modulus**. The hashfunction $h()$ used in the RSA-FDH sign (section 2.2.3) and RSA-FDH verify (section 2.2.3) needs to fulfill all the security properties we defined in chapter 2.2.1. This means that the image is a co-domain of the RSA group $\mathbb{Z}_N^*$. Provided that the hashfunction has properties of a random oracle, **RSA-FDH is provably EUF-CMA secure** under the RSA assumption.

RSA Key Generation The information in this section is from the script of the BFH module *Public Key Cryptography* taught by Prof. Dr. Walter Businger ([Bus21]). The RSA private and public key are generated like this:

1. Generate two random prime numbers p, q where $p \neq q$
2. Calculate $N = pq$
3. Calculate $\lambda = \text{lcm}(p-1, q-1)$
4. Randomly choose a number d which is bigger than p and q and where $\text{gcd}(d, \lambda) = 1$
5. Calculate e, the multiplicative inverse of $d \mod \lambda$
6. The public key is (e, N), the private key is (d, N)
7. Destroy all numbers not included in the private or public key

Note that "lcm" stands for least common multiplier and "gcd" means greatest common divisor. The original RSA specification uses $\phi(n) = (p-1)(q-1)$ instead of $\lambda = \text{lcm}(p-1, q-1)$. $\phi(n)$ is a multiple of λ (for details see [Bus21]).

Signature Algorithm The signature can be calculated as following:
$s \leftarrow (\text{FDH}(m))^d \mod N$

Verification Algorithm The signature can be validated as following:
$\text{FDH}(m) \leftarrow s^e \mod N$

Schnorr Signature Scheme

The Schnorr Signature scheme is a randomized signature scheme, which is proven to be EUF-CMA secure under DLP[14]. More information about the DLP can be found in chapter 3 of *Cryptography made Simple* [Sma16]. In february 2008 the patent expired and Schnorr signatures are now becoming widely deployed. (eg. EdDSA). Schnorr signatures gained quite some attraction lately, since Bitcoin has announced to support Schnorr signatures starting from Block 709632 (see [Pie20], [Bit], and [Repm]). As reference for the Schnorr signature scheme (and later Clause Blind Schnorr Signature Scheme) we use the paper *Blind Schnorr Signatures and Signed ElGamal Encryption in the Algebraic Group Model* [FPS19] as general source for Schnorr related schemes.

[14]Discrete Logarithm Problem

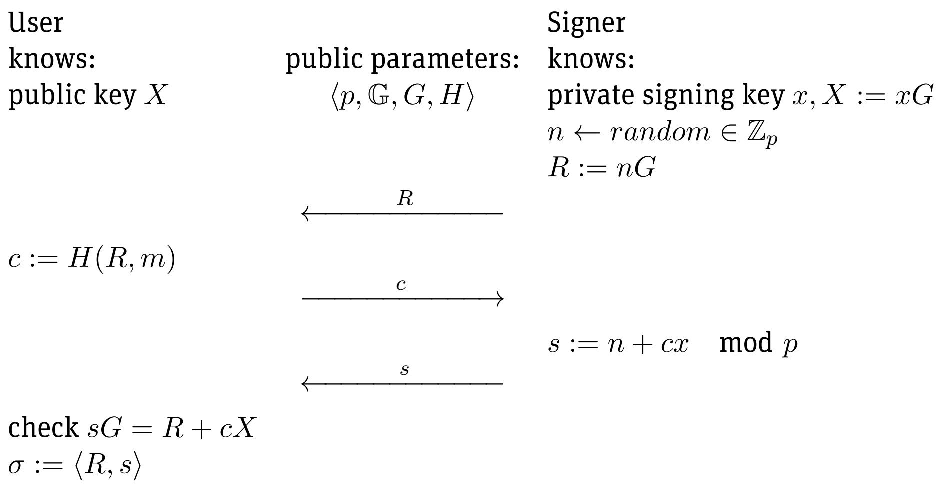
User
knows:
public key X
public parameters:
$\langle p, \mathbb{G}, G, H \rangle$
Signer
knows:
private signing key $x, X := xG$
$n \leftarrow random \in \mathbb{Z}_p$
$R := nG$
R
$c := H(R, m)$
c
$s := n + cx \mod p$
s
check $sG = R + cX$
$\sigma := \langle R, s \rangle$

Figure 2.3.: Schnorr signature protocol with user who wants to sign a message m by the signer

Setup We have a Group $\mathbb{G}$ of order p and a generator G of the group. Further a Hashfunction $H : \{0, 1\}* \rightarrow \mathbb{Z}_p$ is used.

Key Generation The key generation is the same as in DSA[15].

1. private key is a random integer $x \leftarrow random \in \mathbb{Z}_p$
2. public key is $X \leftarrow xG$

Sign The sign function takes the secret key x and the message m to be signed as argument. The interactive version with a signer and a user can be seen in figure 2.3.

1. choose $r \leftarrow random \in \mathbb{Z}_p$
2. calculate $R := rG$
3. $c := H(R, m)$
4. $s := r + cx \mod p$
5. $\sigma := (R, s)$
6. return σ

[15]Digital Signature Algorithm

Verify The verify function takes the public key X, the message m and the signature σ as argument.

1. $c := H(R, m)$
2. check $sG = R + cX$
3. return true if check was successful, false otherwise

The verification holds because $sG = R + cX$ is $(r + cx)G = rG + cxG$ which is equal.

Edwards-curve Digital Signature Algorithm

EdDSA[16] is a scheme for digital signatures based on twisted Edwards curves and the Schnorr signature scheme. The information described here originates from [JL17] and [Wik21c]. EdDSA is a general algorithm that can be used with different curves. A choice in curves consists of 11 parameters. These are the most important (the others can be found in [JL17]:

- odd prime power q (used to generate elliptic curve over finite field $\mathbb{F}_q$)
- integer b, where $2^{b-1} > q$, describing the bit size of various elements
- cryptographic hash function H with output size of $2b$
- base point on curve B (generator)
- number c, (either 2 or 3)
- prime number L where $LB = 0$ and $2^c * L =$ number of points on the curve

Key Creation The private key k is a random bit-string of length b. The public key A is a point on the curve. To generate it, we calculate $A = sB$ where $s = H(k)[: b]$ (meaning that we take the b least significant bits from the output of the hash function as s).

Signature Creation An EdDSA signature of a message M is composed of (R, S), which are generated as follows:

$$\begin{aligned} s &= H(k)[: b] \\ r &= H(H(k)[b+1 : 2b] \,||\, M) \\ R &= rB \\ S &= (r + H(R||A||M) * s) \mod L \end{aligned}$$

Note that $[: b]$ means taking the b least significant bits, $[b+1 : 2b]$ means taking the b most significant bits and $R||A$ means concatenating R and A.

[16] Edwards-curve Digital Signature Algorithm

Signature Verification (R, S) is the signature, M is the message, A is the public key and c, B are curve parameters. To verify a signature, the following equation must be satisfied: $2^c SB = 2^c R + 2^c A * H(R||A||M)$
This means that verify() returns 1 if the equation holds and 0 otherwise.

Ed25519 Ed25519 is an EdDSA based signature scheme and uses Curve25519 (see [Ber06]), which offers 128 security bits. Curve25519 gets its name from the prime $2^{255} - 19$ and is designed for fast computation and to resist side channel attacks.
These are the most important EdDSA parameters for Ed25519 (the others can be found in [JL17]):

- $q = 2^{255} - 19$
- $b = 256$
- $H()$: SHA-512
- $B = (15112221349535400772501151409588531511454012693041857206046113283949847762202, 46316835694926478169428394003475163141307993866256225615783033603165251855960)$
- $c = 3$
- $L = 2^{252} + 27742317777372353535851937790883648493$

2.2.4. Blind Signature Schemes

One could think of blind signatures as a message put into an envelope made of carbon paper. The signer stamps his signature on the envelope and due to the properties of a carbon paper, the message is now signed too. (the stamp ”stamps” through the envelope on the message). The client then can open the envelope, and he possesses a correctly signed message. This is achieved by the client by blinding the message with a blinding factor before sending to the signer (”blind()” operation). The signer signs the blinded message and returns the signature of the blinded message to the client. The client, who possesses the blinding factor can then unblind the signature and gets a signature of the original message (”unblind()” operation). The explanation above leads us to the additional security property of a blind signature, the *blindness* of signatures. This property requires that a signer cannot link a message/signature pair to a particular execution of the signing protocol [FPS19].
A blind signature scheme is called *perfectly blind* if the generated signature (*unblinded* signature) is statistically independent of the interaction with the signer (*blinded* signature). Thus, blind signatures cannot be linked to the signer interaction in an information theoretic sense. [Sch04] [CP93]

RSA Blind Signature Scheme

As source for this section, the course material from ”Applied Cryptography” from BFH and [Dav83] were used. The process for receiving a valid signature from the exchange uses a blind signature scheme invented by David Chaum ([Dav83]) which is based on RSA signatures. The process is described in figure 2.4.
Note that Bob (the signer) uses a standard RSA signature and can’t verify if the message from Alice is blinded. Mathematically a blind signature works similar to the ”naive” RSA

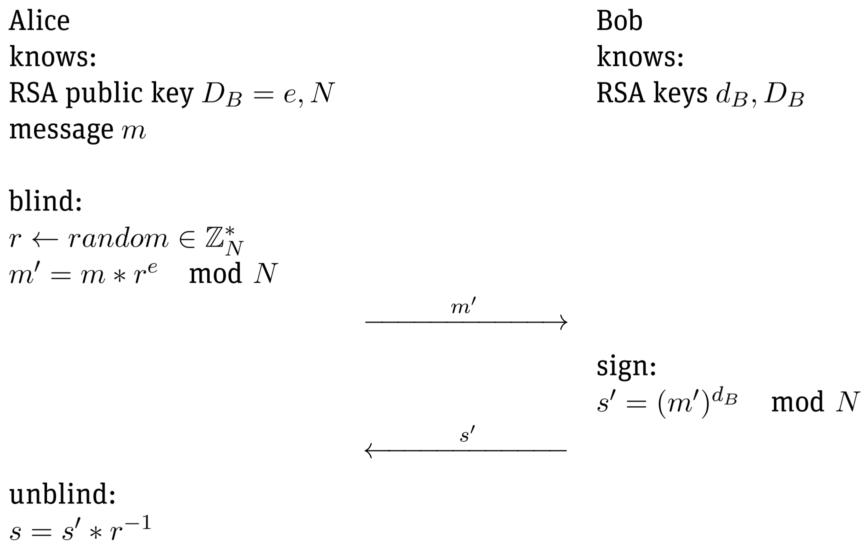

Figure 2.4.: Blind signature scheme

signature scheme. We consider Alice as the party who wants to have a message m blindly signed by Bob. Bob has a public key $D_B = (e, N)$ and his corresponding private key d_B known only by Bob. Alice needs to generate a random blinding factor $r \in \mathbb{Z}_N^*$, which needs to remain secret. Alice then calculates $m' = m*r^e \mod N$. The blinded value m' will now be sent to Bob by Alice. Bob on his side calculates now the signature as usual: $s' = m'^{d_B} \mod N$. The signature s' is sent to Alice by Bob. Alice can calculate the signature as following:
$s = s' * r^{-1}$.
s is a valid signature of m, while the signer, Bob, does not know m nor b.
We now want to analyze this closer to understand why blind signatures work. Let’s look at this equation:
$s' = m'^{d_B} = (m * r^e)^{d_B} = m^{d_B} * (r^e)^{d_B}$.
The interesting part for now is $(r^e)^{d_B}$, since this is r^1. This means the signature s' we got from Bob is $s' = m^{d_B} * r^1$. Now it is quite obvious how the valid signature s can be calculated by multiplying with the inverse of r as in: $s = m^{d_B} * r^1 * r^{-1} = s' * r^{-1}$.

Blindness RSA Blind Signatures are considered *perfectly blind* (see subsubsection 2.2.4). There exist multiple $\langle r, m\rangle$ pairs that matches m' such that $m' = m * r^e \mod N$. Thus, RSA Blind Signatures achieves *perfect blindness* which cannot be attacked by brute-force or similar attacks. Even if a valid $\langle r, m\rangle$ pair is found, the attacker has no possibility to know if it is the correct pair without additional information.

RSA Blinding Attack There are also some possible attacks on this scheme. First this is subject to the RSA blinding attack. In this attack the property is used, that the signing operation is mathematically equivalent to the encrypt operation in RSA. The attacker has a ciphertext $c = m^d$ and he wants to break this message. Now, the attacker uses the ciphertext c as "message" in the blind signature scheme above.
$m'' = cr^e \mod n = (m^e \mod n) * r^e \mod n = (mr)^e \mod n$.
The Attacker then sends the blinded message m'' to the signer who blindly signs the blinded message.
$s' = m''^d \mod n = (mr)^{ed} \mod n = m * r \mod n$.
The attacker recovers the message now with $m = s' * r^{-1} \mod n$.
This attack could be prevented by the use of a padding scheme, however this would break RSA symmetry. In blind signatures the RSA symmetry is needed, otherwise it would produce an incorrect value in the unblind operation.
Due to this issue; One should never use the same key for signing and encryption! A version of blind signatures, RSA-FDH will be discussed, which solves this issue. [Wik21b]

Low Encryption Exponent Attack For this attack a possibly small message m and a small public key e is given. If now $c = m^e < n$, one could compute $m = \sqrt[e]{c}$. Similar to the RSA blinding attack, padding could solve the issue, however RSA symmetry is needed. To overcome this issue, RSA-FDH blind signatures are introduced in the next chapter.

RSA-FDH Blind Signatures

As source for this section, the course material from "Applied Cryptograhy" from BFH and [Dav83] were used. Blind signatures are discussed in 2.2.4. This version is quite similar to the blind signatures already introduced in figure 2.4. In addition, the FDH introduced in section 2.2.3 is used. The difference is that the message does not get directly blinded, it gets hashed before with a Full-Domain Hash.
Given Alice's message m and Bobs public key $D_B = (e, n)$. As in the simple RSA Blind Signatures, a random blinding factor $r \in \mathbb{Z}_N^*$ is generated. Before the message is blinded, the Full-Domain Hash $f = \text{FDH}(m)$ is calculated, which then is blinded as in $f' = fr^e \mod n$. Since the hash function is a Full-Domain Hash, f is in the RSA domain $\mathbb{Z}_N^*$. Now proceed as in the blind signature scheme introduced in the previous section. The blinded hash f' will be transmitted to Bob who then computes the signature $s' = f'^d \mod n$ and sends s' back. Alice unblinds s' and gets the valid signature $s = s'r^{-1} \mod n$.

This version of blind signature is not subject to the attacks introduced in the previous section.

Alice
knows:
RSA public key $D_B = e, N$
message m

$Compute f = FDH(m)$

blind:
$r \leftarrow random \in \mathbb{Z}_N^*$
$f' = f * r^e \mod N$

Bob
knows:
RSA keys d_B, D_B

$$\xrightarrow{f'}$$

sign:
$s' = (f')^{d_B} \mod N$

$$\xleftarrow{s'}$$

unblind:
$s = s' * r^{-1}$

Figure 2.5.: RSA-FDH blind signatues

Blind Schnorr Signature Scheme

The Blind Schnorr Signature Scheme **is considered broken** and should not be implemented. This section is here to explain how blind Schnorr signatures generally work and should help to understand The Clause Blind Schnorr Signature Scheme 2.2.4.

For the signer the calculations are the same as in the original Schnorr Signature Scheme 2.3. The exchange chooses a random $n \leftarrow random \in \mathbb{Z}_p$ and calculates $R := nG$ as before. In comparison to the Schnorr Signature Scheme (see section 2.2.3) we generate two random blinding factors $\alpha, \beta \leftarrow random \in \mathbb{Z}_p$ to achieve *blindness*. The User then calculates $R' := R + \alpha G + \beta X$. This R' is then used to calculate $c' := H(R', m)$ and is blinded with b as in $c := c' + \beta \mod p$. The challenge c is then blindly signed by the signer $s := n + cx \mod p$. The User checks if the signature is valid the same way as in the original protocol. Finally the user has to unblind s as in $s' := s + \alpha \mod p$. Now the unblinded signature is $\sigma := \langle R', s' \rangle$. This scheme is described in figure 2.6. More details can be found in the *Blind Schnorr Signatures and Signed ElGamal Encryption in the Algebraic Group Model* paper [FPS19].

To verify the signature, the verifier has to check if the following equation holds:

$$\begin{aligned} s'G &= R' + c'X \\ &= R' + H(R', m)X \end{aligned}$$

s', R' together form the signature, X is the public key and m is message.

The reason why this works is that the original Schnorr signature verification algorithm remains the same in blind signatures.

$$sG = R + cX$$

By replacing s, R, c, with the values used in the blind signature scheme (as in figure 2.6)

$$\begin{aligned} s &= s' - \alpha \\ R &= R' - \alpha G - \beta X \\ c &= c' + \beta \end{aligned}$$

we receive the following equation:

$$\begin{aligned} sG &= R + cX \\ (s' - \alpha)G &= R' - \alpha G - \beta X + (c' + \beta)X \\ s'G - \alpha G &= R' - \alpha G + c'X \\ s'G &= R' + c'X \end{aligned}$$

User knows: public key X	public parameters: $\langle p, \mathbb{G}, G, H \rangle$	Signer knows: private signing key $x, X := xG$ $r \leftarrow random \in \mathbb{Z}_p$ $R := rG$
	$\xleftarrow{\quad R \quad}$	
$\alpha, \beta \leftarrow random \in \mathbb{Z}_p$ $R' := R + \alpha G + \beta X$ $c' := H(R', m)$ $c := c' + \beta \mod p$		
	$\xrightarrow{\quad c \quad}$	
		$s := r + cx \mod p$
	$\xleftarrow{\quad s \quad}$	
check $sG = R + cX$ $s' := s + \alpha \mod p$ $\sigma := \langle R', s' \rangle$		

Figure 2.6.: The broken Schnorr Blind Signature Scheme

Blindness Blind Schnorr Signatures also achieve *perfect blindness* (subsection 2.2.4). [CP93] [FPS19]

ROS Problem The security of Blind Schnorr Signatures relies on an additional hardness assumption, the *Random inhomogeneities in an Overdetermined, Solvable system of linear equations* or ROS problem. [Sch01] Solving the ROS[17] problem breaks the unforgeability property of blind Schnorr signatures by finding $l+1$ signatures out of l signing operations. David Wagner showed in his paper that the ROS problem can be reduced to the $(l+1)$-sum problem and therefore showed that an attack is practicable. [Wag02] More details about ROS and Wagner's algorithm can also be found in the paper *Blind Schnorr Signatures and Signed ElGamal Encryption in the Algebraic Group Model* [FPS19].
Due to the possible attack, Blind Schnorr Signatures are considered **broken** and should not be used. The next section 2.2.4 introduces a modified version for which the ROS problem is much harder to solve.
The ROS problem is a recent research topic. Recently a paper about the (in)security of ROS was published. [Ben+20] The scheme introduced in the next section 2.2.4 is considered secure in 2021. It is important to keep in mind that the ROS problem is much newer and there is open research done.

Clause Blind Schnorr Signature Scheme

The Clause Blind Schnorr Signature Scheme is a modification of the Blind Schnorr Signature Scheme for which the ROS problem is harder to solve. The Clause Blind Schnorr Signature Scheme does this by choosing two random values r_0, r_1 and calculating $R_0 := r_0G; R_1 := r_1G$. The user generates the blinding factors twice $\alpha_0, \alpha_1, \beta_0, \beta_1 \leftarrow random \in \mathbb{Z}_p$. The user then calculates the challenges as before $c_0' := H(R_0', m); c_0 := c_0' + \beta_0 \mod p$ and $c_1' := H(R_1', m); c_1 := c_1' + \beta_1 \mod p$. After the signer receives the two challenges c_0 and c_1, the signer randomly chooses $b \leftarrow random\{0,1\}$ and calculates only s_b as in $s := r_b + c_b x \mod p$. The User receives s, b and can unblind the signature to receive his signature $\sigma := \langle R_b', s_b' \rangle$. The verification algorithm remains the same for Clause Blind Schnorr Signature Scheme. Figure 2.7 shows the Clause Blind Schnorr Signature Scheme. More details about the scheme can be found in the paper *Blind Schnorr Signatures and Signed ElGamal Encryption in the Algebraic Group Model* [FPS19].

Blindness Clause Blind Schnorr Signatures also achieve *perfect blindness* as in Schnorr Blind Signatures (see subsubsection 2.2.4). [FPS19]

2.2.5. Diffie Hellman Key Exchange

As source for this section, pages 383-386 were used in *Cryptography made Simple* [Sma16]. The Diffie-Hellman key exchange is a well proofed and well understood key exchange mechanism. DHKE[18] relies mainly on the Discrete Logarithm Problem. DHKE is used for key exchange in many protocols today (e.g. TLS cipher suites).

[17] Random inhomogeneities in an Overdetermined, Solvable system of linear equations
[18] Diffie-Hellman key exchange

User		Signer
knows:	public parameters:	knows:
public key X	$\langle p, \mathbb{G}, G, H \rangle$	private signing key $x, X := xG$
		$r_0, r_1 \leftarrow random \in \mathbb{Z}_p$
		$R_0 := r_0 G$
		$R_1 := r_1 G$
	$\xleftarrow{R_0,R_1}$	
$\alpha_0, \alpha_1, \beta_0, \beta_1 \leftarrow random \in \mathbb{Z}_p$		
$R'_0 := R_0 + \alpha_0 G + \beta_0 X$		
$R'_1 := R_1 + \alpha_1 G + \beta_1 X$		
$c'_0 := H(R'_0, m)$		
$c'_1 := H(R'_1, m)$		
$c_0 := c'_0 + \beta_0 \mod p$		
$c_1 := c'_1 + \beta_1 \mod p$		
	$\xrightarrow{c_0,c_1}$	
		$b \leftarrow random \in \{0, 1\}$
		$s := r_b + c_b x \mod p$
	$\xleftarrow{b,s}$	
check $sG = R + cX$		
$s' := s + \alpha_b \mod p$		
$\sigma := \langle R'_b, s' \rangle$		

Figure 2.7.: The Clause Schnorr Blind Signature Scheme

Hardness Assumptions

As already stated, the DHKE relies on the assumption that calculating the discrete logarithm is hard. The DLP is in G, where G is a finite abelian group of prime order q. This could either be a subgroup of the multiplicative group of a finite field or the set of points on an elliptic curve over a finite field. Given $g, h \in G$, find x such that $g^x = h$.

Further, CDH[19] and DDH[20] are important hardness assumption, which can be reduced to the DLP. Hardness assumptions are introduced very briefly. In this work we believe that these well proofed and well tested hardness assumptions hold. (See Chapter 3.1 *Cryptography made Simple* [Sma16] for more details on DH hardness assumptions.)

Protocol

Alice and Bob want to securely exchange a key with DHKE. Alice has a private key a and a corresponding public key $A = g^a \mod p$. Bob has a private key b and a corresponding public key $B = g^b \mod p$. With elliptic curves, the private key is a multiplication factor for a base point g (see example on page 385 *Cryptography made Simple* [Sma16]).

Alice now sends her public key A to Bob. Bob can then calculate $k = A^b \mod p = g^{ab} \mod p$ and sends his public key B to Alice. Alice can then calculate $k = B^a \mod p = g^{ab} \mod p$. Both get the same key k as result of the key exchange. Note: This protocol on its own is not an authenticated key exchange, which means that Man-in-the-Middle attacks are possible.

A different way of looking at DHKE is by thinking of a lock which can be unlocked by two (private) keys. If one of the two private keys are known, one could calculate k on its own. Taler's refresh protocol (see 3.2.3) uses DHKE in a very interesting way.

2.2.6. Cut and Choose Protocol

A good introduction to cut and choose protocols gives the Paper from Claude Crépeau ([Cré] References to the important examples can be found in the paper.):

"A cut and choose protocol is a two-party protocol in which one party tries to convince another party that some data he sent to the former was honestly constructed according to an agreed upon method. Important examples of cut-and-choose protocols are interactive proofs, interactive arguments, zero-knowledge protocols, witness indistinguishable and witness hiding protocols for proving knowledge of a piece of information that is computationally hard to find. Such a protocol usually carries a small probability that it is successful despite the fact that the desired property is not satisfied.

...

The expression cut-and-choose was later introduced by David Chaum in analogy to a popular cake sharing problem: Given a complete cake to be shared among two parties distrusting of each other (for reasons of serious appetite). A fair way for them to share the

[19]Computational Diffie-Hellman
[20]Decisional Diffie-Hellman

cake is to have one of them cut the cake in two equals hares, and let the other one choose his favourite share. This solution guarantes that it is in the formers best interest to cut the shares as evenly as possible."

Talers cut and choose protocol is *zero knowledge*, which means that nothing about the secret is learned. The cut and choose protocol used in Taler is explained further when the refresh protocol is discussed (see 3.2.3).

2.3. Taler Protocols

In section 2.1 a brief overview of how GNU Taler works is given. All the relevant preliminaries are covered in section 2.2. In this section a closer look at the different protocols is taken.

2.3.1. Withdrawal Protocol

The withdrawal protocol is described in chapter 4.7.2 of [Dol19]. Before coins can be withdrawn, the customer generates a reserve key pair $w_s, W_p \leftarrow Ed25519.KeyGen()$. He then transfers a certain amount of money from his bank to the exchange's bank via wire transfer. This payment must include the reserve public key W_p. The customer will later authorize withdrawals with a signature using his private reserve key. As soon as the exchange has received the payment, the withdrawal for coins with a value i can begin (described in figure 2.9).

At this stage the client knows the reserve private key and the public denomination key. The customer can then create coins up to the amount included in the wire transfer. The coin creation and blind signatures are described in section 2.2.4. So the client generates a planchet (an Ed25519 key pair) and blinds it. This blinded planchet is then signed by the customers private reserve key, to prove that the customer is eligible to withdraw the coin. The exchange who receives the blinded planchet and the signature first checks whether the signature is valid with the public reserve key sent with the wire transfer. When successful, the exchange blindly signs the planchet, returns the signature and notes the amount withdrawn of the reserve. The customer unblinds the signature, checks its validity and persists the coin. The state machine of a coin can be seen in figure 2.8.

Withdraw Loophole

The withdraw loophole allows withdraw operations where owner of the resulting coins isn't the owner of the reserve that the coins where withdrawn from. It is used for tipping (described in section 2.3.6) and can therefore be seen as a feature.

By misusing the withdraw loophole, untaxed and untraceable payments can be performed. Figure 2.10 explains how such a payment would work. Note that we omitted the parts leading up to the coin creation (contract, agreement of price, number of coins and their denominations). This is how it works on a high level:

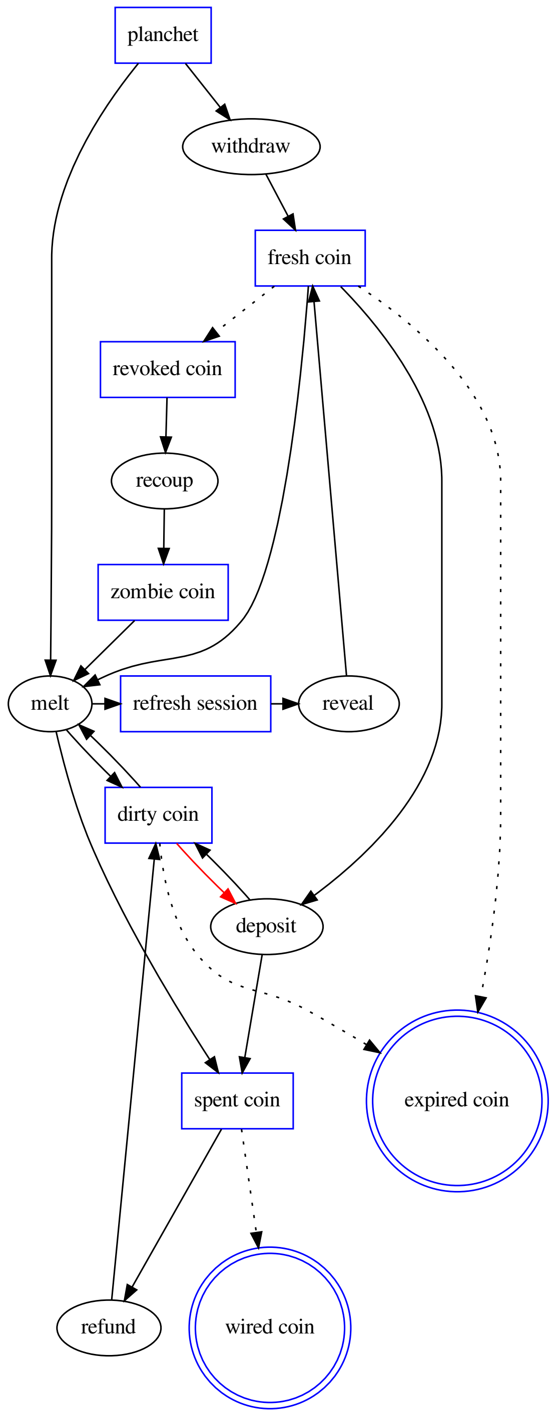

Figure 2.8.: State machine of a coin (source: [Tal22a])

Customer		Exchange
knows:		knows:
reserve keys w_s, W_p		reserve public key W_p
denomination public key $D_p = e, N$		denomination keys d_s, D_p
generate coin key pair:		
$c_s, C_p \leftarrow Ed25519.KeyGen()$		
blind:		
$r \leftarrow random \in \mathbb{Z}_N^*$		
$m' = \text{FDH}(N, C_p) * r^e \mod N$		
sign with reserve private key:		
$\rho_W = D_p, m'$		
$\sigma_W = \text{Ed25519.Sign}(w_s, \rho_W)$		
	$\xrightarrow{\rho=W_p,\sigma_W,\rho_W}$	
		verify if denomination public key is valid
		check Ed25519.Verify(W_p, ρ_W, σ_W)
		decrease balance if sufficient
		sign:
		$\sigma'_c = (m')^{d_s} \mod N$
	$\xleftarrow{\sigma'_c}$	
unblind:		
$\sigma_c = \sigma'_c * r^{-1}$		
verify signature:		
check $\sigma_c^e = \text{FDH}(N, C_p)$		
resulting coin: c_s, C_p, σ_c, D_p		

Figure 2.9.: Withdrawal process

1. The malicous merchant generates and blinds coins, which are then transmitted to the customer
2. The customer authorizes the withdraw from his reserve by signing the blinded coins with the private key of his reserve, thus generating withdraw confirmations.
3. The withdraw confirmations are transmitted to the exchange, which generates the signatures and returns them to the malicous merchant.
4. The malicous merchant unblinds the signatures. He is now in possession of the coin, thus the payment is completed.

2.3.2. Payment Process

The payment process is divided in two steps described by the spend and deposit protocols. Details about the payment process can be found in multiple chapters in [Dol19]: Chapter 4.7.3 describes the spend and deposit protocols. Chapter 4.1.4 describes more general aspects as well as the contract header and deposit permission structure and details.
On a high level, payment works like this:

1. The customer submits a shopping cart (one or more items to buy) and commits his intent to buy them.
2. The merchant puts together the contract terms containing the necessary information for payment, signs it and sends both to the customer (spend protocol).
3. The customer generates a deposit permission and its signature for each coin used in the transaction (spend protocol).
4. The customer forwards the deposit permission(s) to the merchant (spend protocol). If the deposit protocol is performed by the customer, this step can be skipped.
5. Either the customer or the merchant sends the deposit permission(s) to the exchange (deposit protocol).
6. The exchange processes the deposit permission and returns a deposition confirmation when successful (deposit protocol).
7. If the deposit protocol was performed by the customer, the deposit confirmation(s) have to be forwarded to the merchant.

Spend Protocol

The payment process begins when a customer submits a shopping cart (one or more items to buy) and commits his intent to buy them. The merchant has a key pair skM, pkM of which the customer knows the public key. Note that certain details contained in contract header or deposit permission like merchant KYC information, deposit and refund deadlines and fees are left out. The deposit state machine can be seen in figure 2.11.

Customer
knows:
reserve keys w_s, W_p
denomination public key $D_p = \langle e, N \rangle$

malicous Merchant
knows:

denomination public key $D_p = \langle e, N \rangle$

generate coin key pair:
$c_s, C_p \leftarrow$ Ed25519.KeyGen()
blind:
$r \leftarrow random \in \mathbb{Z}_N^*$
$m' := \text{FDH}(N, C_p) * r^e \mod N$

$\xleftarrow{m'}$

sign with reserve private key:
$\rho_W := \langle D_p, m' \rangle$
$\sigma_W :=$ Ed25519.Sign(w_s, ρ_W)

$\xrightarrow{\langle W_p, \sigma_W, \rho_W \rangle}$

---

malicous Merchant
knows:

denomination public key $D_p = \langle e, N \rangle$

Exchange
knows:
reserve public key W_p
denomination keys d_s, D_p

$\xrightarrow{\langle W_p, \sigma_W, \rho_W \rangle}$

$\langle D_p, m' \rangle := \rho_W$
verify if D_p is valid
check Ed25519.Verify(W_p, ρ_W, σ_W)
decrease balance if sufficient
sign:
$\sigma'_c := (m')^{d_s} \mod N$

$\xleftarrow{\sigma'_c}$

unblind:
$\sigma_c := \sigma'_c * r^{-1}$
verify signature:
check if $\sigma_c = \text{FDH}(N, C_p)$

resulting coin: $\langle c_s, C_p, \sigma_c, D_p \rangle$

Figure 2.10.: Untaxed payment using the withdraw loophole

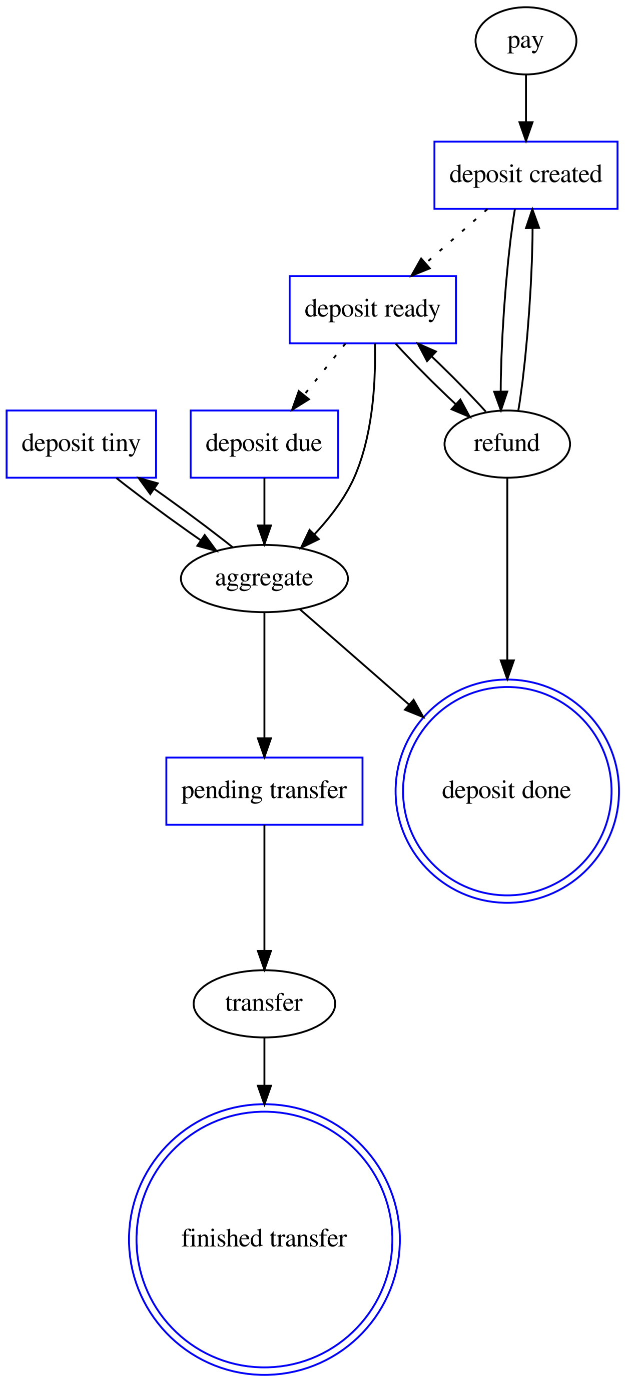

Figure 2.11.: State machine of a deposit (source: [Tal22b])

1. The merchant puts together the following information (without transmitting them) and requests payment:
   - price v
   - exchange E_m (multiple possible)
   - account A_m at the exchange E_m
   - info (free form details containing the full contract)
2. The customer generates an Ed25519 claim key pair p_s, P_p and submits the public key to the merchant. This key can be used by the customer to prove that he didn't copy contract terms from another customer.
3. The merchant puts together the contract terms ρ and signs it with skM, resulting in the signature σ_P.
   The contract terms contains:
   - E_m (exchange)
   - A_m (account at exchange E_m)
   - pkM
   - Hash(v, info)
   - P_p

   ρ_P (contract terms), σ_P (contract terms signature), v (price) and info are submitted to the customer.
4. The customer does the following checks:
   - Is the signature of the contract terms correct?
   - Is the public key referenced in the contract terms the same as the one generated in step 2?
   - Is the hash of price and info the same as the one in the contract terms?

   If all checks are successful, the customer chooses one or more coins to be spent. For each coin, a deposit permission ρ_D and its signature σ_D is generated. The deposit permission contains the following information:
   - Coin public key C_p
   - Coin denomination public key pkD
   - Coin signature σ_C
   - Value to be spent for this coin f (greater than zero, up to the residual value of the coin)
   - Hash of the contract terms ρ_P
   - Account of merchant A_m (at exchange E_m)
   - Merchant public key pkM

The list of deposit permissions and their signatures is transferred to the merchant who then executed the deposit protocol. Note that the customer is also able to deposit the coins (instead of the merchant), this is used in cases where the merchant doesn't have an internet connection, but the customer does. This can be useful in cases where the merchant becomes unresponsive. The customer can prove that he paid in time.

5. The merchant receives the deposit permissions and signatures and uses the deposit protocol to execute the payment.

Before we continue with the deposit protocol, there are a few interesting details to point out (described in [Dol19] section 4.1.4):

- The contract terms and the deposit permission are JSON[21] objects.
- The contract terms only contains a cryptographic hash of the contract. This improves privacy since the exchange doesn't have to know the full contract details, but still makes it possible to identify the contract in case of a dispute or some form of auditing.
- At the point where the merchant completes step three (submits the contract terms and its signature) to the customer, the customer is able to finish the transaction using the deposit protocol without interaction of the merchant. This means that the merchant at this step must be able to fulfill the contract if the customer completes the payment process.

Deposit Protocol

As previously mentioned, both parties (customer and merchant) are able to run the deposit protocol. In the following description, the term merchant will be used, but could be replaced by customer. In cases where there are multiple deposit permissions (meaning that multiple coins are used to pay), the deposit protocol is run separately for each deposit permission.

1. The merchant submits the deposit permission and its signature to the exchange.
2. The exchange runs these checks:
   - Is the denomination public key referenced in the deposit permission valid (issued by the exchange, lifetime between start and deposit/refresh expiration, not revoked)?
   - Is the deposit permission signature σ_D a correct signature of the deposit permission ρ_D with the Ed25519 coin public key C_p referenced in the deposit permission?

[21]JavaScript Object Notation

- Is there a processed deposit recorded in the exchanges databases based on coin public key and contract terms hash (replay/double spending)? If not, continue with the next check since this is correct and expected behavior.
  If there is, does the recorded deposit permission equal the one we're currently checking? If this is the case, further checks can be skipped and the deposit confirmation signature can be returned to the customer. If not, the process should be terminated because there's something wrong with the transaction.
- Is the signature of the coin valid?
- Is f (the value to be spent) smaller or equal the residual value of the coin (check for overspending attempt)?

If all checks are successful, the exchange saves the deposit record containing the deposit permission and its signature in a database, substracts the spent value from the residual value of the coin and schedules the money transfer to the merchant's account A_m (grouping payments is done to reduce payment fees).
The exchange calculates a deposit confirmation signature σ_{DC} for the deposit permission with the exchange signing private key and returns them to the merchant.
This signature is also used to prove that a merchant was the first to receive payment from a certain coin. Without this, an evil exchange could later deny confirming a payment and claim double spending. With the signature, the merchant can prove that the payment was confirmed by the exchange, thus delegating the responsibility (and potential financial loss) for double spending detection to the exchange.

3. The merchant checks the signatures of the deposit confirmations with the exchange signing public key.

It may happen that a payment gets stuck as partially complete, for example when a backup of a wallet is restored and one coin or more have already been spent ([Dol19] chapter 4.1.4). In this case, the customer can retry the payment with a different coin. If this isn't possible, the payment can be refunded (assuming refunds were enabled for this payment). Other scenarios were described in Dold's thesis, but dismissed due to privacy concerns. This means that disputes have to be settled aside from Taler when a customer isn't able to fully pay and refunds are disabled.

Web Payment Scenarios

The following methods are Taler-native methods for paying and payment validation. They are not identity-based, meaning that they do not require a login or similar techniques. Note that other methods could be implemented depending on the scenario.

- **Resource-based web payment** ([Dol19] chapter 4.1.5): All Taler contract terms contain a fulfillment URL. This can either be a direct link to a digital product (like a movie, a song or a document), or to a confirmation page. When a browser opens a fulfillment URL for a resource that hasn't yet been paid for, the merchant requests payment. The wallet then generates and submits a claim key pair, thus claiming the

contract, which then can be paid (if the user accepts the contract). The browser can then retry to navigate to the fulfillment URL, this time submitting the contract order ID as parameter, which the merchant can check if it has been paid (and deliver the content if this is the case). This is known as the extended fulfillment URL
The wallet stores fulfillment URLs and their associated contracts. Upon receiving a payment request, the wallet searches the stored fulfillment URLs and if it found one, automatically forwards the user to the extended fulfillment URL containing the contract.

- **Session-bound payments and sharing** ([Dol19] chapter 4.1.6): So far, validating payment is done using the extended fulfillment URL. The problem with this approach is that this URL can be shared, which is a problem for digital content. To make this more difficult, the seller's website assigns the user a session ID (for example using a session cookie) and extends the extended fulfillment URL with a session signature parameter. This parameter can be used by the merchant to check if the user paid for the resource or replayed the payment in this session.

- **Embedded content** ([Dol19] chapter 4.1.7): When paying to access multiple resources behind a paywall (instead of just one resource), the previously described methods do not work. Dold's thesis suggest two methods:
    1. A session cookie can be set by accessing the fulfillment URL. When the browser requests a subresource, the merchant can verify the session cookie.
    2. In this scenario, the fulfillment URL would show the resources behind the paywall. Upon opening the extended fulfillment URL, the merchant's website would add an authentication token to the URLs of the subresources. When accessing a subresource, the merchant can check the authentication tokens validity.

2.3.3. Refresh Protocol

This section provides a description of the refresh protocol. The technical details can be found in 4.7.4 [Dol19]. All relevant preliminaries needed to understand the technical details were already introduced in this work.

Introduction

A protocol to refresh coins is needed for many reasons. One important reason is giving change. Similar to the real world, there are often situations where one does not have the exact amount of coins. A change protocol therefore provides a lot of convenience for the users. Without such a mechanism it would be quite hard to use.
Giving change is not trivial, since AML and CFT compliance still needs to hold. On the other side, the change still needs to provide privacy for the customer. Thus, the change must be unlinkable to the previous (or any) transaction.
Complying with AML and CFT while preserving the customer's anonymity may sound like

a contradiction at first. However, Taler has a clever way to solve this problem with the refresh protocol.

The general idea is that the new coin can be derived from the private key of the old coin.

DH Lock

DHKE was introduced in section 2.2.5. Taler uses ECDH[22] as a lock with two possible keys to unlock the shared key. To create such a lock, one creates two key pairs $C = cG$ and $T = tG$. To unlock now means calculating k. Both private keys, c and t are now able to calculate $k = tC = t(cG) = c(tG) = cT$ and thus can unlock the lock. This k can then be used to derive the private key of the new coin and the corresponding blinding factor.

Customer Setup

The customer, which holds the old partially spend coin and knows
C_{old} = Ed25519.GetPub(c_{old}). A transfer key T = Ed25519.GetPub(t) is then (randomly) generated by the customer.
The key pairs T = Ed25519.GetPub(t) and C_{old} = Ed25519.GetPub(c_{old}) form the lock with two keys that was introduced before. The customer then creates $x = c_{old}, T = tC_{old}$ and derives c_{new}, the private key of the new coin and b_{new} the blinding factor of the new key. As usual the customer calculates the coins public key C_{new} = Ed25519.GetPub(c_{new}), hashes the new coin with FDH f_{new} = FDH(C_{new}) and blinds the hash $f'_{new} = f_{new} b^e_{new}$. The f'_{new} is then transmitted to the exchange.
Figure 2.12 shows how the new coin is derived as explained above.

```
RefreshDerive$(s, \langle e, N \rangle, C_p)$
$t :=$ HKDF$(256, s, "t")$
$T :=$ Curve25519.GetPub$(t)$
$x :=$ ECDH-EC$(t, C_p)$
$r :=$ SelectSeeded$(x, \mathbb{Z}^*_N)$
$c'_s :=$ HKDF$(256, x, "c")$
$C'_p :=$ Ed25519.GetPub$(c'_s)$
$\overline{m} := r^e * C'_p \mod N$
**return** $\langle t, T, x, c'_s, C'_p, \overline{m} \rangle$
```

Figure 2.12.: The RefreshDerive derives a new coin from a dirty coin with a seed. The DH-Lock is used to create the link used in the linking protocol

Now with the DH Lock the person who is in possession of the old key can always recalculate and thus spend the new coin (as long as it knows the public transfer key T). However, there is one last thing: How does the exchange know that the old key is linked to the new

[22]Elliptic Curve Diffie Hellman

one? To comply with AML and CFT, the exchange wants to ensure that the person who created the new coin is also in possession of the old coin. A link needs to be created in a way that nobody can link the old coin to the new coin, except the person in possession of the old coin. The person in possession of the old coin needs to proof to the exchange that this link was created without revealing the link. This problem is solved with the cut and choose protocol in the next section.

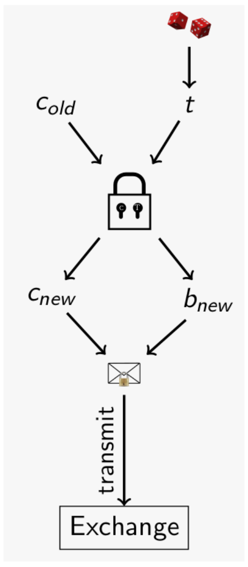

Figure 2.13.: Taler refresh protocol, transfer key setup (source: [Tal21d])

Cut & Choose

Instead of doing the customer setup once, it is done n times. The customer generates n different transfer keys $t_1, t_2 \ldots t_n$. For each key the whole calculations are done and all the blinded coins $f'_1, f'_2 \ldots f'_n$ are sent to the exchange together with the old coins public key and signature.
The exchange responds with a randomly picked number from 1 to n. The customer has to reveal all the transfer keys, **except the one picked by the exchange.** The exchange makes the same calculations with the revealed private transfer keys (without knowing the private key c_{old}). The exchange can now verify whether the customer was honest or not. A evil customer could create a new coin which is not linked to the old coin (without the DH lock). Such attacks will be detected with a high probability in this protocol. Since the t_x picked by the exchange is not checked, an evil customer can win this with a probability of $1/n$. Already with $n = 3$ an attack is not in the customers interest due to economic reasons. In 2 out of 3 cases the exchange would detect the attack and would keep the money and the customer would have lost it. The probability can be adjusted with n. With increasing size of n the attack becomes even less attractive. When the cut and choose protocol ended successfully, the value of the old coin is set to zero.

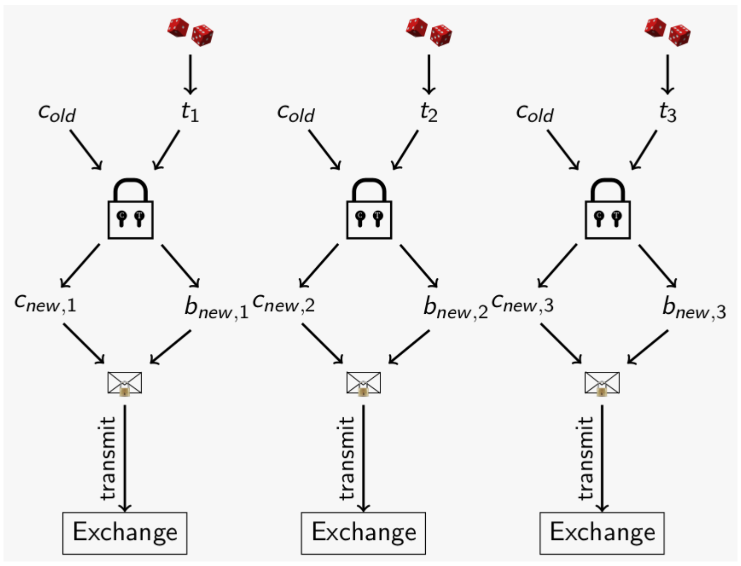

Figure 2.14.: Taler refresh protocol, cut and choose (source: [Tal21d])

2.3.4. Commit Phase

The refresh protocol is implemented in two phases. The commit phases creates k derives and commits to this values by calculating a hash over the derives. On the exchange's side various checks are done to validate the request. Detailed steps of the commit phase are shown in figure 2.15.

```
Customer
knows:
denomination public key $D_{p(i)}$
$\text{coin}_0 = \langle D_{p(0)}, c_s^{(0)}, C_p^{(0)}, \sigma_c^{(0)} \rangle$
$\text{Select}\langle N_t, e_t \rangle := D_{p(t)} \in D_{p(i)}$
for $i = 1, \dots, \kappa$ :
    $s_i \to \{0,1\}^{256}$
    $X_i := \text{RefreshDerive}(s_i, D_{p(t)}, C_p^{(0)})$
    $(t_i, T_i, x_i, c_s^{(i)}, C_p^{(i)}, \overline{m}_i) := X_i$
endfor
$h_T := H(T_1, \dots, T_k)$
$h_{\overline{m}} := H(\overline{m}_1, \dots, \overline{m}_k)$
$h_C := H(h_t, h_{\overline{m}})$
$\rho_{RC} := \langle h_C, D_{p(t)}, D_{p(0)}, C_p^{(0)}, \sigma_C^{(0)} \rangle$
$\sigma_{RC} := \text{Ed25519.Sign}(c_s^{(0), \rho_{RC}})$
Persist refresh-request$\langle \rho_{RC}, \sigma_{RC} \rangle$
```

$$\xrightarrow{\rho_{RC}, \sigma_{RC}}$$

```
Exchange
knows:
denomination keys $d_{s(i)}, D_{p(i)}$

$(h_C, D_{p(t)}, D_{p(0)}, C_p^{(0)}, \sigma_C^{(0)} = \rho_{RC})$
check Ed25519.Verify$(C_p^{(0)}, \sigma_{RC}, \rho_{RC})$
$x \to \text{GetOldRefresh}(\rho_{RC})$
Comment: GetOldRefresh $(\rho_{RC} \mapsto \{\bot, \gamma\})$

if $x = \bot$
    $v := D(D_{p(t)})$
    $\langle e_0, N_0 \rangle := D_{p(0)}$
    check IsOverspending$(C_p^{(0)}, D_{p(0)}, v)$
    check $D_{p(t)} \in \{D_{p(i)}\}$
    check FDH$(N_0, C_p^{(0)}) \equiv_{N_0} (\sigma_0^{(0)})^{e_0}$
    MarkFractionalSpend$(C_p^{(0)}, v)$
    $\gamma \leftarrow \{1, \dots, \kappa\}$
    Persist refresh-record $\langle \rho_{RC}, \gamma \rangle$
else
    $\gamma := x$
endif
```

$$\xleftarrow{\gamma}$$

Continued in figure 2.16

Figure 2.15.: Refresh protocol (commit phase)

Customer | Exchange

Continuation of figure 2.15

$\xleftarrow{\gamma}$

check IsConsistentChallenge(ρ_{RC}, γ)
Comment: IsConsistentChallenge
$(\rho_{RC}, \gamma) \mapsto \{\perp, \top\}$

Persist refresh-challenge$\langle \rho_{RC}, \gamma \rangle$
$S := \langle s_1, \dots, s_{\gamma-1}, s_{\gamma+1}, \dots, s_x \rangle$
$\rho_L = \langle C_p^{(0)}, D_{p(t)}, T_\gamma, \overline{m}_\gamma \rangle$
$\rho_{RR} = \langle T_\gamma, \overline{m}_\gamma, S \rangle$
$\sigma_L =$ Ed25519.Sign$(c_s^{(0)}, \rho_L)$

$\xrightarrow{\rho_{RR}, \rho_L, \sigma_L}$

$\langle T'_\gamma, \overline{m}'_\gamma, S \rangle := \rho_{RR}$
$\langle s_1, \dots, s_{\gamma-1}, s_{\gamma+1}, \dots, s_\kappa \rangle) := S$
check Ed25519.Verify$(C_p^{(0)}, \sigma_L, \rho_L)$
for $i = 1, \dots, \gamma - 1, \gamma + 1, \dots, \kappa$
$X_i :=$ RefreshDerive$(s_i, D_{p(t)}, C_p^{(0)})$
$\langle t_i, T_i, x_i, c_s^{(i)}, C_p^{(i)}, \overline{m}_i \rangle := X_i$
endfor
$h'_T = H(T_1, \dots, T_{\gamma-1}, T'_\gamma, T_{\gamma+1}, \dots, T_\kappa)$
$h'_{\overline{m}} = H(\overline{m}_1, \dots, \overline{m}_{\gamma-1}, \overline{m}'_\gamma, \overline{m}_{\gamma+1}, \dots, \overline{m}_\kappa)$
$h'_C = H(h'_T, h'_{\overline{m}})$
check $h_C = h'_C$
$\overline{\sigma}_C^{(\gamma)} := \overline{m}^{d_{s(t)}}$

$\xleftarrow{\overline{\sigma}_C^{(\gamma)}}$

$\sigma_C^{(\gamma)} := r^{-1} \overline{\sigma}_C^{(\gamma)}$
check $(\sigma_C^{(\gamma)})^{e_t} \equiv_{N_t} C_p^{(\gamma)}$
Persist coin$\langle D_{p(t)}, c_s^{(\gamma)}, C_p^{(\gamma)}, \sigma_C^{(\gamma)} \rangle$

Figure 2.16.: Refresh protocol (reveal phase)

2.3.5. Reveal Phase

In the reveal phase the customer receives γ and he reveals the all the seeds to the exchange, except for s_γ. The exchange can then verify if the customer was honest with probability $1/k$. On success the exchange will return the blinded signature of the new coin and the customer can then unblind and store the coin. The reveal phase is described in figure 2.16

(Un)linkability

The goal of the cut and choose protocol is to ensure with a high probability $(1/n)$ that the customer honestly created the new coin. It ensures that the old coin is linked to the new coin via the DH lock.

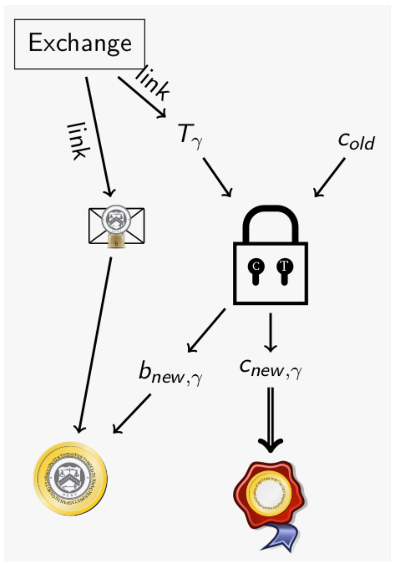

Figure 2.17.: Taler refresh protocol, linkability (source: [Tal21d])

With that, the following attack scenario is prevented (with probability $1/n$):
An third party creates the new coin without the DH lock as described in section 2.2.4. The third party sends the blinded new coin to the customer (who possesses the old coin). The customer then signs the new coin by the exchange and sends the blinded signature back to the third party. The third party would then be in possession of a valid new coin, which is not linked to the old coin. As mentioned, such an attack is detected with a high probability by the exchange with the cut and choose protocol described earlier.

We will now consider the following attack scenario:
Someone could give the private key of the old coin c_{old} to another person. This other person then can derive a new coin using the refresh protocol. The original customer currently can not recreate the new coin with only the knowledge of the old coins private key c_{old}. He would need to know the public key of the transfer key T_x and also the blinded signature of the new coin f'_{new}. For this reason the exchange exposes the public transfer key T_x and the blinded new coin f'_{new} for a given old coin C_{old}. So anybody who knows the public key of the old coin could ask for the public transfer key and the blinded signature of the new coin. Only a person in possession of the old coins private key c_{old} can recreate the new coin's private key.
This mechanism can not be abused for money laundering anymore, since the original customer could trick this third person and spend the coin faster. The linking protocol is described in figure 3.7.

2.3.6. Tipping Protocol

Source for this protocol was section 4.1.10 from [Dol19].
Merchants can give customers a small tip by using the withdraw loophole (described in

Customer
knows:
$\text{coin}_0 = \langle D_{p(0)}, c_s^{(0)}, C_p^{(0)}, \sigma_C^{(0)} \rangle$

Exchange
knows:

$$\xrightarrow{C_{p(0)}}$$

$L := \text{LookupLink}(C_{p(0)})$
Comment: $\text{LookupLink}(C_p) \mapsto \{\langle \rho_L^{(i)},$
$\sigma_L^{(i)}, \overline{\sigma}_C^{(i)} \rangle\}$

$$\xleftarrow{L}$$

for $\langle \rho_L^{(i)}, \overline{\sigma}_L^{(i)}, \sigma_C^{(i)} \rangle \in L$
$\langle \hat{C}_p^{(i)}, D_{p(t)}^{(i)}, T_\gamma^{(i)}, \overline{m}_\gamma^{(i)} \rangle := \rho_L^{(i)}$
$\langle e_t^{(i)}, N_t^{(i)} \rangle := D_{p(t)}^{(i)}$
check $\hat{C}_p^{(i)} \equiv C_p^{(0)}$
check $\text{Ed25519.Verify}(C_p^{(0)}, \rho_L^{(i)}, \sigma_L^{(i)})$
$x_i := \text{ECDH}(c_s^{(0)}, T_\gamma^{(i)})$
$r_i := \text{SelectSeeded}(x_i, \mathbb{Z}_{N_t}^*)$
$c_s^{(i)} := \text{HKDF}(256, x_i, \text{"}c\text{"})$
$C_p^{(i)} := \text{Ed25519.GetPub}(c_s^{(i)})$
$\sigma_C^{(i)} := (r_i)^{-1} \cdot \overline{m}_\gamma^{(i)}$
check $(\sigma_C^{(i)})^{e_t^{(i)}} \equiv_{N_t^{(i)}} C_p^{(i)}$
(Re-)obtain coin$\langle D_{p(t)}^{(i)}, c_s^{(i)}, C_p^{(i)}, \sigma_C^{(i)} \rangle$

Figure 2.18.: Linking protocol

section 2.3.1). This can be for a variety of different reasons, for example for submitting a survey. The merchant needs to create a reserve with the exchange. The reserve keys is now used to sign blinded coins generated by the user.

1. The Merchant triggers the Payment required response with the Taler-Tip header set
2. The taler tip header contains information like amount, exchange to use, deadline and more. (details section 4.1.10 [Dol19])
3. The customer creates planchets that sum up the amount and blinds the token with the denomination key of the specified exchange and sends the blinded planchets to the merchant.
4. The merchant creates withdrawal confirmations (by signing them with the reserver private key) for these planchets and responds with a list of signatures.
5. The customer then uses these signatures to create coins as in the withdrawal protocol

The received coins are still anonymized and only spendable by the customer.

2.3.7. Refund Protocol

A merchant can undo a deposit on a coin by signing a refund permission. The protocol details can be found in section 4.7.5 of [Dol19]. Since a refund is mainly done by the merchant, to provide refunds a merchant need to support refunds. A refund can be either fully or partially. After a refund, the customer is able to spend the coin, but it should be refreshed first to prevent linking of transactions. The refund deadline is specified in the contract header, after the deadline the exchange makes a wire transfer with the money to the merchants bank. There is a refund fee, which is subtracted from the remaining coin value. This also prevents denial of service attacks, or at least makes them economically uninteresting. There exists automatic refunds when a payment only partially succeeds for many reasons. Refunds are also an important business case for many merchants who want to provide a convenient experience. A merchant can for example provide a refund when the customer is not happy with the product. Such a refund can be made by the merchant with a signature without the customers consent. Now should be clear what the purpose of a refund protocol is, the rest of this section will look at the refund protocol.

In the protocol the customer requests a refund from the merchant. If the merchant accepts the request, it authorizes the exchange to apply the refund.

1. The customer asks for a refund for payment p with reason m
2. The merchant decides whether it accepts the refund or not according to the merchants business rules.
3. If accepted, the merchant signs the refund permission with the merchants Ed25519 key and sends it to exchange.

4. The exchange checks the signature and refunds the coin(s) and sends a signed confirmation to the merchant.
5. The merchant sends the signed confirmation from the exchange to the customer.

2.4. Trust and PKI in Taler

In this section Taler's PKI[23] is explained and how Taler handles trust. This section is included due to the reason that we have to create Schnorr denomination keys to add the Clause Blind Schnorr Signature scheme to Taler. Taler uses TLS, however it does not rely on TLS for authenticity or integrity. (More detailed in chapter 4.1.3 of [Dol19])

Auditor

In Taler the auditors serves as trust anchor, and they are identified by a single Ed25519 public key. Similar to the list of trusted root CA[24] that come with web browsers and operating systems, a wallet comes with a list of trusted auditor certificates. In the rest of this section, different parts of Taler and how they are integrated in Taler's PKI are discussed. The section ends with a discussion about security risks of Taler's trust model. For details, refer to chapter 4.1.3 of [Dol19].

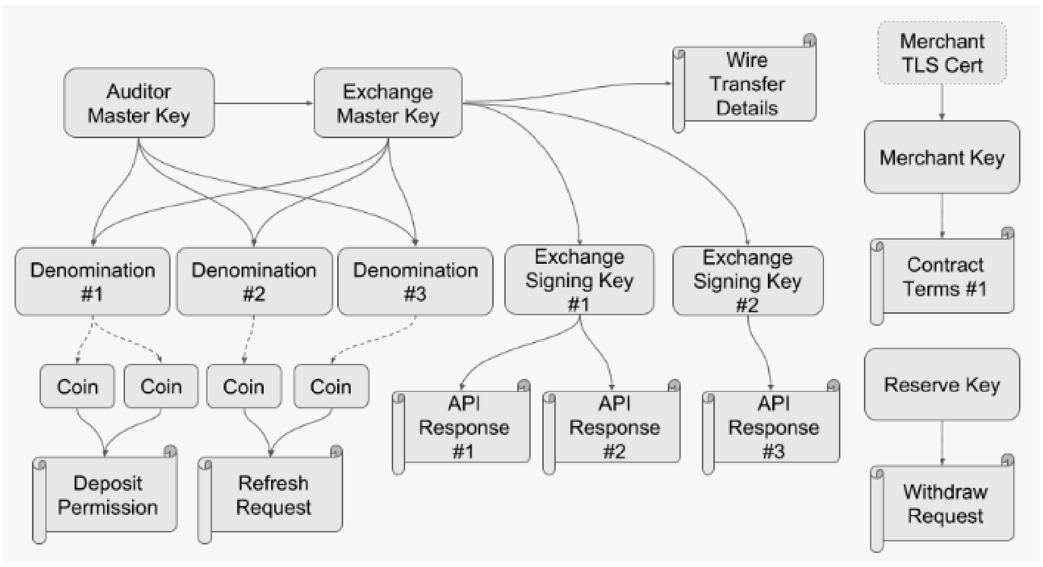

Figure 2.19.: GNU Taler PKI entities (source: [Dol19])

[23]Public Key Infrastructure
[24]Certificate Authority

Exchange

The exchange has to expose an API in order to enable customers (wallets), merchants and auditors to access keys and other information. An exchange has a long term master key (Ed25519 key) and a base URL. The URL and the long term MK[25] identifies an exchange. The MK is only used as an offline signing key and should be stored on an air-gapped machine. Further, the exchange has online signing keys (Ed25519 key), which are signed by the exchanges MK. This MK is on his side signed by one or possibly more auditors master key(s). The exchange's (online) signing keys are used to sign API responses. The denomination keys of an exchange are also signed by the exchanges offline MK and the auditors MK. The bank accounts supported by the exchange for withdrawals and deposits are also signed by the exchanges offline MK.

API requests are made to the base URL appending the name of the endpoint (eg. <base-url>/keys) The endpoint <base-url>/keys is used to get the exchanges signing keys and other information. Similar to the CA trust model, the client (customer or merchant) can validate the signature of the keys, with the list of trusted auditor certs.

Coins

As seen in the withdrawal protocol blind signatures are done with RSA public keys (section 2.2.4). These keys are called denomination keys and they represent the coin value of the signed coins. The following information concerning the denomination keys are signed by the exchanges master key (citation from [Dol19] chapter 4.1.3):

- The RSA public key
- The start date, after which coins of this denomination can be withdrawn and deposited.
- The withdraw expiration date, after which coins cannot be withdrawn anymore, must be after the start date.
- The deposit expiration date, after which coins cannot be deposited anymore, must be after the withdraw expiration date.
- The legal expiration date, after which the exchange can delete all records about operations with coins of this denominations, must be (typically quite a long time!) after the deposit expiration date.
- The fees for a withdraw, deposit, refresh and refund operation with this coin, respectively.

As mentioned, the denomination keys are signed by the exchanges MK and also by the auditor.

[25] Master Key

Merchant

The merchant has one Ed25519 public key. With that key the merchant authenticates to the exchange and signs responses to the customer. Depending on the jurisdiction, an exchange needs to comply to KYC regulations. A merchant which accepts payments from all exchanges (audited by a trusted auditor) therefore needs to fulfill KYC registration for all accepted exchange separately. This is needed to be legally compliant.
Like the customer, also the merchant is configured with a set of trusted auditors and exchanges. A merchant only accepts payments with coins of denominations from a trusted exchange which is audited by a trusted auditor.

For this reason Taler separates this service into an isolated service, similar to on-premise or external payment gateways, which are used by most e-commerce shops nowadays.

Customer

A customer has private keys of reserves that they own to authenticate with the exchange. The public key was communicated to the exchange with the wire transfer. (A bank however is not part of Taler's PKI.) A customer is therefore not registered with an exchange.

Further a customer possesses the private keys of his coins and stores them in a digital wallet.

Security Discussion

Taler's trust model is technically similar to the CA trust model we know from TLS certificates. The trust anchor lies with the auditors, whose certificates are pre-configured by the merchant or customer respectively. However, trust is always somehow attackable. That does not mean that there is a security issue in the trust model. When the list of trusted auditor certs of a customer/merchant somehow can be manipulated, the trust model breaks for this entity.
One attack scenario would be to attack customers/merchants with a supply-chain attack on the wallets or merchant backends' implementation. With software supply-chain attacks on the rise in 2020/21 (although the concept is not new) such an attack could have a big impact.
Since auditor certs are coupled with the wallet (or merchant) implementation, a bank, country, central bank or auditor will most likely publish a wallet and a merchant implementation for the corresponding Taler ecosystem.

3. Protocol Design

This chapter describes the necessary changes on the protocol level to implement a Blind Schnorr Signature Scheme to Taler.

3.1. Analysis of Current Protocols

The blind RSA signature scheme is only used for coin signatures. Note that we omitted protocols (or parts of them) where the coin signature is transmitted, but no other actions using it is performed.

An important property to mention here is *abort-idempotency*. Idempotence in the context of computer science is a property to ensure that the state of a system will not change, no matter how many times the same request was made. A more in-depth explanation is given within the cited source [Z21].
abort-idempotency goes a bit further. When the protocol is aborted at any stage, for example due to power cuts or network issues, the protocol still needs to ensure that the same response is sent for the same request. This is especially challenging when dealing with random values as we will see in the redesigned protocols in the following sections. For RSA Blind Signatures it is inherently easier to provide *abort-idempotency* since signature creation only needs one round-trip and requires less random values.

The following protocols currently use RSA Blind Signatures:

- **Withdraw Protocol:** The customer uses the blind signature scheme to blind the coins before transmitting them to the exchange, which blindly signs it (standard RSA signature) and the returns the signatures. After the customer receives the signatures, he unblinds and stores them together with the coins.
  Components:
  - Customer
  - Exchange
- **Deposit Protocol:** During the Deposit, the exchange verifies the coin signature derived using the blind RSA signature scheme.
  Components:
  - Exchange
- **Refresh Protocol:** The refresh protocol is used to derive a new coin from an old one which was partially spent. Parts of the protocol are similar to the withdraw protocol,

but it is more complex due to the added DH lock and cut-and-choose.
Components:

- Customer
- Exchange

- **Tipping:** Tipping is a variation of the withdraw protocol where the message containing the blinded planchets is transmitted to the merchant, who signs them using his reserve private, key and returns the signatures back to the customer. Here, the details from the withdraw protocol apply.
  Components:

  - Customer
  - Exchange

- **Recoup Protocol:** The recoup protocol distinguishes three different cases, which either use the refresh protocol or disclose either the withdraw transcript or refresh protocol transcript to the exchange.
  Components:

  - Customer
  - Exchange

3.2. Protocol Changes

The goal of the thesis is to add support for the Clause Blind Schnorr Signature scheme to Taler, besides the existing RSA Blind Signatures implementation (see section 2.2.4). For the design of the Clause Blind Schnorr Signatures the existing protocols with RSA Blind Signatures were redesigned.

The goal of the blind signature is to keep the exchange from knowing which coin a user withdraws and thus preventing the exchange linking a coin to a user. The biggest impact is on the withdrawal and refresh protocols, but all protocols that include some operation around denomination signatures are affected.

During the thesis the protocols will be redesigned, implemented and the differences to the current version will be outlined. These results will be delivered to the Taler team. Feedback is very important when (re)designing protocols. For that reason the redesigned protocols were discussed and reviewed with Christian Grothoff multiple times.

As signature scheme the Clause Blind Schnorr Signature Scheme described in section 2.2.4 was chosen for multiple reasons. First of all it is currently considered to be secure (see [FPS19]). Schnorr Signatures on Curve25519 are much shorter than RSA signatures. This should provide notable performance improvements in speed and storage, and therefore scales better. The paper describes a security analysis of the Blind Schnorr Signature scheme and introduces a modification (the "clause" part in the name) that is resistant to Wagner's algorithm (which solves ROS problem).

Curve25519 [Ber06] will be used for the implementation because it is a widely accepted curve (see [BL21], [RFC7748]) and is already used by Taler (Taler uses Ed25519 which is built upon Curve25519).

3.2.1. Withdraw Protocol

The modified protocol using the Clause Blind Schnorr Signature Scheme is described in figures 3.1 and 3.2.

The proposed change introduces an additional round trip. It must be prevented that the exchange has to track sessions or persist values during the first stage 3.1, while still ensuring abort-idempotency. In order to ensure *abort-idempotency*, the exchange has to generate the same R_0, R_1 for the same withdrawal request, while r_0, r_1 still needs to be unpredictable for the customer. For this reason a withdrawal-nonce combined with a HKDF comes into play. The redesigned protocol makes extensive use of HKDF's functionality as PRNG[1] and one-way function, thus random becomes *unpredictable*.

In the beginning of the protocol, the customer generates a coin key pair. Its private key is used to generate the withdraw-nonce n_w and the blinding factors $\alpha_0, \alpha_1, \beta_0, \beta_1$. The exchange uses the withdraw nonce together with the reserve key and a long-term secret to generate r_0, r_1. The coin and denomination private keys can be used as long-term secrets due to the one-way property of the HKDF.

Another question evolved around which key to use for the derivation of r_0, r_1. Obvious options are the denomination key or the exchange's online signing key. The denomination key was chosen because it has the recopu protocol in place that would handle coin recovery in case of a key compromise and subsequent revocation.

3.2.2. Deposit Protocol

The deposit protocol remains unchanged, except for the verification of the coin signature. To verify the signature, the exchange has to check if the following equation holds:

$$\begin{aligned} s'G &= R' + c'D_p \\ &= R' + H(R', C_p)D_p \end{aligned}$$

s', R' together form the signature, D_p is the denomination public key and C_p is the coin public key.

Further details regarding the verification process can be found in section 2.2.4.

3.2.3. Refresh Protocol

The refresh protocol blindly signs the new derived coins. The replacement of RSA Blind Signatures with the Clause Blind Schnorr Signature Scheme (see 2.2.4) makes the refresh protocol a bit more complex.

[1]Pseudo Random Number Generator

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:		
$\omega := randombytes(32)$		
persist $\langle \omega, D_p \rangle$		
$n_w :=$ HKDF$(256, \omega,$ "n"$)$		
	$\xrightarrow{n_w, D_p}$	
		verify if D_p is valid
		$r_0 :=$ HKDF$(256, n_w \| \| d_s,$ "wr0"$)$
		$r_1 :=$ HKDF$(256, n_w \| \| d_s,$ "wr1"$)$
		$R_0 := r_0 G$
		$R_1 := r_1 G$
	$\xleftarrow{R_0, R_1}$	
derive coin key pair :		
$c_s :=$ HKDF$(256, \omega \| \| R_0 \| \| R_1,$ "cs"$)$		
$C_p :=$ Ed25519.GetPub(c_s)		
blind:		
$b_s :=$ HKDF$(256, \omega \| \| R_0 \| \| R_1,$ "b-seed"$)$		
$\alpha_0 :=$ HKDF$(256, b_s,$ "a0"$)$		
$\alpha_1 :=$ HKDF$(256, b_s,$ "a1"$)$		
$\beta_0 :=$ HKDF$(256, b_s,$ "b0"$)$		
$\beta_1 :=$ HKDF$(256, b_s,$ "b1"$)$		
$R'_0 := R_0 + \alpha_0 G + \beta_0 D_p$		
$R'_1 := R_1 + \alpha_1 G + \beta_1 D_p$		
$c'_0 := H(R'_0, C_p)$		
$c'_1 := H(R'_1, C_p)$		
$c_0 := c'_0 + \beta_0 \mod p$		
$c_1 := c'_1 + \beta_1 \mod p$		

Continued in figure 3.2

Figure 3.1.: Withdrawal process using Clause Blind Schnorr Signatures part 1

Customer
knows:
reserve keys w_s, W_p
denomination public key D_p

Exchange
knows:
reserve public key W_p
denomination keys d_s, D_p

Continuation of figure 3.1

sign with reserve private key:
$\rho_W := \langle n_w, D_p, c_0, c_1 \rangle$
$\sigma_W :=$ Ed25519.Sign(w_s, ρ_W)

$$\xrightarrow{W_p, \sigma_W, \rho_W}$$

$\langle n_w, D_p, c_0, c_1 \rangle := \rho_W$
verify if D_p is valid
check Ed25519.Verify(W_p, ρ_W, σ_W)
$b :=$ HKDF$(1, n_w || d_s,$ "b"$)$
$s \leftarrow$ GetWithdraw(n_w, D_p)
if $s = \perp$
check !NonceReuse(n_w, D_p, ρ_W)
$r_b :=$ HKDF$(256, n_w || d_s,$ "rb"$)$
$s := r_b + c_b d_s \mod p$
decrease balance if sufficient and
persist NonceUse $\langle n_w, D_p, \rho_W \rangle$
persist $\langle D_p, s \rangle$
endif

$$\xleftarrow{b, s}$$

verify signature:
check if $sG = R_b + c_b D_p$
unblind:
$s' := s + \alpha_b \mod p$
verify signature:
check if $s'G = R'_b + c'_b D_p$
$\sigma_C := \langle R'_b, s' \rangle$
resulting coin: c_s, C_p, σ_C, D_p

Figure 3.2.: Withdrawal process using Clause Blind Schnorr Signatures part 2

RefreshDerive Schnorr

The RefreshDerive protocol is described in figure 3.3. For this protocol, the main change is that more values need to be derived somehow. These blinding factors are also derived from x. Then the challenges $\overline{c_0}$ and $\overline{c_1}$ are generated as in the Clause Blind Schnorr Signature Scheme.

```
RefreshDerive$(t, D_{p(t)}, C_p, R_0, R_1)$
$T :=$ Curve25519.GetPub$(t)$
$x :=$ ECDH-EC$(t, C_p)$
$c'_s :=$ HKDF$(256, x, "c")$
$C'_p :=$ Ed25519.GetPub$(c'_s)$
$b_s :=$ HKDF$(256, x||R_0||R_1, "b-seed")$
$\alpha_0 :=$ HKDF$(256, b_s, "a0")$
$\alpha_1 :=$ HKDF$(256, b_s, "a1")$
$\beta_0 :=$ HKDF$(256, b_s, "b0")$
$\beta_1 :=$ HKDF$(256, b_s, "b1")$
$R'_0 = R_0 + \alpha_0 G + \beta_0 D_p$
$R'_1 = R_1 + \alpha_1 G + \beta_1 D_p$
$c'_0 = H(R'_0, C'_p)$
$c'_1 = H(R'_1, C'_p)$
$\overline{c_0} = c'_0 + \beta_0 \mod p$
$\overline{c_1} = c'_1 + \beta_1 \mod p$
**return** $\langle T, c'_s, C'_p, \overline{c_0}, \overline{c_1} \rangle$
```

Figure 3.3.: The RefreshDerive replaced with Schnorr blind signature details. As before the uses the seed s on the dirty coin for generating the new coin. The new coin needs to be signed later on with the denomination key.

Refresh Protocol

In the commit phase (see figure 3.4) there needs to be requested an R_0 and R_1 before deriving the new coins. There now needs to be calculated two different commit hashes, one for $\overline{c_0}$ and one for $\overline{c_1}$. The exchange needs to additionally generate a random $b \leftarrow \{0, 1\}$ to choose a $\overline{c_b}$. The reveal phase (see figure 3.5) now is continued only with the chosen $\overline{c_b}$. In the reveal phase, the RSA signing and unblinding is exchanged with Schnorr's blind signature counterparts.

Customer
knows:
denomination public key D_p
coin$_0 = \langle D_{p(0)}, c_s^{(0)}, C_p^{(0)}, \sigma_c^{(0)} \rangle$

Exchange
knows:
old denomination keys $d_{s(0)} D_{p(0)}$
new denomination keys d_s, D_P

$n_r := randombytes(32)$
persist $\langle n_r, D_p \rangle$

$$\xrightarrow{n_r, D_p}$$

verify if D_p is valid
$r_0 :=$ HKDF$(256, n_r || d_s,$ ”mr0”$)$
$r_1 :=$ HKDF$(256, n_r || d_s,$ ”mr1”$)$
$R_0 := r_0 G$
$R_1 := r_1 G$

$$\xleftarrow{R_0, R_1}$$

```
for $i = 1, \ldots, \kappa$ :
$t_i :=$ HKDF$(256, c_s^{(0)}, n_r || R_0 || R_1,$ ”t$i$”$)$
$X_i :=$ RefreshDerive$(t_i, D_p, C_p^{(0)}, R_0, R_1)$
$(T_i, c_s^{(i)}, C_p^{(i)}, \overline{c_0}, \overline{c_1}) := X_i$
endfor
$h_T := H(T_1, \ldots, T_k)$
$h_{\overline{c_0}} := H(\overline{c_0}_1, \ldots, \overline{c}_{0_k})$
$h_{\overline{c_1}} := H(\overline{c_1}_1, \ldots, \overline{c}_{1_k})$
$h_{\overline{c}} := H(h_{\overline{c_0}}, h_{\overline{c_1}}, n_r)$
$h_C := H(h_T, h_{\overline{c}})$
$\rho_{RC} := \langle h_C, D_p, D_{p(0)}, C_p^{(0)}, \sigma_C^{(0)} \rangle$
$\sigma_{RC} :=$ Ed25519.Sign$(c_s^{(0)}, \rho_{RC})$
Persist refresh-request
$\langle n_r, R_0, R_1, \rho_{RC}, \sigma_{RC} \rangle$
```

Continued in figure 3.5

Figure 3.4.: Refresh protocol (commit phase part 1) using Clause Blind Schnorr Signatures

Customer | Exchange

Continuation of figure 3.4

$\xrightarrow{\rho_{RC}, \sigma_{RC}, n_r}$

Exchange:

$\langle h_C, D_p, D_{p(0)}, C_p^{(0)}, \sigma_C^{(0)} \rangle := \rho_{RC}$
check Ed25519.Verify$(C_p^{(0)}, \sigma_{RC}, \rho_{RC})$

$\gamma \leftarrow$ GetOldRefresh(ρ_{RC})
Comment: GetOldRefresh$(\rho_{RC} \mapsto \{\perp, \gamma\})$
if $\gamma = \perp$
$v :=$ Denomination(D_p)
check IsOverspending$(C_p^{(0)}, D_{p(0)}, v)$
verify if D_p is valid
check !NonceReuse(n_r, D_p, ρ_{RC})
check Schnorr.Verify$(D_{p(0)}, C_p^{(0)}, \sigma_C^{(0)})$
MarkFractionalSpend$(C_p^{(0)}, v)$
$\gamma \leftarrow \{1, \dots, \kappa\}$
persist NonceUse $\langle n_r, D_p, \rho_{RC} \rangle$
persist refresh-record $\langle \rho_{RC}, \gamma \rangle$

$\xleftarrow{\gamma}$

Customer:

check IsConsistentChallenge(ρ_{RC}, γ)
Comment: IsConsistentChallenge $(\rho_{RC}, \gamma) \mapsto \{\perp, \top\}$

Persist refresh-challenge$\langle \rho_{RC}, \gamma \rangle$
$S := \langle t_1, \dots, t_{\gamma-1}, t_{\gamma+1}, \dots, t_\kappa \rangle$
$\rho_L := \langle C_p^{(0)}, D_p, T_\gamma, \overline{c_0}_\gamma, \overline{c_1}_\gamma \rangle$
$\rho_{RR} := \langle \rho_L, S \rangle$
$\sigma_L :=$ Ed25519.Sign$(c_s^{(0)}, \rho_L)$

$\xrightarrow{\rho_{RR}, \rho_L, \sigma_L}$

Continued in figure 3.6

Figure 3.5.: Refresh protocol (commit phase part 2) using Clause Blind Schnorr Signatures

Customer | Exchange

Continuation of figure 3.5

$\xrightarrow{\rho_{RR}, \rho_L, \sigma_L}$

$\langle C_p^{(0)}, D_p, T_\gamma, \overline{c_0}_\gamma, \overline{c_1}_\gamma \rangle := \rho_L$
$\langle T'_\gamma, \overline{c_0}_\gamma, \overline{c_1}_\gamma, S \rangle := \rho_{RR}$
$\langle t_1, \dots, t_{\gamma-1}, t_{\gamma+1}, \dots, t_\kappa \rangle := S$
check Ed25519.Verify$(C_p^{(0)}, \sigma_L, \rho_L)$
$b :=$ HKDF$(1, n_r || d_{s(i)},$ "b"$)$
for $i = 1, \dots, \gamma - 1, \gamma + 1, \dots, \kappa$
$X_i :=$ RefreshDerive$(t_i, D_p, C_p^{(0)}$
$, R_0, R_1)$
$\langle T_i, c_s^{(i)}, C_p^{(i)}, \overline{c_1}_i, \overline{c_2}_i \rangle := X_i$
endfor
$h'_T = H(T_1, \dots, T_{\gamma-1}, T'_\gamma, T_{\gamma+1}, \dots, T_\kappa)$
$h'_{\overline{c0}} := H(\overline{c0_1}, \dots, \overline{c}0_k)$
$h'_{\overline{c1}} := H(\overline{c1_1}, \dots, \overline{c}1_k)$
$h'_{\overline{c}} := H(h_{\overline{c0}}, h_{\overline{c1}}, n_r)$
$h'_C = H(h'_T, h'_{\overline{c}})$
check $h_C = h'_C$
$r_b :=$ HKDF$(256, n_r || d_s,$ "mr$b$"$)$
$\overline{s}_{C_p}^{(\gamma)} = r_b + \overline{c_{b_\gamma}} d_s \mod p$
persist $\langle \rho_L, \sigma_L, S \rangle$

$\xleftarrow{b, \overline{s}_C^{(\gamma)}}$

unblind:
$s_C'^{(\gamma)} := \overline{s}_C^{(\gamma)} + \alpha_b \mod p$
verify signature:
check if $\overline{s'}_C^{(\gamma)} G \equiv R'_b + \overline{c'}_{0_\gamma} D_p$
$\sigma_C^{(\gamma)} := \langle s_C'^{(\gamma)}, R'_b \rangle$
Persist coin$\langle D_p, c_s^{(\gamma)}, C_p^{(\gamma)}, \sigma_C^{(\gamma)} \rangle$

Figure 3.6.: Refresh protocol (reveal phase) using Clause Blind Schnorr Signatures

Linking Protocol

The beginning of the linking protocol (see figure 3.7) is the same as in the current protocol. After the customer received the answer L the only difference is in obtaining the coin. To re-obtain the derived coin, the same calculations as in 3.3 are made.

Customer
knows:
coin$_0 = \langle D_{p(0)}, c_s^{(0)}, C_p^{(0)}, \sigma_C^{(0)} \rangle$

Exchange
knows:

$$\xrightarrow{C_{p(0)}}$$

$L :=$ LookupLink$(C_{p(0)})$
Comment: LookupLink$(C_p^{(0)}) \mapsto \{\langle \rho_L^{(i)}, \sigma_L^{(i)}, \overline{\sigma}_C^{(i)}, b \rangle\}$

$$\xleftarrow{L}$$

for $\langle \rho_L^{(i)}, \overline{\sigma}_L^{(i)}, \overline{\sigma}_C^{(i)}, b \rangle \in L$
$\langle \hat{C}_p^{(i)}, D_p^{(i)}, T_\gamma^{(i)}, \overline{c_0}_\gamma^{(i)}, \overline{c_1}_\gamma^{(i)}, n_r \rangle := \rho_L^{(i)}$
$\langle \overline{s}_C^{(i)}, R_b^{(i)} \rangle := \overline{\sigma}_C^{(i)}$
check $\hat{C}_p^{(i)} \equiv C_p^{(0)}$
check Ed25519.Verify$(C_p^{(0)}, \rho_L^{(i)}, \sigma_L^{(i)})$
$\langle \overline{s}_C^{(i)}, R_b^{(i)} \rangle := \sigma_C^{(i)}$
$x_i :=$ ECDH$(c_s^{(0)}, T_\gamma^{(i)})$
$c_s^{(i)} :=$ HKDF$(256, x, \text{"c"})$
$C_p^{(i)} :=$ Ed25519.GetPub$(c_s^{(i)})$
$b_s^{(i)} :=$ HKDF$(256, x_i || R_0^{(i)} || R_1^{(i)}, \text{"b-seed"})$
$\alpha_b :=$ HKDF$(256, b_s^{(i)}, \text{"a}b\text{"})$
$\beta_b :=$ HKDF$(256, b_s^{(i)}, \text{"b}b\text{"})$
$R'^{(i)}_b = R_b^{(i)} + \alpha_b G + \beta_b D_p^{(i)}$
$c'_b = H(R'_b, C_p^{(i)})$
$c_b = c'_b + \beta_b \mod p$
$s'^{(i)}_C := \overline{s}_C^{(i)} + \alpha_b \mod p$
$\sigma_C^{(i)} := \langle s'^{(i)}_C, R'_b \rangle$
check $s'^{(i)}_C G \equiv R'^{(i)}_b + c'_b D_p^{(i)}$
(Re-)obtain coin$\langle D_p^{(i)}, c_s^{(i)}, C_p^{(i)}, \sigma_C^{(i)} \rangle$

Figure 3.7.: Linking protocol using Clause Blind Schnorr Signatures

3.2.4. Tipping

Tipping remains unchanged, except for the content of the message $\rho_W = D_p, c_0, c_1$ signed by the merchant using its reserve private key.

3.2.5. Recoup Protocol

The recoup protocol distinguishes three different cases, which all depend on the state of a coin whose denomination key has been revoked. The following listing outlines the necessary changes on the protocol, please refer to Dold's documentation section 2.2.1 [Dol19] for details regarding the different cases.

- **The revoked coin has never been seen by the exchange**:
  The withdraw transcript (and verification) must be adjusted in order for the exchange to be able to retrace the blinding.
- **The coin has been partially spent**:
  In this case the refresh protocol will be invoked on the coin. The necessary changes are outlined in 3.2.3.
- **The revoked coin has never been seen by the exchange and resulted from a refresh operation**:
  The refresh protocol transcript and its blinding factors must be adjusted to consider the changes in the blind signature scheme.

4. Protocol Specification

The proposed Taler protocols using the Clause Blind Schnorr Signature Scheme will be implemented as an additional option besides the existing RSA Blind Signatures variant of the protocol as suggested by Christian Grothoff. A Taler Exchange operator should be able to configure whether he wants to use RSA Blind Signatures or Clause Blind Schnorr Signatures.
This variant allows to choose the signature scheme globally or per denomination. Furthermore, it allows a change of signature scheme in a non-breaking way by revoking (or letting expire) a denomination and offering new denominations with the other scheme.
The following key points are specified in this chapter:

- Architecture of the different components
- Explain and specify needed changes
- Data strucutures
- Public APIs[1]
- Persistence
- Used libraries

4.1. Architecture

Before specifying the implementation of the different protocols, a deeper understanding of the technical architecture of Talers components is needed. this section introduces the architecture of the exchange and wallet components and explains where the needed changes need to be implemented on a high-level.

4.1.1. Exchange

An introduction to the exchange can be found in section 2.1.1. An exchange operator needs to run and maintain some additional services besides Taler's exchange. Although this is not directly relevant for the implementation, it helps to better understand the environment in which the exchange runs. The perspective of an exchange operator can be seen in figure 4.1.

[1]Application Programming Interfaces

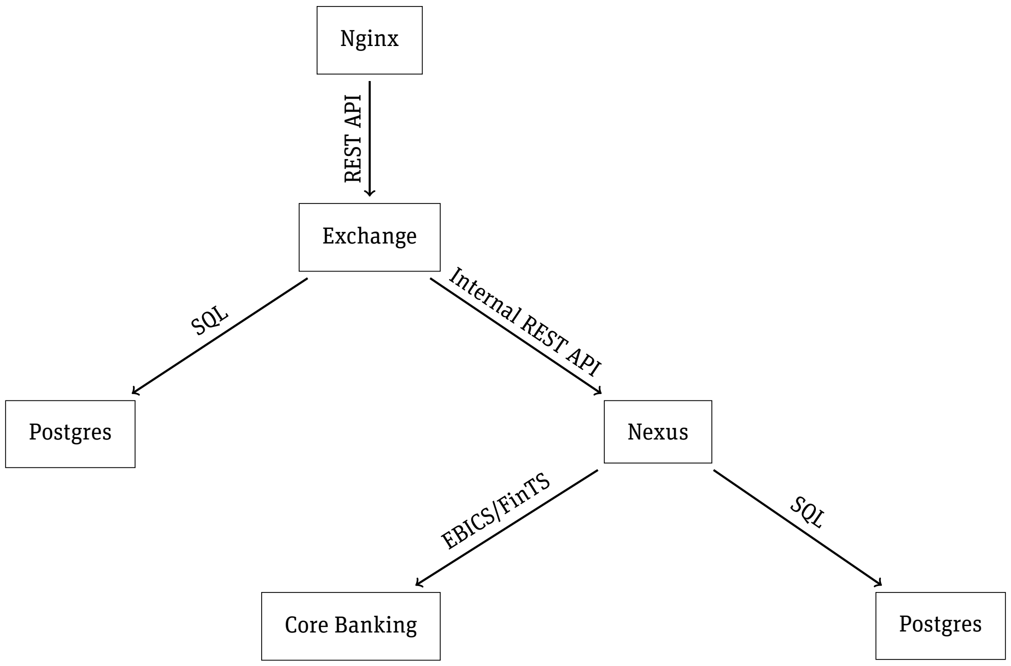

Figure 4.1.: Taler exchange operator architecture (source: [Tal21a])

The software architecture of the exchange can be seen in figure 4.2. The API runs under the httpd service, where the API endpoints need to be adjusted/added to incorporate the changes of this thesis. The httpd server has no access to the private keys of the denomination and online signing keys. Only the corresponding security module can perform operations requiring the private key. Further the keys are also managed by these security modules. To support Clause Blind Schnorr Signatures a new security module, which performs signature operations, is added. To persist the new data structures, the postgres helpers need to be adjusted to serialize/deserialize the new Clause Blind Schnorr Signatures data structures. More details on what changes are needed in these places is discussed in the following sections.

4.1.2. Wallet

The architecture of the wallet implementation (as seen in figure 4.3) is quite straightforward. To add support for Clause Blind Schnorr Signatures in the wallet, the cryptographic routines need to be reimplemented in Typescript. Taler uses tweetnacl [Dan14] which provides functionality for the group operations. There are existing HKDF and FDH implementations, that can be reused.
Furthermore, the Taler protocols need to be adjusted to support Clause Blind Schnorr Signatures in the wallet-core.

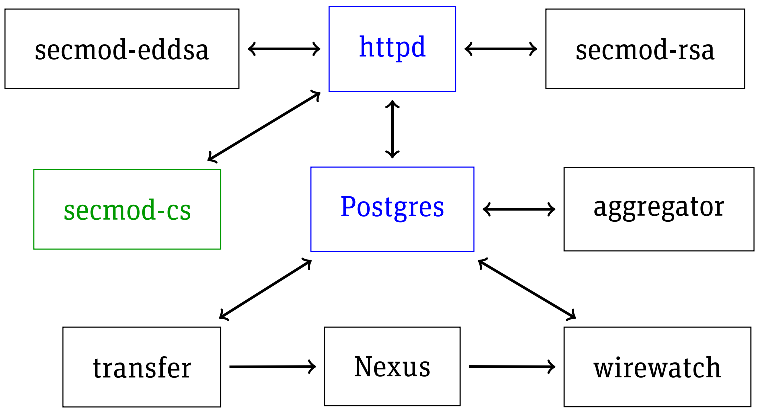

Figure 4.2.: Taler exchange architecture (source: [Tal21a])

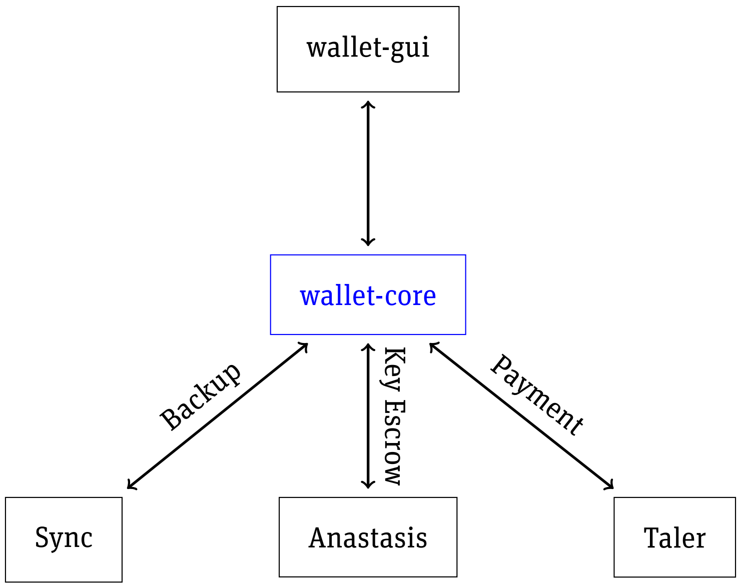

Figure 4.3.: Taler wallet architecture (source: [Tal21a])

4.2. Persistence

The Clause Blind Schnorr Signature scheme is quite different to RSA Blind Signatures. Despite the differences, the database model does not need to be changed. The only change needed an additional type field, specifying whether RSA or CS is used as signature algorithm. To persist the new structs introduced with the support for Clause Blind Schnorr Signatures, only the postgres helpers need to support serialization and deserialization of the new structs.

4.3. Testing

We will partially use test-driven development, meaning that we will write tests (at least for the known good case) before implementing functions, and extend them during and after development. This allows us to check the functionality (code section, function(s)) during development, while being able to extend testing whenever we identify new test cases during development.

Test cases can be used to verify different aspects of a functionality. These are the ones we will focus on.

- **Known good**: Known good cases test whether a functionality works as expected. They are the most useful during development, because they indicate whether the code is working as expected.
- **Known Bad**: Known bad cases test whether functionality that is known not to work behaves as expected.
- **Determinism**: This case type checks whether the same input leads to the same output. It is important for code that must work deterministic (same output), non-deterministic (e.g. random output) or based on a state that impacts the functionality.
- **Performance testing**: Performance testing is used to gather timing information that can be used to identify functionality with long duration, or to compare performance between different implementations or major changes. We will restrict performance testing to the comparison of the Blind RSA Signature Scheme and the Clause Blind Schnorr Signature Scheme.

4.4. Signature Scheme Operations in GNUnet

The signature scheme operations implemented are needed in all other parts of the implementation. Taler's cryptographic primitives (e.g. RSA Blind Signatures, HKDF, hash functions) are mostly implemented in GNUnet utils, therefore the Clause Blind Schnorr Signature routines will be implemented in GNUnet too. It is important to provide a clear API for the cryptographic routines and to test them thoroughly. Libsodium will be used for finite field arithmetic ([doc]) and for other functionality when available (e.g. for key generation). Thus, a widely used and well tested cryptographic library is used for group operations.

For Full-Domain Hash and HKDF existing implementations provided by GNUnet are used. The HKDF is used with SHA-512 for the extraction phase and SHA-256 for the expansion phase.

4.4.1. Data Structures

Libsodium represents Ed25519 points and scalars as 32-byte char arrays. To provide a more user-friendly API, structs were created to represent each type. For example `struct`

`GNUNET_CRYPTO_CsPrivateKey` or `struct GNUNET_CRYPTO_RSecret` The main reason is to increase readability and to prevent misusage of the API. Unlike RSA, our Clause Blind Schnorr Signatures on Ed25519 data structures have a fixed sizes. The different data structures can be found in table 4.1.

Values	Data Structure	Data Type
Curve25519 Scalar	GNUNET_CRYPTO_Cs25519Scalar	32 byte char array
Curve25519 Point	GNUNET_CRYPTO_Cs25519Point	32 byte char array
Private Key	GNUNET_CRYPTO_CsPrivateKey	GNUNET_CRYPTO_Cs25519Scalar
Public Key	GNUNET_CRYPTO_CsPublicKey	GNUNET_CRYPTO_Cs25519Point
α, β	GNUNET_CRYPTO_CsBlindingSecret	2x GNUNET_CRYPTO_Cs25519Scalar
r	GNUNET_CRYPTO_CsRSecret	GNUNET_CRYPTO_Cs25519Scalar
R	GNUNET_CRYPTO_CsRPublic	GNUNET_CRYPTO_Cs25519Point
c	GNUNET_CRYPTO_CsC	GNUNET_CRYPTO_Cs25519Scalar
s	GNUNET_CRYPTO_CsBlindS	GNUNET_CRYPTO_Cs25519Scalar
s'	GNUNET_CRYPTO_CsS	GNUNET_CRYPTO_Cs25519Scalar
$\sigma := \langle s', R' \rangle$	GNUNET_CRYPTO_CsSignature	GNUNET_CRYPTO_Cs25519Scalar
		GNUNET_CRYPTO_Cs25519Point
Nonce	GNUNET_CRYPTO_CsNonce	32 byte char array

Table 4.1.: Data structures for cryptographic routines

4.4.2. Library API

The public API and related data structures are specified in the C header file `src/include/gnunet_crypto_lib.h` in the GNUnet repository [Repl]. It was developed in multiple iterations based on feedback from Christian Grothoff. The complete C header API can be found in the repository. This section provides an overview of the implemented crypto API.

Some design decisions need to be explained further:

- In order to prevent misusage of our implementation and increase readability, the functions that represent different stages in the signature scheme takes different data types as in- and output. Internally most variables are either scalars or curve points (except for nonces, secrets and messages).
- Operations that are performed twice in the Clause Blind Schnorr Signature Scheme (e.g. derivation of r) do not have to be called twice. Instead, the API returns an array of two instead of a single value.
  For these functions, we also optimized the HKDF (as proposed by Christian Grothoff).

Instead of calling HKDF twice (with different salts, e.g. ”r0” and ”r1”), we call it one time (e.g. with salt ”r”) and double the output length.

- The cryptographic hash function used to derive c' (hash of R' and message) must map the results into the main subgroup for scalars, meaning that it has to be a FDH (see 2.2.3).

The following API examples should provide an overview on how the API works and how to use it.

First of all the API must provide functionality to create a Curve25519 keypair as in listing 4.1

```
/**
 * Create a new random private key.
 *
 * @param[out] priv where to write the fresh private key
 */
void
GNUNET_CRYPTO_cs_private_key_generate (
    struct GNUNET_CRYPTO_CsPrivateKey *priv);


/**
 * Extract the public key of the given private key.
 *
 * @param priv the private key
 * @param[out] pub where to write the public key
 */
void
GNUNET_CRYPTO_cs_private_key_get_public (
    const struct GNUNET_CRYPTO_CsPrivateKey *priv,
    struct GNUNET_CRYPTO_CsPublicKey *pub);
```

Listing 4.1: GNUnet create keypair API

The signer needs an API to generate his secret r and calculate his public point R. As specified in the redesign of the protocols, the r must not be chosen randomly because we need to provide *abort-idempotency*. However, the secret r still needs to be *unpredictable* and look random to the client. The r_derive API derives such a secret r from a nonce and a long-term secret with HKDF. Further, the API ensures that a caller must generate two secret r as in the Clause Blind Schnorr Signature scheme. This should discourage people from using the unsecure Blind Schnorr Signature scheme. See 4.2.

```
/**
 * Derive a new secret r pair r0 and r1.
 * In original papers r is generated randomly
 * To provide abort-idempotency, r needs to be derived but still needs to be UNPREDICTABLE
 * To ensure unpredictability a new nonce should be used when a new r needs to be derived.
 * Uses HKDF internally.
 * Comment: Can be done in one HKDF shot and split output.
 *
 * @param nonce is a random nonce
 * @param lts is a long-term-secret in form of a private key
 * @param[out] r array containing derived secrets r0 and r1
 */
void
GNUNET_CRYPTO_cs_r_derive (const struct GNUNET_CRYPTO_CsNonce *nonce,
                           const struct GNUNET_CRYPTO_CsPrivateKey *lts,
                               struct GNUNET_CRYPTO_CsRSecret r[2]);


/**
 * Extract the public R of the given secret r.
 *
 * @param r_priv the private key
 * @param[out] r_pub where to write the public key
 */
void
GNUNET_CRYPTO_cs_r_get_public (const struct GNUNET_CRYPTO_CsRSecret *r_priv,
                               struct GNUNET_CRYPTO_CsRPublic *r_pub);
```

Listing 4.2: GNUnet r derive API

Same as the r_derive, the blinding secrets are also derived and not generated randomly. The blinding secrets are generated by a client who provides a secret as seed to derive the secrets from as in listing 4.3.

```
/**
 * Derives new random blinding factors.
 * In original papers blinding factors are generated randomly
 * To provide abort-idempotency, blinding factors need to be derived but still need to be UNPREDICTABLE
 * To ensure unpredictability a new nonce has to be used.
 * Uses HKDF internally
 *
 * @param secret is secret to derive blinding factors
 * @param secret_len secret length
 * @param[out] bs array containing the two derivedGNUNET_CRYPTO_CsBlindingSecret
 */
void
GNUNET_CRYPTO_cs_blinding_secrets_derive (
    const struct GNUNET_CRYPTO_CsNonce *blind_seed,
    struct GNUNET_CRYPTO_CsBlindingSecret bs[2]);
```

Listing 4.3: GNUnet blinding secrets derive API

Further the Clause Blind Schnorr API provides an API to calculate the two blinded c of the message with the two public R, the blinding factors and the public key as in listing 4.4.

```
/**
 * Calculate two blinded c's
 * Comment: One would be insecure due to Wagner's algorithm solving ROS
 *
 * @param bs array of the two blinding factor structs each containing alpha and beta
 * @param r_pub array of the two signer's nonce R
 * @param pub the public key of the signer
 * @param msg the message to blind in preparation for signing
 * @param msg_len length of message msg
 * @param[out] blinded_c array of the two blinded c's
 */
void
GNUNET_CRYPTO_cs_calc_blinded_c (
    const struct GNUNET_CRYPTO_CsBlindingSecret bs[2],
    const struct GNUNET_CRYPTO_CsRPublic r_pub[2],
    const struct GNUNET_CRYPTO_CsPublicKey *pub,
    const void *msg,
    size_t msg_len,
    struct GNUNET_CRYPTO_CsC blinded_c[2]);
```

Listing 4.4: GNUnet calculate blinded c API

The sign function in our API is called sign_derive, since we derive $b \in \{0,1\}$ from the long-term secret and then calculate the signature scalar of c_b. See listing 4.5.

```c
/**
 * Sign a blinded c
 * This function derives b from a nonce and a longterm secret
 * In original papers b is generated randomly
 * To provide abort-idempotency, b needs to be derived but still need to be UNPREDICTABLE.
 * To ensure unpredictability a new nonce has to be used for every signature
 * HKDF is used internally for derivation
 * r0 and r1 can be derived prior by using GNUNET_CRYPTO_cs_r_derive
 *
 * @param priv private key to use for the signing and as LTS in HKDF
 * @param r array of the two secret nonce from the signer
 * @param c array of the two blinded c to sign c_b
 * @param nonce is a random nonce
 * @param[out] blinded_signature_scalar where to write the signature
 * @return 0 or 1 for b (see Clause Blind Signature Scheme)
 */
int
GNUNET_CRYPTO_cs_sign_derive(
    const struct GNUNET_CRYPTO_CsPrivateKey *priv,
    const struct GNUNET_CRYPTO_CsRSecret r[2],
    const struct GNUNET_CRYPTO_CsC c[2],
    const struct GNUNET_CRYPTO_CsNonce *nonce,
    struct GNUNET_CRYPTO_CsBlindS *blinded_signature_scalar);
```

Listing 4.5: GNUnet sign API

The API for the unblind operation can be called with the blinding secrets and the signature scalar received from the signer as in listing 4.6.

```c
 /**
 * Unblind a blind-signed signature using a c that was blinded
 *
 * @param blinded_signature_scalar the signature made on the blinded c
 * @param bs the blinding factors used in the blinding
 * @param[out] signature_scalar where to write the unblinded signature
 */
void
GNUNET_CRYPTO_cs_unblind (
    const struct GNUNET_CRYPTO_CsBlindS *blinded_signature_scalar,
    const struct GNUNET_CRYPTO_CsBlindingSecret *bs,
    struct GNUNET_CRYPTO_CsS *signature_scalar);
```

Listing 4.6: GNUnet unblind API

The verify API takes the message and its signature with the public key and returns GNUNET_OK for a valid signature and GNUNET_SYSERR otherwhise. See listing 4.7.

```c
  /**
  * Verify whether the given message corresponds to the given signature and the
  * signature is valid with respect to the given public key.
  *
  * @param sig signature that is being validated
  * @param pub public key of the signer
  * @param msg is the message that should be signed by @a sig  (message is used to calculate c)
  * @param msg_len is the message length
  * @returns #GNUNET_YES on success, #GNUNET_SYSERR if signature invalid
  */
 enum GNUNET_GenericReturnValue
 GNUNET_CRYPTO_cs_verify (const struct GNUNET_CRYPTO_CsSignature *sig,
                              const struct GNUNET_CRYPTO_CsPublicKey *pub,
                              const void *msg,
                              size_t msg_len);
```

Listing 4.7: GNUnet verify API

4.4.3. Testing

For digital signature schemes, the most important test case is the *known good* case where a signature is created and successfully validated. This test case already tests very much in a digital signature scheme. When the signature creation or verification has a bug, a test will not succeed, because the mathematic operations need to be correct to be validated correctly.

The cryptographic operations are further tested for deterministicy (where it applies), meaning that multiple function calls with the same input must lead to the same output.

Since libsodium is used for the finite field arithmetic operations and is a well tested library, many cryptographic tests are already done in libsodium.

The performance is measured in a benchmark to see how performant Clause Blind Schnorr Signatures are in comparison to the RSA Blind Signature Scheme.

4.5. Taler Cryptographic Utilities

Taler provides utility functions to support cryptographic operations.
This chapter provides an overview of these utility functions and about the functionality they provide.

4.5.1. Planchet Creation

In crypto.c many utility functions are provided to create planchets (for planchet details see 2.8), blinding secrets and much more. One difference between RSA Blind Signatures and Clause Blind Schnorr Signatures is, that the coin private key and RSA blinding secret can be created at the same point in time, since the RSA blinding secret is created randomly. However, for Clause Blind Schnorr secrets an additional step is needed, the public R_0 and R_1 are required to calculate the blinding seed to derive the secrets.

A planchet in the Clause Blind Schnorr Signature Scheme can be created as followed (implementation details ommited).

1. Create planchet with new EdDSA private key
2. Derive withdraw nonce
3. Request public R_0, R_1 from signer
4. Derive blinding seed

5. Prepare (blind) the planchet

After the planchet is created, it is sent to the exchange to be signed.

4.5.2. Taler CS Security Module

The exchange segregates access to the private keys with separate security module processes. The security module has sole access to the private keys of the online signing keys and thus, only a security module can create signatures. The different *taler-exchange-secmod* processes (separated by signature scheme) are managing the exchanges online signing keys. The RSA denomination keys for example are managed with *taler-exchange-secmod-rsa*.

Now a new *taler-exchange-secmod-cs* needs to be created for managing the Clause Blind Schnorr Signatures denomination keys. These security modules run on the same machine as the httpd process and they use UNIX Domain Sockets as method for Inter Process Communication. A short introduction about UNIX Domain Sockets can be found in the blog post [Lim22]. Furthermore, the security modules are used to protect the online signing keys by performing the actual signing operations in the dedicated taler-secmod-cs process. This abstraction makes it harder for an attacker who has already compromised the http daemon to gain access to the private keys. However, such an attacker would still be able to sign arbitrary messages (see [SAd]). A crypto helper exists for each security module, these functions can be called inside the exchange for operations requiring the private online signing keys. The new Clause Schnorr security module and corresponding crypto helper provides the following functionality:

- Private Key Management and creation
- Request public R_0, R_1
- Request a signature of a c_0, c_1 pair
- Revoke an online signing key

4.5.3. Testing

All of the operations have tests and are included in unit tests. As a template for testing, the existing RSA tests were used and adjusted for Clause Blind Schnorr Signatures.

4.6. Denomination Key Management

Since we introduce a type of denomination keys, related operations like connection to the Clause Blind Schnorr Signatures security module, making the denominations available for customers, persisting them in the database and offline signing using the exchange's offline signature key have to be extended to incorporate the Clause Blind Schnorr Signature Scheme.

The exchange offline signer requests the future, not yet signed keys by calling GET `/management/keys` as described in table 4.2.

`GET /management/keys`

Field	**Value**
future_denoms	Information about denomination keys
future_signkeys	Information about exchange online signing keys
master_pub	Exchange's master public key
denom_secmod_public_key	RSA security module public key
denom_secmod_cs_public_key	Clause Blind Schnorr Signatures security module public key
signkey_secmod_public_key	Online singing security module public key

Table 4.2.: GET `/management/keys` response data

It then signs the keys and returns them using POST on the same URL[2] with the data described in table 4.3.

`POST /management/keys`

Field	**Value**
denom_sigs	Denomination key signatures
signkey_sigs	Online signing key signatures

Table 4.3.: POST `/management/keys` response data

Wallets can then call GET `/keys` to obtain the current denominations and other information, the response is described in table 4.4.

`GET /keys`

4.7. New Endpoint for R

The withdraw and refresh protocols using the Claude Blind Schnorr Signature Scheme introduce an additional round trip. In this round trip, the customer requests two R from the exchange. The exchange uses a secret r to calculate $R := rG$.
In contrast to the plain Clause Blind Schnorr Signature Scheme (see 2.2.4), r isn't generated randomly but instead derived using a HKDF with a nonce from the customer and a denomination private key (secret only known by the exchange). This still ensures that the private r can't be anticipated, but has multiple advantages regarding abort-idempotency.

[2]uniform resource locator

Field	Value
version	Exchange's protocol version
currency	Currency
master_public_key	Exchange's master public key
reserve_closing_delay	Delay before reserves are closed
signkeys	Exchange's online signing public keys
recoup	Revoked keys
denoms	List of denominations
auditors	Auditors for this exchange
list_issue_date	Timestamp
eddsa_pub	Exchange's online signing public key
eddsa_sig	Signature (use "eddsa_pub" for verification)

Table 4.4.: GET `/keys` response data

Abort-idempotency means that a withdraw or refresh operation can be aborted in any step and later tried again (using the same values) without yielding a different result. The challenge for r, R regarding abort-idempotency is to ensure that the same r is used during the complete signature creation process.

The only drawback of this process is that we have to ensure that the same nonce and secret aren't used for different withdraw- or refresh-operations. This is done during signature creation and will be described in the withdraw protocol section 4.8.

4.7.1. Public APIs and Data Structures

This is a new functionality, meaning a new endpoint accessible to customers has to be introduced. It will be made available in the exchange HTTP server under `POST /csr` and will take the input parameters described in table 4.5 (as JSON).

Field	Type	Value
nonce	String	32 Bytes encoded in Crockford base32 Hex
denom_pub_hash	String	Denomination Public Key encoded in Crockford base32 Hex

Table 4.5.: POST `/csr` request data

The exchange will then check the denomination and return one of these HTTP status codes:

- **200 (HTTP_OK)**: Request Successful

- **400 (BAD_REQUEST)**: Invalid input parameters
- **404 (NOT_FOUND)**: Denomination unknown or not Clause Schnorr
- **410 (GONE)**: Denomination revoked/expired
- **412 (PRECONDITION_FAILED)**: Denomination not yet valid

When the request was successful, the exchange returns the data described in table 4.6 (as JSON).

Field	Type	Value
r_pub_0	String	32 Bytes encoded in Crockford base32 Hex
r_pub_1	String	32 Bytes encoded in Crockford base32 Hex

Table 4.6.: POST `/csr` response data

4.7.2. Persistence

This API does not persist anything. This is because the resulting R_0, R_1 are derived and can be derived in a later step.

4.8. Withdraw Protocol

The withdraw protocol has been introduced in section 2.3.1. For the Clause Blind Schnorr Signature Scheme necessary adjustments are described in section 3.2.1.

4.8.1. Public APIs and Data Structures

The existing endpoint is available under `POST /reserves/[reserve]/withdraw` where ”reserve” is the reserve public key encoded as Crockford base32. It takes the following input parameters described in table 4.7 as JSON.

```
POST /reserves/[reserve]/withdraw
```

Field	Value
denom_pub_hash	Denomination Public Key
coin_ev	RSA blinded coin public key
reserve_sig	Signature over the request using the reserve’s private key

Table 4.7.: Withdraw request data

In order to facilitate parsing, Christian Grothoff suggested to include the cipher type in the ”coin_ev” field, thus creating a nested JSON (as described in table 4.8).

Field	Type	Value
cipher	Integer	Denomination cipher: 1 stands for RSA
rsa_blinded_planchet	String	RSA blinded coin public key

Table 4.8.: Withdraw ”coin_ev” field (RSA)

For the Clause Schnorr implementation, the field ”rsa_blinded_planchet” will be replaced with the necessary values as described in table 4.9.

Field	Type	Value
cipher	Integer	Denomination cipher: 2 stands for Clause Blind Schnorr Signatures
cs_nonce	String	32 Bytes encoded in Crockford base32 Hex
cs_blinded_c0	String	32 Bytes encoded in Crockford base32 Hex
cs_blinded_c1	String	32 Bytes encoded in Crockford base32 Hex

Table 4.9.: Withdraw ”coin_ev” field (Clause Blind Schnorr Signatures)

The exchange will then process the withdraw request and return one of these HTTP status codes:

- **200 (HTTP_OK)**: Request Successful
- **400 (BAD_REQUEST)**: Invalid input parameters (can also happen if denomination cipher doesn’t match with cipher in ”coin_ev”)
- **403 (FORBIDDEN)**: Signature contained in ”reserve_sig” invalid
- **404 (NOT_FOUND)**: Denomination unknown
- **410 (GONE)**: Denomination revoked/expired
- **412 (PRECONDITION_FAILED)**: Denomination not yet valid

When the request was successful, the exchange returns the RSA signature as JSON (described in table 4.10).

Field	Type	Value
cipher	Integer	Denomination cipher: 1 stands for RSA
blinded_rsa_signature	String	RSA signature

Table 4.10.: Withdraw response (RSA)

Table 4.11 describes the response for Clause Blind Schnorr Signatures.

Field	Type	Value
cipher	Integer	Denomination cipher: 2 stands for Clause Blind Schnorr Signatures
b	Integer	Clause Blind Schnorr Signatures signature session identifier (either 0 or 1)
s	String	signature scalar (32 Bytes encoded in Crockford base32 Hex)

Table 4.11.: Withdraw response (Clause Blind Schnorr Signatures)

4.8.2. Persistence

Persistence for withdrawing is implemented in the function `postgres_do_withdraw` in `src/exchangedb/plugin_exchangedb_postgres.c` For Clause Blind Schnorr Signatures, persisting the blinded signature must be implemented.

4.9. Deposit Protocol

For the deposit protocol (described in section 2.3.2) only the handling and verification of Clause Blind Schnorr Signatures signatures has to be added.

4.9.1. Public APIs and Data Structures

Deposit is an existing endpoint available under `POST /coins/[coin public key]/deposit` where "coin public key" is encoded as Crockford base32. Additional parameters are passed as JSON (as described in table 4.12).

```
POST /coins/[coin public key]/deposit
```

Relevant field for the Clause Blind Schnorr Signatures implementation is the field "ub_sig" containing the unblinded denomination signature of the coin. For RSA, the (nested) JSON is described in table 4.13.

Table 4.14 describes the values in "ub_sig" required for Clause Blind Schnorr Signatures.

4.9.2. Persistence

Persistence is handled in the functions `postgres_insert_deposit` and `postgres_have_deposit` located in `src/exchangedb/plugin_exchangedb_postgres.c`. However, these functions are not containing Clause Blind Schnorr Signatures-specific persistence.

What needs to be adjusted however, is the function `postgres_ensure_coin_known` called by the function `TEH_make_coin_known` (located in `src/exchange/taler-exchange-httpd_db.c`).

Field	Value
merchant_payto_uri	Account that is credited
wire_salt	Salt used by the merchant
contribution	Amount to use for payment (for one specific coin)
denom_pub_hash	Denomination public key hash
ub_sig	(unblinded) denomination signature of coin
merchant_pub	Merchant public key
h_contract_terms	Contract terms hash
coin_sig	Deposit permission signature
timestamp	Timestamp of generation
refund_deadline (optional)	Refund deadline
wire_transfer_deadline (optional)	Wire transfer deadline

Table 4.12.: Spend request

Field	Type	Value
cipher	Integer	Denomination cipher: 1 stands for RSA
rsa_signature	String	Unblinded RSA signature

Table 4.13.: ub_sig (RSA)

Field	Type	Value
cipher	Integer	Denomination cipher: 2 stands for Clause Blind Schnorr Signatures
cs_signature_r	String	Curve point R' (32 Bytes encoded in Crockford base32 Hex)
cs_signature_s	String	Signature scalar (32 Bytes encoded in Crockford base32 Hex)

Table 4.14.: ub_sig (Clause Blind Schnorr Signatures)

5. Implementation

This chapter gives an overview on the implementation challenges and discusses special parts in the implementation.

5.1. Signature Scheme Operations

The signature scheme operations are implemented in the GNUnet core repository [Repl] (and have been merged into the master branch). This would allow other GNUnet projects to use our implementation of the Clause Blind Schnorr Signature Scheme.

The implementation is done in multiple locations:

- `src/include/gnunet_crypto_lib.h`: This header file is included when using GNUnet's cryptography implementation.
- `src/util/crypto_cs.c`: The functions specified in `gnunet_crypto_lib.h` will be implemented here.
- `src/util/test_crypto_cs.c`: The test cases for the signature scheme will be implemented here.
- `src/util/perf_crypto_cs.c`: This file houses the implementation of a small program that will be used to compare the performance against the blind RSA Signature Scheme.

The specification explaining the API can be found in section 4.4. There are two internal functions that have to be explained further in this section.

The `map_to_scalar_subgroup` function clamps scalars, which is necessary for values that are derived using a HKDF. It sets the three least significant bits to zero (making the scalar a multiple of 8), sets the most significant bit to zero and the second-most significant bit to one. This process is further described in [RFC7748] and [Mad20].

```
static void
map_to_scalar_subgroup (struct GNUNET_CRYPTO_Cs25519Scalar *scalar)
{
    scalar->d[0] &= 248;
    scalar->d[31] &= 127;
    scalar->d[31] |= 64;
}
```

Listing 5.1: Function map_to_scalar_subgroup - Crypto API

Another important function is the FDH (see 2.2.3) used to map the message to a scalar. GNUnet provides a FDH function, which expects libgcrypt's multi precision format. A conversion function is provided by GNUnet, which requires the data to be in big endian format. Since libsodium uses a little endian representation, the conversion process must include endianness conversion. The complete FDH including the required conversions is implemented in the function described in listing 5.2.

```
static void
cs_full_domain_hash (const struct GNUNET_CRYPTO_CsRPublic *r_dash,
                     const void *msg,
                     size_t msg_len,
                     const struct GNUNET_CRYPTO_CsPublicKey *pub,
                     struct GNUNET_CRYPTO_CsC *c)
{
   ...
```

Listing 5.2: Function cs_full_domain_hash - Crypto API

Last but not least, the implementation has one notable performance improvement not mentioned in the redesigned protocols. In various steps HKDF is used multiple times in a row. For example to derive the four blinding secrets $\alpha_0, \alpha_1, \beta_0, \beta_1$. The derivation can be done in one HKDF call with bigger output size, 128 bit in this case. The output then can be split in four parts and then mapped to the ed25519 subgroup. This can be done secure, because as explained in subsection 2.2.2 a HKDF output is truly random.

5.2. Taler Cryptographic Utilities

Implementation is done in Taler's exchange. From here on the implementation can be found in the exchange git repository [Repd].

The cryptographic utilities of Taler can be found in `src/util`.
The implementation is done in various locations:

- `src/include/taler_crypto_lib.h`: This header file is included when using Taler's cryptography implementation. The different data structures and functionality are defined here.
- `src/util/denom.c`: Implement denomination utility functions for Clause Blind Schnorr Signatures cases
- `src/util/crypto.c`: Adjust all utility functions to support Clause Blind Schnorr Signatures. crypto.c contains many cryptographic utility functions, for example to create planchets or blinding factors.
- `src/util/test_crypto.c`: Functionality tests for crypto.c and denom.c
- `src/include/taler_signatures.h`: In this header file message formats and signature constants are defined (not modified)

- `src/util/secmod_signatures.c`: Utility functions for Taler security module signatures

The security module `taler-secmod-cs` is implemented here:

- `src/util/taler-exchange-secmod-cs.c`: Standalone process to perform private key Clause Blind Schnorr signature operations.
- `src/util/taler-exchange-secmod-cs.h`: Specification of IPC[1] messages for the CS secmod process
- `src/util/taler-exchange-secmod-cs.conf`: Configuration file for the secmod process
- `src/util/secmod_common.c` and `src/util/secmod_common.h`: Common functions for the exchange's security modules (not modified)

The corresponding crypto helper, that talks with the security module, and its tests & benchmarks are implemented here:

- `src/util/crypto_helper_cs.c`: Utility functions to communicate with the security module
- `src/util/crypto_helper_common.c`: and `crypto_helper_common.h`: Common functions for the exchange security modules (not modified)
- `src/util/test_helper_cs.c`: Tests and benchmarks for the Clause Blind Schnorr Signatures crypto helper

5.3. Denomination Key Management

For the implementation, the Clause Blind Schnorr Signatures security module had to be connected to the key handling and the Clause Blind Schnorr Signatures denominations had to be integrated:

- `src/exchange/taler-exchange-httpd_keys.h` and `src/exchange/taler-exchange-httpd_keys.c`: Integrate Clause Blind Schnorr Signatures secmod and denomination key management
- `src/exchange-tools/taler-exchange-offline.c`: Implement Clause Blind Schnorr Signatures case for offline signing of denomination keys
- `src/include/taler_exchange_service.h`:
  Add Clause Blind Schnorr Signatures secmod public key to struct
  TALER_EXCHANGE_FutureKeys

[1]Inter Process Communication

- `src/json/json_helper.c`: Implement CS case in function parse_denom_pub (used in taler-exchange-offline.c)
- `src/json/json_pack.c`: Implement Clause Blind Schnorr Signatures case in function TALER_JSON_pack_denom_pub (used in taler-exchange-httpd_keys.c)
- `src/pq/pq_query_helper.c`: Implement Clause Blind Schnorr Signatures case in function qconv_denom_pub
- `src/pq/pq_result_helper.c`: Implement Clause Blind Schnorr Signatures case in function extract_denom_pub

In order for the tests to pass, the following changes had to be implemented:

- `src/lib/exchange_api_management_get_keys.c`: Add denom_secmod_cs_public_key JSON parsing, implement Clause Blind Schnorr Signatures case in function TALER_EXCHANGE_ManagementGetKeysHandle
- `src/testing/.gitignore`: Add paths where Clause Blind Schnorr Signatures keys are stored (secmod-helper)
- `src/testing/test_auditor_api.conf`: Add section taler-exchange-secmod-cs
- `src/testing/test_exchange_api_keys_cherry_picking.conf`: Add section taler-exchange-secmod-cs
- `src/testing/testing_api_helpers_exchange.c`: Add Clause Blind Schnorr Signatures secmod start and stop logic

5.4. New Endpoint for R

The new endpoint is available in the exchange's HTTP server under `/csr`. It parses and checks the input, passes the request for derivation of the two R's down to the Clause Blind Schnorr Signatures security module and returns them to the requestor. The implementation can be found in:

- `src/exchange/taler-exchange-httpd.c`: Definition for the new endpoint, calls the function that handles `/csr` requests
- `src/exchange/taler-exchange-httpd_responses.h` and `src/exchange/taler-exchange-httpd_responses.c`:
  Added function TEH_RESPONSE_reply_invalid_denom_cipher_for_operation that indicates a failure when the endpoint is called for a non-Clause Blind Schnorr Signatures denomination
- `src/exchange/taler-exchange-httpd_csr.h` and `src/exchange/taler-exchange-httpd_csr.c`:
  Implementation of the request handler for the new endpoint

- `src/exchange/taler-exchange-httpd_keys.h` and `src/exchange/taler-exchange-httpd_keys.c`:
  Additional function TEH_keys_denomination_cs_r_pub that passes down the request to derive the R to the taler-exchange-secmod-cs helper

The tests that check the functionality of the procotols are defined in `src/testing/` and use code that calls the API (located in `src/lib/`). Since the new endpoint is used during withdrawing coins, testing for the `/csr` endpoint is integrated in these protocol tests. Therefore, a call to the endpoint was implemented and later integrated into the calls to the withdraw-API. Code for calling the endpoint is located in these files:

- `src/include/taler_exchange_service.h`:
  Header describing functions and data structures used in withdraw and refresh testing:
  - struct TALER_EXCHANGE_CsRHandle: Handle containing request information
  - struct TALER_EXCHANGE_CsRResponse: Response details
  - function TALER_EXCHANGE_CsRCallback: Callback function to deliver the results (used in withdraw and refresh)
  - function TALER_EXCHANGE_csr: Used to call endpoint
  - function TALER_EXCHANGE_csr_cancel: Used to free dynamically allocated resources
- `src/lib/exchange_api_csr.c`: Implementation of `/csr` request

5.5. Withdraw Protocol

Since this is an existing endpoint, it was adjusted to support Clause Blind Schnorr Signatures. Mainly, the in- and output-handling had to be adjusted as described in section 4.8.1, additional cipher checks for the denomination were added and the Clause Blind Schnorr Signatures for persisting the request in the database was implemented.

An interesting part of the implementation is the check whether a nonce was already used for this denomination or not (step: $s \leftarrow \text{GetWithdraw}(n_w, D_p)$). This step ensures that the same signature will always be returned for a certain nonce. Using the same nonce for the same denomination twice without this check would lead to the same random value r. This is due to derivation of $r := \text{HKDF}(256, n_w || d_s, \text{"r"})$. An attacker could then immediately recover the secret key by the following equation: $(h' - h) * x \mod q = s - s' \mod q$ [Tib17]. There are popular examples of this vulnerability in Sony Playstation 3's or Bitcoins ECDSA implementation [OBE18] [Wan+19]. More details on how such a vulnerability can be exploited can be found in one of the author's blog posts [Dem21].
The designed Taler protocols using Clause Blind Schnorr Signatures are preventing this attack by checking the nonce and return the previously generated signature. Additionally the denomination's public key is included in this check to prevent another issue explained

in section 5.7.
The check is implemented by persisting a hash value over n_w and D_p. On every withdrawal `check_request_idempotent()` is called, which checks whether the persisted hash matches with the current n_w, D_p pair.

- `src/exchange/taler-exchange-httpd_withdraw.c`: Implementation of Clause Blind Schnorr Signatures case for withdraw endpoint
- `src/exchange/taler-exchange-httpd_keys.c`: Implement Clause Blind Schnorr Signatures case in function
  TEH_keys_denomination_sign (passes the signature creation down to the crypto helpers)
- `src/include/taler_json_lib.h` and `src/json/json_helper.c`:
  Add function TALER_JSON_spec_blinded_planchet
- `src/json/json_pack.c`:
  Implement Clause Blind Schnorr Signatures case in function
  TALER_JSON_pack_blinded_denom_sig
- `src/pq/pq_query_helper.c`: implement Clause Blind Schnorr Signatures case in functions qconv_denom_sig and qconv_blinded_denom_sig
- `src/pq/pq_result_helper.c`: Implement Clause Blind Schnorr Signatures case in function extract_blinded_denom_sig

For testing, the Clause Blind Schnorr Signatures-related data structures and procedures as well as the request to the additional endpoint `/csr` (before performing the actual withdrawal) were integrated:

- `src/testing/test_exchange_api.c`: Add additional tests for Clause Blind Schnorr Signatures withdraw
- `src/include/taler_testing_lib.h`: Specification for functions
  TALER_TESTING_cmd_withdraw_cs_amount and
  TALER_TESTING_cmd_withdraw_cs_amount_reuse_key, add denomination cipher parameter to function TALER_TESTING_find_pk
- `src/testing/testing_api_cmd_withdraw.c`: add functions
  TALER_TESTING_cmd_withdraw_cs_amount and
  TALER_TESTING_cmd_withdraw_cs_amount_reuse_key, implement Clause Blind Schnorr Signatures-specific logic for withdraw
- `src/testing/testing_api_helpers_exchange.c`: add cipher parameter to function TALER_TESTING_find_pk
- `src/lib/exchange_api_withdraw.c`: Implement Clause Blind Schnorr Signatures-specific withdraw logic, integrate `/csr` request

- `src/lib/exchange_api_withdraw2.c`: implement Clause Blind Schnorr Signatures case
- `src/include/taler_json_lib.h` and `src/json/json_pack.c`:
  Add function TALER_JSON_pack_blinded_planchet
- `src/json/json_helper.c` implement Clause Blind Schnorr Signatures case in function parse_blinded_denom_sig

5.6. Deposit Protocol

For deposit, only few changes were necessary because some of the required functionality has already been added for the previously implemented protocols, and only the coin signature verification is Clause Blind Schnorr Signatures-specific in this protocol.

- `/src/exchange/taler-exchange-httpd_deposit.c`: Add check whether denomination cipher and denomination signature cipher are equal
- `/src/json/json_helper.c`: Implement Clause Blind Schnorr Signatures case in function parse_denom_sig
- `/src/pq/pq_result_helper.c`: Implement Clause Blind Schnorr Signatures case in function extract_denom_sig

Tests for deposit are implemented here:

- `/src/testing/test_exchange_api.c`: Add tests (see ”struct TALER_TESTING_Command spend_cs[]”) that spend Clause Blind Schnorr Signatures coins withdrawn in tests added for withdrawal
- `/src/json/json_pack.c`: Implement Clause Blind Schnorr Signatures case in function TALER_JSON_pack_denom_sig

5.7. Fixing a Minor Security Issue in Taler’s RSA Blind Signature Protocols

While implementing the nonce check in the Clause Blind Schnorr Signatures protocol (see section 5.5), a minor security issue in Taler’s current RSA Blind Signature implementation was detected and fixed. The issue was only in the implementation of the current RSA Blind Signature protocols, the fix for this scenario was already implemented in Clause Blind Schnorr Signatures since the beginning.

5.7.1. Security Issue

The redesigned Clause Blind Schnorr Signatures protocols already include the denomination key in the nonce check, which fixes this issue (see 3.2.1). In the case of RSA Blind Signatures, the current protocol includes an idempotence check by persisting the hash value of the blinded coin m'. On a withdrawal/refresh the idempotence check compares if the hash value of m' was seen in the past and returns the 'old' signature on a match. This could lead to the following scenario:

1. A broken wallet withdraws a coin with denomination $D_{p_{(1)}}$.
2. The wallet sends a request to withdraw the same coin for denomination $D_{p_{(2)}}$.
3. The exchange returns the signature for the denomination $D_{p_{(1)}}$ due to the idempotence check.
4. Since the exchange returned an invalid signature, the customer can file a complaint at the auditor.
5. The auditor then has to investigate why the exchange returned invalid signatures.
6. The auditor can disprove the complaint by querying the persisted hash used for the idempotence check. With the associated denomination public key that is also persisted, the auditor can successfully verify the signature and thus prove that the exchange operated honestly.

Including the denomination public key into the persisted hash for the idempotence check solves this issue. If a broken wallet now sends the same coin for more than one denomination, the exchange returns valid signatures in both cases.
While this is still an issue, this case is already handled nicely in Taler since this situation could also occur if a broken value tries to withdraw the same coin with two different blinding factors.

5.7.2. Impact

The impact of this security vulnerability is considered as very low. An auditor investigating such an issue can simply retrace what happened by checking the persisted hash and associated denomination. The impact of the issue is, that an auditor needs to investigate an issue, which can be prevented inside the protocol.
In the previous section the client was considered a broken wallet. While this could be done on purpose by malicious a customer, there is no real motivation for abusing this issue due the easy detection of an auditor.

5.7.3. Fix

Listing 5.3 shows the code of calculating the hash for the idempotency check in the RSA case before it was fixed. By trying to implement the Clause Blind Schnorr Signatures case, the question came up why the RSA case has not included the denomination key into the check. After discussing this issue with Christian Grothoff, the conclusion was to include the denomination public key to prevent the discussed issue.

```
enum GNUNET_GenericReturnValue
TALER_coin_ev_hash (const struct TALER_BlindedPlanchet *blinded_planchet,
                    struct TALER_BlindedCoinHash *bch)
{
  switch (blinded_planchet->cipher)
  {
  case TALER_DENOMINATION_RSA:
    GNUNET_CRYPTO_hash (
      blinded_planchet->details.rsa_blinded_planchet.blinded_msg,
      blinded_planchet->details.rsa_blinded_planchet.blinded_msg_size,
      &bch->hash);
    return GNUNET_OK;
  case TALER_DENOMINATION_CS:
  ...
```

Listing 5.3: Idempotency check on RSA

The issue is fixed by adding a hash of the current denomination key into the calculation of the hash used in the idempotence check. The applied fix can be seen in listing 5.4.

```
enum GNUNET_GenericReturnValue
TALER_coin_ev_hash (const struct TALER_BlindedPlanchet *blinded_planchet,
                    const struct TALER_DenominationHash *denom_hash,
                    struct TALER_BlindedCoinHash *bch)
{
  switch (blinded_planchet->cipher)
  {
  case TALER_DENOMINATION_RSA:
    {
      struct GNUNET_HashContext *hash_context;
      hash_context = GNUNET_CRYPTO_hash_context_start ();

      GNUNET_CRYPTO_hash_context_read (hash_context,
                                       &denom_hash->hash,
                                       sizeof(denom_hash->hash));
      GNUNET_CRYPTO_hash_context_read (hash_context,
                                       blinded_planchet->details.
                                       rsa_blinded_planchet.blinded_msg,
                                       blinded_planchet->details.
                                       rsa_blinded_planchet.blinded_msg_size);
      GNUNET_CRYPTO_hash_context_finish (hash_context,
                                         &bch->hash);
      return GNUNET_OK;
    }
  case TALER_DENOMINATION_CS:
    {
      struct GNUNET_HashContext *hash_context;
      hash_context = GNUNET_CRYPTO_hash_context_start ();

      GNUNET_CRYPTO_hash_context_read (hash_context,
                                       &denom_hash->hash,
                                       sizeof(denom_hash->hash));
      GNUNET_CRYPTO_hash_context_read (hash_context,
                                       &blinded_planchet->details.
                                       cs_blinded_planchet.nonce,
                                       sizeof (blinded_planchet->details.
                                               cs_blinded_planchet.nonce));
      GNUNET_CRYPTO_hash_context_finish (hash_context,
                                         &bch->hash);
      return GNUNET_OK;
    }
  default:
    GNUNET_break (0);
    return GNUNET_SYSERR;
  }
}
```

Listing 5.4: Fixed idempotency check

6. Discussion

This chapter analyses the Clause Blind Schnorr Signature Scheme implementation and compares it to the existing implementation with RSA Blind Signatures. The comparison will include the schemes itself, performance comparisons and a discussion on the security assumptions. For the performance comparison CPU usage, latency, bandwidth and storage space are compared.

6.1. Cipher Agility

One of the benefits of having another blind signature scheme in Taler is *cipher agility*. Cipher agility means that one scheme can substitute another, for example if one scheme gets compromised in the future.

Cipher agility is considered harmful in certain situations. TLS 1.2 [RD08] and IPSEC/IKEv2 [FK11] are good examples on how dangerous cipher agility inside protocols can be. There are many ways these protocols can be set up insecure.

Taler's protocols are built around blind signature schemes. Therefore it is crucial to have an additional secure blind signature scheme that works with all Taler protocols. As described in section 2.2.4, blind signature schemes can vary and may be complex to substitute. The Clause Blind Schnorr Signatures implementation provides such an alternative and thus *cipher agility*.

6.2. Scheme Comparison

Both schemes are explained in the preliminaries chapter (RSA Blind Signatures in section 2.5 and Clause Blind Schnorr Signatures in 2.7).
There are multiple differences worth mentioning. The first difference is that Schnorr signatures are inherently randomized. This is also where the additional step in Schnorr signatures comes from. A random number is chosen by the signer for every signature.
In Clause Blind Schnorr Signatures two blinding secrets are used instead of one in RSA Blind Signatures. On top of that, Clause Blind Schnorr Signatures needs to do most computations for signature creation twice, due to the ROS problem (see 2.2.4).

Abort-idempotency is a very important property for Taler. Ensuring abort-idempotency with the Clause Blind Schnorr Signatures scheme is harder than it was with RSA, due to the many random elements in the scheme ($r_0, r_1, \alpha_0, \alpha_1, \beta_0, \beta_1, b$). The reason that these

values are chosen randomly is the need for *unpredictability*.
In the protocols (see chapter 3) HKDF is extensively used to derive these values instead of randomly generating them. That way, the values are still *unpredictable* (due to HKDF properties), but now the protocols also ensure *abort-idempotency*. In comparison to the RSA Blind Signature scheme, this is a clever and elegant solution, but the protocol complexity is increased.

One could now think that RSA would be much simpler to implement, since the scheme looks easier and more accessible for many. This can go horribly wrong and many developers still underestimate implementing RSA. There are a lot of attacks on RSA, some examples are listed on the famous tool RsaCtfTool [Gan22]. Ben Perez made a popular talk and blog post, about why one should stop using RSA and should preferably use libsodium and ECC[1] [Per22]. Using RSA Blind Signatures in Taler is still a reasonable and fine choice. Taler uses libgcrypt, a well-known and tested library.
To conclude, the Clause Blind Schnorr Signatures protocols might be more complex to understand than the RSA Blind Signature protocols. One has to keep in mind that implementing RSA correctly is hard.
Another difference worth mentioning is, that the Clause Blind Schnorr Signatures scheme does not need scheme specific configurations, whereas RSA needs a key size specified. This is because the implemented Clause Blind Schnorr Signatures version only supports Curve25519.

Furthermore, both schemes provide *perfect blindness*, see paragraph 2.2.4 for RSA and paragraph 2.2.4 for Clause Blind Schnorr Signatures.

6.3. Performance Comparison

This section compares how the two schemes perform regarding CPU usage, latency, bandwidth and space. Clause Schnorr has fixed key sizes with 256 bits (32 bytes), which we compare against different RSA key sizes (1024, 2048, 3072 and 4096 bits). In terms of security, Clause Blind Schnorr Signatures 256 bit keys could be compared to 3072 bit RSA keys (see `https://www.keylength.com/` for more information).

6.3.1. CPU Usage

Various benchmarks were made on different CPU architectures. This section discusses the main results, detailed information about the performance comparison can be found in appendix B. We thank the Taler team for providing measurements from additional systems and architectures.

Table 6.1 shows how Clause Blind Schnorr Signatures compares to RSA 3072. RSA 3072 was chosen for comparison, since they both provide a comparable level of security. Both provide about 128 bits of security, which means that roughly 2^{128} attempts in average are

[1]Elliptic Curve Cryptography

needed for a successful brute-force attack.
The table shows that Clause Blind Schnorr Signatures has better performance compared to RSA 3072 in all operations. The biggest difference can be seen in the key generation. In RSA, two random primes are needed, whereas DLP algorithms like Clause Blind Schnorr Signatures only need to generate a random value. Since key generation is done rarely compared to the other operations, the time needed for key generation does not matter that much.
Furthermore, the blinding in Clause Blind Schnorr Signatures is still faster than blinding in RSA, although in the Clause Blind Schnorr Signatures case the calculation is done twice. Also the derivation of r_0, r_1, the generation of R_0, R_1 and the derivation of $\alpha_0, \beta_0, \alpha_1, \beta_1$ is included in the measurement for the blinding operation of Clause Blind Schnorr Signatures. Signing and blinding operations are much faster in Clause Blind Schnorr Signatures, also Clause Blind Schnorr Signatures signature verification is faster than RSA 3072.

Setup

CPU: 8-core AMD Ryzen 7 PRO 5850U
OS: Ubuntu 21.10 Linux 5.13.0-25-generic #26-Ubuntu SMP Fri Jan 7 15:48:31 UTC 2022 x86_64 x86_64 x86_64 GNU/Linux
libsodium version: 1.0.18-1build1
libgcrypt version: 1.8.7-5ubuntu2

Benchmarks with other hardware setups can be found in appendix B.

Signature Scheme	Operation	Speed
CS	10x key generation	0.204 ms
RSA 3072 bit	10x key generation	2684 ms
CS	10x derive R_0, R_1 & blinding	3.870 ms
RSA 3072 bit	10x blinding	5 ms
CS	10x signing	0.077 ms
RSA 3072 bit	10x signing	86 ms
CS	10x unblinding	0.001 ms
RSA 3072 bit	10x unblinding	24 ms
CS	10x verifying	1.358 ms
RSA 3072 bit	10x verifying	3.075 ms

Table 6.1.: Comparison on CS vs. RSA 3072

Table 6.2 shows a comparison between Clause Blind Schnorr Signatures and RSA 1024

bit. RSA 1024 is in some situations faster than the Clause Blind Schnorr Signatures implementation. Note that 1024 bit keys are not recommended for many use cases, but the highest currently known RSA factorization done is 829 bits [Wik21d]. The following section 6.5 explains the risk running RSA 1024 or Clause Blind Schnorr Signatures denominations further.
The blind and unblind operations are running in a wallet implementation, therefore the comparison with RSA 1024 is very interesting for devices with less CPU power. Comparison of such hardware can be found in appendix B, these comparison results come to the same conlcusion.
Although RSA 1024 bit is much faster in the blinding operation, Clause Blind Schnorr Signatures still perform better when calculating the blinding and unblinding operations together. Clause Blind Schnorr Signatures unblinding computes only an addition of two scalars $s + \alpha \mod p$, while RSA computes $s * r^{-1}$. To conclude, Clause Blind Schnorr Signatures are faster than RSA 1024 bit and provide a better level of security. This can be especially useful for wallets running on devices with less CPU power. The verification on RSA 1024 is faster than Clause Blind Schnorr Signatures. Therefore, it has to be further investigated which algorithm would overall perform better for the exchange or merchants. While RSA 1024 bit can compete in certain operations, Clause Blind Schnorr Signatures provide a better level of security and are still faster in most operations.

Signature Scheme	Operation	Speed
CS	10x key generation	0.204 ms
RSA 1024 bit	10x key generation	126 ms
CS	10x derive R_0, R_1 & blinding	3.870 ms
RSA 1024 bit	10x blinding	1.282 ms
CS	10x signing	0.077 ms
RSA 1024 bit	10x signing	7 ms
CS	10x unblinding	0.001 ms
RSA 1024 bit	10x unblinding	2.991 ms
CS	10x verifying	1.358 ms
RSA 1024 bit	10x verifying	0.876 ms

Table 6.2.: Comparison on CS vs RSA 1024

6.3.2. Disk Space

⚠ These are theoretical calculations, implementations may choose to persist additional values.

Clause Blind Schnorr Signatures save disk space due to the much smaller key sizes. Even more disk space is saved by deriving values with the HKDF, these values do not have to be stored.
Table 6.3 shows the disk space comparison of signatures, the private keys alone need even less space with 256 bits per key.
The wallet saves a lot of disk space by deriving most of the values. In the Clause Blind Schnorr Signatures case a wallet must at least persist the private key c_s, R_0, R_1, s', D_p, each being 256 bits (32 bytes). A wallet needs to persist 150 bytes per coin in total. In the RSA Blind Signature case the wallet persists c_s, b, σ_c, D_p.
Note: for refreshed coins an additional 32 byte value is persisted as seed.
c_s is still a 32 byte value in the RSA case, the other values depend on the RSA key size. (32 byte + 3 * *rsa_keysize*). The disk space comparison for a wallet can be found in 6.4.

Signature Scheme	Disk Space	Factor	Disk Space 1M signatures
CS	512 bits	1x	64 MB
RSA 1024 bit	1024 bits	2x	128 MB
RSA 2048 bit	2048 bits	4x	256 MB
RSA 3072 bit	3072 bits	6x	384 MB
RSA 4096 bit	4096 bits	8x	512 MB

Table 6.3.: Comparison disk space signatures

Signature Scheme	Disk Space	Factor	Disk Space 1M coins
CS 256 bits	150 bytes	1x	150 MB
RSA 1024 bit	416 bytes	2.7x	416 MB
RSA 2048 bit	800 bytes	5.3x	800 MB
RSA 3072 bit	1184 bytes	7.9x	1184 MB
RSA 4096 bit	1568 bytes	10.4x	1568 MB

Table 6.4.: Comparison disk space wallet

6.3.3. Bandwidth

These are theoretical calculations, implementations may choose to persist additional values.

The reasons that Clause Blind Schnorr Signatures use less bandwidth is mostly because the signature/key sizes are much smaller. The bandwith improvements for the `/keys` API is the same as specified in the table with disk space comparison 6.3. For Clause Blind Schnorr Signatures many calculations are performed twice, therefore also two values are submitted. Table 6.5 compares the bandwidth used in a withdrawal. The 32 byte values $2 * n_w, 2 * D_p, R_0, R_1, s, W_p, c_0, c_1, \sigma_W$ as well as an integer b are transmitted for Clause Blind Schnorr Signatures.
For RSA, the values D_p, m', σ'_c have the same size as the key size. Additionally, the 32 byte values W_p, σ_W are transmitted.

In the refresh protocol the only difference is an additional hash (h_{C_0}, h_{C_1} instead of only h_C) sent in the commit phase. Depending on the hash size another 32 byte (or 64 byte) value is transmitted.

Signature Scheme	Bandwith used	Factor	1M coins
CS 256 bits	356 bytes	1x	324 MB
RSA 1024 bit	448 bytes	1.3x	448 MB
RSA 2048 bit	832 bytes	2.5x	832 MB
RSA 3072 bit	1216 bytes	3.75x	1216 MB
RSA 4096 bit	1600 bytes	4.9x	1600 MB

Table 6.5.: Bandwith comparison withdrawal

6.3.4. Latency

This section the notion of Round-Trip Time (see [pre22]) is used. There are many factors that influence the measurement of a Round-Trip Time. Following factors can bring huge changes in the value of RTT[2]s.

- Distance
- Transmission medium
- Network hops
- Traffic levels

[2]Round-Trip Time

- Server response time

All of these factors will vary in reality and are independent of the scheme.
The important comparison here is the number of RT[3]s as in table 6.6.

Signature Scheme	Number of RTs
RSA Blind Signatures	1
Clause Blind Schnorr Signatures	2

Table 6.6.: Comparison of Round-Trips

While creating RSA Blind Signatures have one RT, Clause Blind Schnorr Signatures need an additional RT for requesting the public R_0, R_1. This means that the time spend for withdrawing is almost **doubled** (the R request doesn't have any persistence and therefore requires less time) in comparison to RSA.

A coin should not be spent immediately after withdrawal or refresh. Otherwise, an adversary could deanonymize a customer by correlating the timestamps. The additional RT is a drawback of Clause Blind Schnorr Signatures compared to RSA, but negligible due to the fact that a coin must not be spent immediately.

6.4. Security Assumptions

This section discusses the differences regarding the security assumptions of the schemes. This section should not explain nor analyze the security assumptions, instead the section focuses on explaining what these assumptions mean and what should be kept in mind about them. Read section 2.2.3 and it's references for more information on the assumptions.

RSA's security assumptions are well known since quite a long time and a lot of research is done. Despite being a lot of attacks [Gan22] [Per22], RSA is still considered a secure scheme after decades.
For Schnorr Signatures the Discrete Logarithm Problem (see subsubsection 2.2.5) needs to be hard. Also the DLP is well-known and is being researched since decades.
However, with Blind Schorr Signatures an additional assumption needs to hold; the ROS problem. Compared to the other assumptions, ROS is relatively new and still a recent research topic. A recent paper from 2020 on the (in)security of ROS [Ben+20] broke many schemes relying on ROS being hard, including Schnorr Blind signatures. The paper on which we rely on (updated in 2021) with the Clause Blind Schnorr Signature Scheme [FPS19] is considered secure at the time of writing.

[3]Round-Trip

6.5. Risk

As introduced in section 6.4, Clause Blind Schnorr Signatures rely on an additional assumption currently being researched. Compared to other schemes, the chosen Clause Blind Schnorr Signatures are very new (published in 2019, updated in 2021). While every scheme could potentially be broken, older ones already went through a lot of research and their assumptions are well-known. Therefore, the risk that a vulnerability in Clause Blind Schnorr Signatures will be discovered is probably higher than a newly discovered vulnerability breaking RSA.

Unpredictability of r is a key aspect of the signature creation process of Clause Blind Schnorr Signatures. The redesigned Taler protocols solve this by persisting the nonce and denomination key (described in section 5.5) and checking for reuse of this combination before signature creation. If this process is malfunctioning (broken implementation, faulty database) or can be circumvented in any way, recovery of a denomination private key is possible.

An exchange operator can still consider using Clause Blind Schnorr Signatures as denomination scheme, as there are multiple benefits (see section 6.3). The financial loss in the worst case can be calculated and capped by the validity of a denomination key. If a vulnerability in the Clause Blind Schnorr Signatures would be detected, an exchange operator could revoke the corresponding denomination keys and change the scheme to RSA Blind Signatures. The wallets can then follow the refund protocol to get the money back.

6.6. Comparison Conclusion

A detailed comparison of the two blind signature schemes was made. This last section interprets the results and concludes the comparison.

Clause Blind Schnorr Signatures on Curve25519 provide the same security level as RSA Blind Signatures with 3072 bit key sizes. The implementation of Clause Blind Schnorr Signatures is the clear winner in all performance comparisons with RSA 3072 bits.

1024 bit RSA is faster than the Clause Blind Schnorr Signatures implementation in certain operations. The Clause Blind Schnorr Signatures implementation still offers better performance for wallets with less CPU power and provides a much higher level of security (comparable to RSA 3072). As further comparisons show, RSA scales very bad the larger the keys get and Clause Blind Schnorr Signatures performs much better overall.

As discussed in the risk section 6.5, Clause Blind Schnorr Signatures have an additional security assumption, which is still a recent research topic. Clause Blind Schnorr Signatures provide various benefits and the risk can be calculated and capped. An exchange operator who is aware of the discussed risk can use Clause Blind Schnorr Signatures safely. Clause Blind Schnorr Signatures are best suited for denominations with low value, where many coins are being withdrawn/refreshed.

7. Conclusion

This section provides a summary of this work, presents the results and gives an outlook on future work.

7.1. Summary

In the beginning of the project good knowledge on the current state in research about Blind Schnorr signatures was needed. Therefore, various papers were read and then the paper "Blind Schnorr Signatures and Signed ElGamal Encryption in the Algebraic Group Model" [FPS19] was chosen as basis for the redesign of the Taler protocols.
The next step was to analyze the current Taler protocols and understand the required properties including *abort-idempotency*.
With the gathered knowledge (see chapter 2) the Taler protocols were redesigned to support Clause Blind Schnorr Signatures (see chapter 3). These redesigned protocols were then further specified (chapter 4) and then implemented (chapter 5). The implementation includes the main protocols, key management, cryptographic utilities in Taler and the Clause Blind Schnorr Signatures cryptographic routines.
The Clause Blind Schnorr Signatures scheme was analyzed and compared in detail to the RSA Blind Signature scheme (see 6).

7.2. Results

The thesis provides several results to add support for Schnorr's blind signature in Taler, including:

- Redesigned Taler protocols to support Clause Blind Schnorr Signatures
- Implementation of cryptographic routines
- Implementation of Taler protocols in Exchange
  - Key Management and security module
  - Cryptographic utilities
  - Withdraw protocol
  - Deposit protocol
- Comparison and Analysis

  - Performance (speed, space, latency & bandwith)
  - Security
  - Scheme Comparison

- Fixing a minor security issue in Taler's current protocols

The code is tested, and those tests are integrated in the existing testing framework. Benchmarks are added for the cryptographic routines and the security module.

7.3. Future Work

Like in any other project, there is always more that could be done. This section provides an outlook on what can be done in future work.

- Implement wallet
- Implementing remaining Clause Blind Schnorr Signatures protocols (refresh, tipping protocol, refund etc.)
- Implementing merchant
- Security audit of CS implementation
- Find a solution for withdraw loophole
- Evaluating & implementing Clause Blind Schnorr Signatures on other curves

There are some remaining protocols to implement, which were out of scope for this thesis. To run Clause Blind Schnorr Signatures in production, these protocols have to be implemented too. Further, the merchant needs to support Clause Blind Schnorr Signatures too. The merchant implementation can be done fast, as the merchant only verifies denomination signatures in most cases.
Currently, the exchange runs both security modules, the Clause Blind Schnorr Signatures and the RSA security modules. To reduce unnecessary overhead, this should be changed so that only one security has to be running. To run Clause Blind Schnorr Signatures in production a security audit from an external company is recommended (as done for other parts in the exchange, see [Gmb20]). A security audit should always be made when implementing big changes like these.
As mentioned in the scope section, the optional goal to find and implement a good solution for the withdraw loophole was dropped. This was due to the scope shift and because the analysis of the problem showed that finding a good solution needs more research and is a whole project in itself (see 1.3 for more information).
Furthermore, Clause Blind Schnorr Signatures could be implemented on other curves. For example Curve448 [Ham15] could be used, as it provides 224 bits of security, wheras Curve25519 [Ber06] provides about 128 bits of security. Curve secp256k1 could further improve Clause Blind Schnorr Signatures performance. While providing support for Curve448

should not be problematic, a potential implementation for secp256k1 needs further analysis (see [BL21] and [Pie20] for more information).

7.4. Personal Conclusion

This thesis includes understanding, analyzing, integrating and implementing a recent academic paper [FPS19] containing a modern cryptographic scheme. Furthermore, the implementation is done in Taler, an intuitive and modern solution for a social responsible payment system with high ethical standards. Although there was a lot of work, we enjoyed working on such a modern and very interesting topic. Especially the first successful signature verification and the signature scheme performance benchmarks motivated us to push the implementation and integration into Taler forward.
We are happy to provide an implementation of a modern scheme and making it available as free software.

List of Figures

List of Tables

Listings

Bibliography

[Ben+20] Fabrice Benhamouda et al. *On the (in)security of ROS*. Cryptology ePrint Archive, Report 2020/945. `https://ia.cr/2020/945`. 2020.

[Ben21] Dr. Emmanuel Benoist. *Adding Schnorr's blind signature in Taler*. `https://fbi.bfh.ch/fbi/2022/Studienbetrieb/BaThesisHS21/aufgabestellungen/BIE1-1-21-en.html`. 2021.

[Ber06] Daniel J. Bernstein. *Curve25519: new Diffie-Hellman speed records*. `https://cr.yp.to/ecdh/curve25519-20060209.pdf`. 2.09.2006.

[Big] BigCommerce. *Payment fraud: What is it and how it can be avoided?* `https://www.bigcommerce.com/ecommerce-answers/payment-fraud-what-it-and-how-it-can-be-avoided/`.

[Bit] Bitcoin.org. *0.21.1 Release Notes*. `https://bitcoin.org/en/releases/0.21.1/`.

[BL21] Daniel J. Bernstein and Tanja Lange. *SafeCurves: choosing safe curves for elliptic-curve cryptography*. `https://safecurves.cr.yp.to`. accessed 17 October 2021.

[Bus21] Walter Businger. *Skript Public-Key Kryptographie*. 2021.

[Chr21] Florian Dold Christian Grothoff. *Why a Digital Euro should be Online-first and Bearer-based*. `https://taler.net/papers/euro-bearer-online-2021.pdf`. 2021.

[CP93] David Chaum and Torben Pryds Pedersen. "Wallet Databases with Observers". In: *Advances in Cryptology — CRYPTO' 92*. Ed. by Ernest F. Brickell. Berlin, Heidelberg: Springer Berlin Heidelberg, 1993, pp. 89–105. ISBN: 978-3-540-48071-6.

[Cré] Claude Crépeau. "Cut-and-choose protocols". In: School of Computr Science, McGill University, Montréal (QC), Canada. URL: `http://crypto.cs.mcgill.ca/~crepeau/EoC/Cut&Choose.pdf`.

[Dan14] Wesley Janssen Daniel J. Bernstein Bernard van Gastel. *TweetNaCl: a crypto library in 100 tweets*. `https://tweetnacl.cr.yp.to/papers.html`. 17.09.2014.

[Dav83] Chaum David. *Blind Signatures for Untraceable Payments*. `https://www.chaum.com/publications/Chaum-blind-signatures.PDF`. 1983.

[Dem21] Gian Demarmels. *Nonce-Sense - Romhack CTF Crypto Challenge*. `https://blog.c4pr1c0rn.ch/writeups/romhack_21/nonce_sence.html`. [Online; accessed 19-January-2022]. 2021.

[doc] libsodium documentation. *Finite field arithmetic*. https://doc.libsodium.org/advanced/point-arithmetic.

[Dol19] Florian Dold. "The GNU Taler System". PhD thesis. Université de Rennes, 2019.

[FK11] Sheila Frankel and Suresh Krishnan. *IP Security (IPsec) and Internet Key Exchange (IKE) Document Roadmap*. RFC 6071. Feb. 2011. DOI: 10.17487/RFC6071. URL: https://rfc-editor.org/rfc/rfc6071.txt.

[FPS19] Georg Fuchsbauer, Antoine Plouviez, and Yannick Seurin. *Blind Schnorr Signatures and Signed ElGamal Encryption in the Algebraic Group Model*. Cryptology ePrint Archive, Report 2019/877. https://ia.cr/2019/877 and https://www.youtube.com/watch?v=W-uwVdGeUUs. 2019.

[Gan22] Ganapati. *RsaCtfTool*. https://github.com/Ganapati/RsaCtfTool. accessed 13 January 2022.

[Gmb20] Code Blau GmbH. *Report for the GNU Taler security audit in Q2/Q3 2020*. https://taler.net/papers/codeblau-report-2020-q2.pdf. 2020.

[Ham15] Mike Hamburg. *Ed448-Goldilocks, a new elliptic curve*. Cryptology ePrint Archive, Report 2015/625. https://ia.cr/2015/625. 2015.

[JL17] Simon Josefsson and Ilari Liusvaara. *Edwards-Curve Digital Signature Algorithm (EdDSA)*. RFC 8032. Jan. 2017. DOI: 10.17487/RFC8032. URL: https://rfc-editor.org/rfc/rfc8032.txt.

[Lim22] Matt Lim. *Getting Started With Unix Domain Sockets*. https://medium.com/swlh/getting-started-with-unix-domain-sockets-4472c0db4eb1. accessed 08 January 2022.

[Mad20] Neil Madden. *What's the Curve25519 clamping all about?* https://neilmadden.blog/2020/05/28/whats-the-curve25519-clamping-all-about/. 2020.

[OBE18] Prof Bill Buchanan OBE. *Not Playing Randomly: The Sony PS3 and Bitcoin Crypto Hacks*. https://medium.com/asecuritysite-when-bob-met-alice/not-playing-randomly-the-sony-ps3-and-bitcoin-crypto-hacks-c1fe92bea9bc. 12.11.2018.

[Per22] Ben Perez. *Seriously, stop using RSA*. https://blog.trailofbits.com/2019/07/08/fuck-rsa/. accessed 13 January 2022.

[Pie20] Tim Ruffing Pieter Wuille Jonas Nick. *Schnorr Signatures for secp256k1*. Bitcoin Improvement Proposal, bip-0340. https://github.com/bitcoin/bips/blob/master/bip-0340.mediawiki. 2020.

[pre22] preetikagupta8171. *What is RTT(Round Trip Time)?* https://www.geeksforgeeks.org/what-is-rttround-trip-time/. accessed 13 January 2022.

[RD08] Eric Rescorla and Tim Dierks. *The Transport Layer Security (TLS) Protocol Version 1.2*. RFC 5246. Aug. 2008. DOI: 10.17487/RFC5246. URL: https://rfc-editor.org/rfc/rfc5246.txt.

[Repa] GNU Taler Git Repositories. *auditor.git*. https://git.taler.net/auditor.git/.

[Repb] GNU Taler Git Repositories. *backoffice.git*. https://git.taler.net/backoffice.git/.

[Repc] GNU Taler Git Repositories. *django-payments-taler.git*. https://git.taler.net/django-payments-taler.git/.

[Repd] GNU Taler Git Repositories. *exchange.git*. https://git.taler.net/exchange.git/.

[Repe] GNU Taler Git Repositories. *merchant.git*. https://git.taler.net/merchant.git/.

[Repf] GNU Taler Git Repositories. *saleor-frontend.git*. https://git.taler.net/saleor-frontend.git/.

[Repg] GNU Taler Git Repositories. *taler-android.git*. https://git.taler.net/taler-android.git.

[Reph] GNU Taler Git Repositories. *taler-ios.git*. https://git.taler.net/taler-ios.git/.

[Repi] GNU Taler Git Repositories. *taler-merchant-demos.git*. https://git.taler.net/taler-merchant-demos.git/.

[Repj] GNU Taler Git Repositories. *wallet-core.git*. https://git.taler.net/wallet-core.git/.

[Repk] GNU Taler Git Repositories. *woocommerce-taler.git*. https://git.taler.net/woocommerce-taler.git/.

[Repl] GNUnet Git Repositories. *gnunet.git*. https://git.gnunet.org/gnunet.git/.

[Repm] Bitcoin Repository. *BIP-340 - Module for Schnorr signatures in libsecp256k1*. https://github.com/bitcoin/bitcoin/tree/master/src/secp256k1.

[RFC2104] R. Canetti H. Krawczyk M.Bellare. *HMAC: Keyed-Hashing for Message Authentication*. RFC 2104. IETF, Feb. 1997. URL: https://tools.ietf.org/html/rfc2104.

[RFC5869] P.Eronen H. Krawczyk. *HMAC-based Extract-and-Expand Key Derivation Function (HKDF)*. RFC 5869. IETF, May 2010. URL: https://tools.ietf.org/html/rfc5869.

[RFC7748] Adam Langley, Mike Hamburg, and Sean Turner. *Elliptic Curves for Security*. RFC 7748. Jan. 2016. DOI: 10.17487/RFC7748. URL: https://rfc-editor.org/rfc/rfc7748.txt.

[SAa] Taler Systems SA. *Back-office Web service manual*. https://docs.taler.net/taler-backoffice-manual.html.

[SAb] Taler Systems SA. *GNU Taler Auditor Operator Manual*. https://docs.taler.net/taler-auditor-manual.html.

[SAc] Taler Systems SA. *GNU Taler Documentation*. `https://docs.taler.net/`.

[SAd] Taler Systems SA. *GNU Taler Exchange Operator Manual*. `https://docs.taler.net/taler-exchange-manual.html`.

[SAe] Taler Systems SA. *GNU Taler Merchant API Tutorial*. `https://docs.taler.net/taler-merchant-api-tutorial.html`.

[SAf] Taler Systems SA. *GNU Taler Merchant Backend Operator Manual*. `https://docs.taler.net/taler-merchant-manual.html`.

[SAg] Taler Systems SA. *GNU Taler Merchant POS Manual*. `https://docs.taler.net/taler-merchant-pos-terminal.html`.

[SAh] Taler Systems SA. *GNU Taler Wallet CLI Manual*. `https://docs.taler.net/taler-wallet-cli-manual.html`.

[SAi] Taler Systems SA. *GNU Taler Wallet Developer Manual*. `https://docs.taler.net/taler-wallet.html`.

[Sch01] Claus Peter Schnorr. "Security of Blind Discrete Log Signatures against Interactive Attacks". In: *ICICS 2001, LNCS 2229*. Springer-Verlag, 2001, pp. 1–12.

[Sch04] Claus Peter Schnorr. *Enhancing the Security of Perfect Blind DL-Signatures*. Universität Frankfurt. `https://www.math.uni-frankfurt.de/~dmst/teaching/SS2012/Vorlesung/EBS5.pdf`. 2004.

[Sma16] Nigel P. Smart. *Cryptography Made Simple*. Ed. by Kenny Paterson David Basin. Springer International Publishing Switzerland AG, 2016.

[Tal21a] GNU Taler. *GNU Taler*. `https://git.taler.net/marketing.git/tree/presentations/comprehensive/main.pdf`. 2021.

[Tal21b] GNU Taler. *Operations*. `https://git.taler.net/marketing.git/plain/presentations/comprehensive/operations.png`. [Online; accessed 2-November-2021].

[Tal21c] GNU Taler. *Simple Taler Diagram*. `https://taler.net/images/diagram-simple.png`. [Online; accessed 2-November-2021].

[Tal21d] GNU Taler. *Taler Refresh protocol*. `https://git.taler.net/marketing.git/plain/presentations/comprehensive/main.pdf`. [Online; accessed 2-November-2021].

[Tal22a] GNU Taler. *Coin State Machine*. `https://git.taler.net/exchange.git/tree/doc/system/taler/coin.pdf`. [Online; accessed 13 January 2022].

[Tal22b] GNU Taler. *Deposit State Machine*. `https://git.taler.net/exchange.git/tree/doc/system/taler/deposit.pdf`. [Online; accessed 13 January 2022].

[Tib17] Mehdi Tibouchi. *Attacks on Schnorr signatures with biased nonces*. `https://ecc2017.cs.ru.nl/slides/ecc2017-tibouchi.pdf`. 13.11.2017.

[Wag02] David Wagner. *A Generalized Birthday Problem*. University of California Berkeley. https://www.iacr.org/archive/crypto2002/24420288/24420288.pdf. 2002.

[Wan+19] Ziyu Wang et al. “ECDSA weak randomness in Bitcoin”. In: *Future Generation Computer Systems* 102 (Sept. 2019). DOI: 10.1016/j.future.2019.08.034.

[Wik20] Wikipedia. *Know your customer — Wikipedia, Die freie Enzyklopädie*. [Online; Stand 3. April 2021]. 2020. URL: %5Curl%7Bhttps://de.wikipedia.org/w/index.php?title=Know_your_customer&oldid=205456999%7D.

[Wik21a] Wikipedia contributors. *Birthday attack — Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/w/index.php?title=Birthday_attack&oldid=1019272750. [Online; accessed 24-April-2021]. 2021.

[Wik21b] Wikipedia contributors. *Blind signature — Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/w/index.php?title=Blind_signature&oldid=1001105629. [Online; accessed 12-April-2021]. 2021.

[Wik21c] Wikipedia contributors. *EdDSA — Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/w/index.php?title=EdDSA&oldid=1013094030. [Online; accessed 22-April-2021]. 2021.

[Wik21d] Wikipedia contributors. *RSA Factoring Challenge — Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/w/index.php?title=RSA_Factoring_Challenge&oldid=1055393696. [Online; accessed 16-January-2022]. 2021.

[Z21] Yuchen Z. *A Deep Dive Into Idempotence*. https://betterprogramming.pub/a-deep-dive-into-idempotence-1a39393df7e6. [Online; accessed 16-January-2022]. 2021.

Abbreviations

AES	Advanced Encryption Standard
AML	Anti Money Laundering
API	Application Programming Interface
CA	Certificate Authority
CDH	Computational Diffie-Hellman
CFT	Combating Financing of Terrorism
CMA	Choosen-Message Attack
CS	Clause Blind Schnorr Signature Scheme
DDH	Decisional Diffie-Hellman
DHKE	Diffie-Hellman key exchange
DLP	Discrete Logarithm Problem
DSA	Digital Signature Algorithm
ECC	Elliptic Curve Cryptography
ECDH	Elliptic Curve Diffie Hellman
EdDSA	Edwards-curve Digital Signature Algorithm
EUF	Existentially Unforgeability
FDH	Full-Domain Hash
GNU AGPL	GNU Affero General Public License
GNU GPL	GNU General Public License
GNU LGPL	GNU Lesser General Public License
IPC	Inter Process Communication
JSON	JavaScript Object Notation
KDF	Key Derivation Function
KYC	Know Your Customer
MAC	Message Authentication Code
MK	Master Key
PKI	Public Key Infrastructure
PRF	Pseudo Random Function

PRNG	Pseudo Random Number Generator
ROS	Random inhomogeneities in an Overdetermined, Solvable system of linear equations
RT	Round-Trip
RTT	Round-Trip Time
SPOF	Single Point of Failure
Taler	GNU Taler
URL	uniform resource locator

A. Installation

These installation instructions are meant to run the code developed within this thesis for development- and review-purposes. For a comprehensive installation instruction follow the Taler documentation [SAc].

These instructions are used and tested on Ubuntu 21.10.

A.1. Dependencies and Setup

The following dependencies need to be installed for GNUnet and Taler Exchange:

```
student@ubuntu:~$ sudo apt update
student@ubuntu:~$ sudo apt install git curl build-essential gcc automake
    make \ texinfo autoconf uncrustify libtool pkgconf gettext gnutls-bin \
    libcurl4-gnutls-dev libgcrypt20-dev libidn2-dev libjansson-dev \
    libnss3-dev sqlite pipenv libltdl-dev libsodium-dev libpq-dev \
    autopoint libunistring-dev libextractor-dev libpng-dev \ libpulse-dev
    libsqlite3-dev recutils python3-jinja2 sqlite yapf3 \ postgresql
    libpq-dev wget libmicrohttpd-dev
student@ubuntu:~$ export LD\_LIBRARY\_PATH=/usr/local/lib
```

Install in a container

The installation can also be done in a docker or podman container with the ubuntu:21.10 image:

```
student@ubuntu:~$ podman run -it --name talertest ubuntu:21.10
```

A.2. Install GNUnet Core

GNUnet core is both a dependency of the Taler exchange and where we implemented the Clause Blind Schnorr Signature Scheme.

```
student@ubuntu:~$ git clone https://git.gnunet.org/gnunet.git
student@ubuntu:~$ cd gnunet
student@ubuntu:~$ ./bootstrap
student@ubuntu:~$ ./configure --enable-benchmarks --prefix=/usr/local
student@ubuntu:~$ make
student@ubuntu:~$ make install
student@ubuntu:~$ make check # Run optionally to verify installation and run tests
```

To run benchmarks run:

```
student@ubuntu:~$ ./src/util/perf_crypto_cs
student@ubuntu:~$ ./src/util/perf_crypto_rsa
```

A.3. Install Taler Exchange

Ensure that the current user has privileges in postgresql. One possible way to do this is:
(where [user] has to be replaced with the name of the system user running the tests)

```
student@ubuntu:~$ service postgresql start
student@ubuntu:~$ sudo su
student@ubuntu:~$ su - postgres
student@ubuntu:~$ psql
student@ubuntu:~$ CREATE ROLE [user] LOGIN SUPERUSER;
student@ubuntu:~$ CREATE DATABASE [user] OWNER [user];
student@ubuntu:~$ exit
```

The Taler exchange can be installed as followed:

```
student@ubuntu:~$ service postgresql start
student@ubuntu:~$ createdb talercheck
student@ubuntu:~$ git clone https://git.taler.net/exchange.git
student@ubuntu:~$ cd exchange
student@ubuntu:~$ ./bootstrap
student@ubuntu:~$ ./configure --with-gnunet=/usr/local --prefix=/usr/local
student@ubuntu:~$ ./make
student@ubuntu:~$ ./make install
student@ubuntu:~$ ./make check # Run optionally to verify installation and run tests
```

To execute the security module benchmarks run:

```
student@ubuntu:~$ cd src/util
student@ubuntu:~$ ./test_helper_cs
student@ubuntu:~$ ./test_helper_rsa
```

B. Performance Measurements

B.1. AMD Ryzen 7 PRO 5850U (Notebook)

Detailed comparison of each operation can be found in table B.1.

Setup

CPU: 8-core AMD Ryzen 7 PRO 5850U
Architecture: amd64
OS: Ubuntu 21.10 Linux 5.13.0-25-generic #26-Ubuntu SMP Fri Jan 7 15:48:31 UTC 2022 x86_64 x86_64 x86_64 GNU/Linux
libsodium:amd64 version: 1.0.18-1build1
libgcrypt:amd64 version: 1.8.7-5ubuntu2

B.2. Intel(R) Core(TM) i7-8565U

Detailed comparison of each operation can be found in table B.2.

Setup

CPU: 8-core Intel(R) Core(TM) i7-8565U CPU @ 1.80GHz
Architecture: amd64
OS: Ubuntu 21.10 Linux 5.13.0-25-generic #26-Ubuntu SMP Fri Jan 7 15:48:31 UTC 2022 x86_64 x86_64 x86_64 GNU/Linux
libsodium:amd64 version: 1.0.18-1build1
libgcrypt:amd64 version: 1.8.7-5ubuntu2

B.3. AMD Ryzen Threadripper 1950X 16-Core Processor

Detailed comparison of each operation can be found in table B.3.

Setup

CPU: AMD Ryzen Threadripper 1950X 16-Core Processor
Architecture: amd64
OS: Linux 5.13.0-trunk-amd64 #1 SMP Debian 5.13.12-1 exp1 (2021-08-20) x86_64 GNU/Linux
libsodium:amd64 version: 1.9.4-5
libgcrypt:amd64 version: 1.0.18-1

B.4. Intel(R) Xeon(R) CPU E5-2630

Detailed comparison of each operation can be found in table B.4.

Setup

CPU: Intel(R) Xeon(R) CPU E5-2630 0 @ 2.30GHz
Architecture: amd64
OS: Linux 5.10.0-8-amd64 #1 SMP Debian 5.10.46-4 (2021-08-03) x86_64
libsodium:amd64 version: 1.0.18-1
libgcrypt:amd64 version: 1.8.7-6

B.5. Intel(R) Pentium(R) 3558U

Detailed comparison of each operation can be found in table B.5.

Setup

CPU: Intel(R) Pentium(R) 3558U @ 1.70GHz
Architecture: amd64
OS: Linux 5.10.0-8-amd64 #1 SMP Debian 5.10.46-3 (2021-07-28) x86_64
libsodium:amd64 version: 1.0.18-1
libgcrypt:amd64 version: 1.8.7-6

B.6. arm64

Detailed comparison of each operation can be found in table B.6.

Setup

CPU: 8-core arm64
Architecture: ARM64
OS: Linux ten64 5.11.0-31-generic #33+testsfp1 SMP Mon Aug 23 16:07:41 UTC 2021 aarch64 aarch64 aarch64 GNU/Linux
libsodium:arm64 version: 1.8.7-2ubuntu2.1
libgcrypt:arm64 version: 1.0.18-1

B.7. AMD Ryzen Embedded R1606G

Detailed comparison of each operation can be found in table B.7.

Setup

CPU: 4-core AMD Ryzen Embedded R1606G with Radeon Vega Gfx
Architecture: amd64
OS: Linux computer 5.13.0-25-generic #26-Ubuntu SMP Fri Jan 7 15:48:31 UTC 2022 x86_64 x86_64 x86_64 GNU/Linux
libsodium:amd64 version: 1.8.7-5ubuntu2
libgcrypt:amd64 version: 1.0.18-1build1

B.8. risc64

Detailed comparison of each operation can be found in table B.8.

Setup

CPU: 4-core risc64 processor
OS: Linux risc-v-unleashed-000 5.11.0-1022-generic #23 20.04.1-Ubuntu SMP Thu Oct 21 10:16:27 UTC 2021 riscv64 riscv64 riscv64 GNU/Linux
libsodium:riscv64 version: 1.8.7-5ubuntu2
libgcrypt:riscv64 version: 1.0.18-1build1

B.9. POWER9

Detailed comparison of each operation can be found in table B.9.

Setup

CPU: 176-core power9
architecture: pp64le
OS: Linux power9 5.11.0-34-generic #36-Ubuntu SMP Thu Aug 26 19:19:54 UTC 2021 ppc64le ppc64le ppc64le GNU/Linux
libsodium:a::ppc64el version: 1.8.7-2ubuntu2.1
libgcrypt::ppc64el version: 1.0.18-1

B.10. ARMv7 Processor

Detailed comparison of each operation can be found in table B.10.

Setup

CPU: 8-core ARMv7 Processor rev 3 (v7l) Architecture: armv7
OS: Linux odroidxu4 4.14.150-odroidxu4 #2 SMP PREEMPT Mon Oct 28 08:07:45 CET 2019 armv7l GNU/Linux
libsodium:armhf version: 1.9.4-5
libgcrypt:armhf version: 1.0.18-1

B.11. Performance of the Security Module

These performance measurements are only done on one hardware setup. The performance tests of the cryptographic routines are more meaningful, the architecture of the Taler exchange could change a lot. Furthermore, there could be made performance improvements at costs of security by doing the operations requiring the private keys directly in the httpd process. Because of security reasons, the current design with the security module makes a lot of sense. It has to be kept in mind that the following performance benchmarks are interesting to see, but could vary a lot with changes inside the codebase. The performance of the signatures with the security module can be found in table B.11

Setup

CPU: 8-core AMD Ryzen 7 PRO 5850U
OS: Ubuntu 21.10 Linux 5.13.0-25-generic #26-Ubuntu SMP Fri Jan 7 15:48:31 UTC 2022 x86_64 x86_64 x86_64 GNU/Linux
libsodium version: 1.0.18-1build1
libgcrypt version: 1.8.7-5ubuntu2

Signature Scheme	Operation	Speed
CS	10x key generation	0.204 ms
RSA 1024 bit	10x key generation	126 ms
RSA 2048 bit	10x key generation	903 ms
RSA 3072 bit	10x key generation	2684 ms
RSA 4096 bit	10x key generation	10 000 ms
CS	10x r0, r1 derive and R1,R2 calculation	0.444 ms
CS	10x derivation of blinding secrets	0.094 ms
CS	10x blinding	3.332 ms
RSA 1024 bit	10x blinding	1.282 ms
RSA 2048 bit	10x blinding	3.012 ms
RSA 3072 bit	10x blinding	5 ms
RSA 4096 bit	10x blinding	9 ms
CS	10x signing	0.077 ms
RSA 1024 bit	10x signing	7 ms
RSA 2048 bit	10x signing	34 ms
RSA 3072 bit	10x signing	86 ms
RSA 4096 bit	10x signing	183 ms
CS	10x unblinding	0.001 ms
RSA 1024 bit	10x unblinding	2.991 ms
RSA 2048 bit	10x unblinding	10 ms
RSA 3072 bit	10x unblinding	24 ms
RSA 4096 bit	10x unblinding	44 ms
CS	10x verifying	1.358 ms
RSA 1024 bit	10x verifying	0.876 ms
RSA 2048 bit	10x verifying	1.836 ms
RSA 3072 bit	10x verifying	3.075 ms
RSA 4096 bit	10x verifying	5 ms

Table B.1.: Comparison on AMD Ryzen 7

Signature Scheme	Operation	Speed
CS	10x key generation	1.05 ms
RSA 1024 bit	10x key generation	189 ms
RSA 2048 bit	10x key generation	1555 ms
RSA 3072 bit	10x key generation	5000 ms
RSA 4096 bit	10x key generation	11 000 ms
CS	10x r0, r1 derive and R1,R2 calculation	2.261 ms
CS	10x derivation of blinding secrets	0.521 ms
CS	10x blinding	13 ms
RSA 1024 bit	10x blinding	2.6 ms
RSA 2048 bit	10x blinding	4.12 ms
RSA 3072 bit	10x blinding	7 ms
RSA 4096 bit	10x blinding	11 ms
CS	10x signing	0.405 ms
RSA 1024 bit	10x signing	9 ms
RSA 2048 bit	10x signing	44 ms
RSA 3072 bit	10x signing	108 ms
RSA 4096 bit	10x signing	216 ms
CS	10x unblinding	0.005 ms
RSA 1024 bit	10x unblinding	3.353 ms
RSA 2048 bit	10x unblinding	12 ms
RSA 3072 bit	10x unblinding	27 ms
RSA 4096 bit	10x unblinding	47 ms
CS	10x verifying	4.413 ms
RSA 1024 bit	10x verifying	1.202 ms
RSA 2048 bit	10x verifying	2.304 ms
RSA 3072 bit	10x verifying	4.094 ms
RSA 4096 bit	10x verifying	6 ms

Table B.2.: Comparison on Intel(R) Core(TM) i7-8565U

Signature Scheme	Operation	Speed
CS	10x key generation	0.442 ms
RSA 1024 bit	10x key generation	145 ms
RSA 2048 bit	10x key generation	1167 ms
RSA 3072 bit	10x key generation	6000 ms
RSA 4096 bit	10x key generation	11 000 ms
CS	10x r0, r1 derive and R1,R2 calculation	1.043 ms
CS	10x derivation of blinding secrets	0.242 ms
CS	10x blinding	7 ms
RSA 1024 bit	10x blinding	2.258 ms
RSA 2048 bit	10x blinding	4.744 ms
RSA 3072 bit	10x blinding	9 ms
RSA 4096 bit	10x blinding	14 ms
CS	10x signing	0.270 ms
RSA 1024 bit	10x signing	10 ms
RSA 2048 bit	10x signing	47 ms
RSA 3072 bit	10x signing	119 ms
RSA 4096 bit	10x signing	248 ms
CS	10x unblinding	0.003 ms
RSA 1024 bit	10x unblinding	4.086 ms
RSA 2048 bit	10x unblinding	14 ms
RSA 3072 bit	10x unblinding	34 ms
RSA 4096 bit	10x unblinding	60 ms
CS	10x verifying	2.392 ms
RSA 1024 bit	10x verifying	1.137 ms
RSA 2048 bit	10x verifying	2.797 ms
RSA 3072 bit	10x verifying	5 ms
RSA 4096 bit	10x verifying	7 ms

Table B.3.: Comparison on AMD Ryzen Threadripper 1950X

Signature Scheme	Operation	Speed
CS	10x key generation	0.606 ms
RSA 1024 bit	10x key generation	329 ms
RSA 2048 bit	10x key generation	3210 ms
RSA 3072 bit	10x key generation	12 000 ms
RSA 4096 bit	10x key generation	40 000 ms
CS	10x r0, r1 derive and R1,R2 calculation	1.527 ms
CS	10x derivation of blinding secrets	0.329 ms
CS	10x blinding	9 ms
RSA 1024 bit	10x blinding	4.026 ms
RSA 2048 bit	10x blinding	9 ms
RSA 3072 bit	10x blinding	18 ms
RSA 4096 bit	10x blinding	27 ms
CS	10x signing	0.274 ms
RSA 1024 bit	10x signing	21 ms
RSA 2048 bit	10x signing	96 ms
RSA 3072 bit	10x signing	237 ms
RSA 4096 bit	10x signing	482 ms
CS	10x unblinding	0.004 ms
RSA 1024 bit	10x unblinding	7 ms
RSA 2048 bit	10x unblinding	25 ms
RSA 3072 bit	10x unblinding	58 ms
RSA 4096 bit	10x unblinding	99 ms
CS	10x verifying	4.334 ms
RSA 1024 bit	10x verifying	2.190 ms
RSA 2048 bit	10x verifying	5 ms
RSA 3072 bit	10x verifying	11 ms
RSA 4096 bit	10x verifying	14 ms

Table B.4.: Comparison on Intel(R) Xeon(R) CPU E5-2630

Signature Scheme	Operation	Speed
CS	10x key generation	0.53 ms
RSA 1024 bit	10x key generation	524 ms
RSA 2048 bit	10x key generation	3357 ms
RSA 3072 bit	10x key generation	15 000 ms
RSA 4096 bit	10x key generation	37 000 ms
CS	10x r0, r1 derive and R1,R2 calculation	1.375 ms
CS	10x derivation of blinding secrets	0.349 ms
CS	10x blinding	8 ms
RSA 1024 bit	10x blinding	4.86 ms
RSA 2048 bit	10x blinding	11 ms
RSA 3072 bit	10x blinding	19 ms
RSA 4096 bit	10x blinding	31 ms
CS	10x signing	0.283 ms
RSA 1024 bit	10x signing	26 ms
RSA 2048 bit	10x signing	117 ms
RSA 3072 bit	10x signing	292 ms
RSA 4096 bit	10x signing	571 ms
CS	10x unblinding	0.003 ms
RSA 1024 bit	10x unblinding	8 ms
RSA 2048 bit	10x unblinding	30 ms
RSA 3072 bit	10x unblinding	67 ms
RSA 4096 bit	10x unblinding	111 ms
CS	10x verifying	3.769 ms
RSA 1024 bit	10x verifying	2.616 ms
RSA 2048 bit	10x verifying	6 ms
RSA 3072 bit	10x verifying	11 ms
RSA 4096 bit	10x verifying	17 ms

Table B.5.: Comparison on Intel(R) Pentium(R) 3558U

Signature Scheme	Operation	Speed
CS	10x key generation	2.896 ms
RSA 1024 bit	10x key generation	839 ms
RSA 2048 bit	10x key generation	8000 ms
RSA 3072 bit	10x key generation	17 000 ms
RSA 4096 bit	10x key generation	82 000 ms
CS	10x r0, r1 derive and R1,R2 calculation	6 ms
CS	10x derivation of blinding secrets	0.713 ms
CS	10x blinding	23 ms
RSA 1024 bit	10x blinding	11 ms
RSA 2048 bit	10x blinding	28 ms
RSA 3072 bit	10x blinding	51 ms
RSA 4096 bit	10x blinding	81 ms
CS	10x signing	0.321 ms
RSA 1024 bit	10x signing	57 ms
RSA 2048 bit	10x signing	263 ms
RSA 3072 bit	10x signing	685 ms
RSA 4096 bit	10x signing	1385 ms
CS	10x unblinding	0.006 ms
RSA 1024 bit	10x unblinding	23 ms
RSA 2048 bit	10x unblinding	79 ms
RSA 3072 bit	10x unblinding	171 ms
RSA 4096 bit	10x unblinding	296 ms
CS	10x verifying	11ms
RSA 1024 bit	10x verifying	5 ms
RSA 2048 bit	10x verifying	15 ms
RSA 3072 bit	10x verifying	27 ms
RSA 4096 bit	10x verifying	45 ms

Table B.6.: Comparison on arm64

Signature Scheme	Operation	Speed
CS	10x key generation	2.373 ms
RSA 1024 bit	10x key generation	184 ms
RSA 2048 bit	10x key generation	2132 ms
RSA 3072 bit	10x key generation	8000 ms
RSA 4096 bit	10x key generation	21 000 ms
CS	10x r0, r1 derive and R1,R2 calculation	1.09 ms
CS	10x derivation of blinding secrets	0.43 ms
CS	10x blinding	6 ms
RSA 1024 bit	10x blinding	3.886 ms
RSA 2048 bit	10x blinding	7 ms
RSA 3072 bit	10x blinding	14 ms
RSA 4096 bit	10x blinding	23 ms
CS	10x signing	0.379 ms
RSA 1024 bit	10x signing	15 ms
RSA 2048 bit	10x signing	71 ms
RSA 3072 bit	10x signing	177 ms
RSA 4096 bit	10x signing	357 ms
CS	10x unblinding	0.001 ms
RSA 1024 bit	10x unblinding	6 ms
RSA 2048 bit	10x unblinding	24 ms
RSA 3072 bit	10x unblinding	53 ms
RSA 4096 bit	10x unblinding	93 ms
CS	10x verifying	2.610 ms
RSA 1024 bit	10x verifying	2.303 ms
RSA 2048 bit	10x verifying	4.386 ms
RSA 3072 bit	10x verifying	7 ms
RSA 4096 bit	10x verifying	11 ms

Table B.7.: Comparison on AMD Ryzen Embedded R1606G

Signature Scheme	Operation	Speed
CS	10x key generation	4.144 ms
RSA 1024 bit	10x key generation	2923 ms
RSA 2048 bit	10x key generation	28 000 ms
RSA 3072 bit	10x key generation	174 000 ms
RSA 4096 bit	10x key generation	600 000 ms
CS	10x r0, r1 derive and R1,R2 calculation	10 ms
CS	10x derivation of blinding secrets	2.514 ms
CS	10x blinding	72 ms
RSA 1024 bit	10x blinding	37 ms
RSA 2048 bit	10x blinding	93 ms
RSA 3072 bit	10x blinding	170 ms
RSA 4096 bit	10x blinding	277 ms
CS	10x signing	1.697 ms
RSA 1024 bit	10x signing	215 ms
RSA 2048 bit	10x signing	1040 ms
RSA 3072 bit	10x signing	2883 ms
RSA 4096 bit	10x signing	5000 ms
CS	10x unblinding	0.022 ms
RSA 1024 bit	10x unblinding	62 ms
RSA 2048 bit	10x unblinding	150 ms
RSA 3072 bit	10x unblinding	275 ms
RSA 4096 bit	10x unblinding	431 ms
CS	10x verifying	29 ms
RSA 1024 bit	10x verifying	22 ms
RSA 2048 bit	10x verifying	54 ms
RSA 3072 bit	10x verifying	99 ms
RSA 4096 bit	10x verifying	166 ms

Table B.8.: Comparison on risc64

Signature Scheme	Operation	Speed
CS	10x key generation	0.275 ms
RSA 1024 bit	10x key generation	290 ms
RSA 2048 bit	10x key generation	3743 ms
RSA 3072 bit	10x key generation	15 000 ms
RSA 4096 bit	10x key generation	45 000 ms
CS	10x r0, r1 derive and R1,R2 calculation	0.749 ms
CS	10x derivation of blinding secrets	0.267 ms
CS	10x blinding	4.996 ms
RSA 1024 bit	10x blinding	3.952 ms
RSA 2048 bit	10x blinding	10 ms
RSA 3072 bit	10x blinding	17 ms
RSA 4096 bit	10x blinding	27 ms
CS	10x signing	0.221 ms
RSA 1024 bit	10x signing	25 ms
RSA 2048 bit	10x signing	135 ms
RSA 3072 bit	10x signing	381 ms
RSA 4096 bit	10x signing	762 ms
CS	10x unblinding	0.002 ms
RSA 1024 bit	10x unblinding	9 ms
RSA 2048 bit	10x unblinding	34 ms
RSA 3072 bit	10x unblinding	80 ms
RSA 4096 bit	10x unblinding	141 ms
CS	10x verifying	2.458 ms
RSA 1024 bit	10x verifying	2.365 ms
RSA 2048 bit	10x verifying	6 ms
RSA 3072 bit	10x verifying	10 ms
RSA 4096 bit	10x verifying	16 ms

Table B.9.: Comparison on POWER9

Signature Scheme	Operation	Speed
CS	10x key generation	1.719 ms
RSA 1024 bit	10x key generation	1050 ms
RSA 2048 bit	10x key generation	8000 ms
RSA 3072 bit	10x key generation	53 000 ms
RSA 4096 bit	10x key generation	159 000 ms
CS	10x r0, r1 derive and R1,R2 calculation	3.621 ms
CS	10x derivation of blinding secrets	0.514 ms
CS	10x blinding	24 ms
RSA 1024 bit	10x blinding	10 ms
RSA 2048 bit	10x blinding	26 ms
RSA 3072 bit	10x blinding	45 ms
RSA 4096 bit	10x blinding	78 ms
CS	10x signing	0.481 ms
RSA 1024 bit	10x signing	87 ms
RSA 2048 bit	10x signing	385 ms
RSA 3072 bit	10x signing	1038 ms
RSA 4096 bit	10x signing	2073 ms
CS	10x unblinding	0.008 ms
RSA 1024 bit	10x unblinding	26 ms
RSA 2048 bit	10x unblinding	90 ms
RSA 3072 bit	10x unblinding	195 ms
RSA 4096 bit	10x unblinding	344 ms
CS	10x verifying	11 ms
RSA 1024 bit	10x verifying	5 ms
RSA 2048 bit	10x verifying	15 ms
RSA 3072 bit	10x verifying	28 ms
RSA 4096 bit	10x verifying	42 ms

Table B.10.: Comparison on ARMv7

Signature Scheme	**Test**	**Speed**
CS	100 sequential signature operations	2.591 ms
RSA 1024 bit	100 sequential signature operations	79 ms
RSA 2048 bit	100 sequential signature operations	350 ms
RSA 3072 bit	100 sequential signature operations	893 ms
RSA 4092	100 sequential signature operations	1811 ms
CS	100 parallel signature operations	14 ms
RSA 1024 bit	100 parallel signature operations	125 ms
RSA 2048 bit	100 parallel signature operations	573ms
RSA 3072 bit	100 parallel signature operations	1420 ms
RSA 4092	100 parallel signature operations	3279 ms
CS	800 parallel signature operations	19 ms
RSA 1024 bit	800 parallel signature operations	137 ms
RSA 2048 bit	800 parallel signature operations	653 ms
RSA 3072 bit	800 parallel signature operations	1451 ms
RSA 4092	800 parallel signature operations	3388 ms

Table B.11.: Performance comparison of the security module

C. Redesigned RSA Protocols

In order to bring the RSA and Clause Blind Schnorr Signatures protocols closer, this chapter describes a variant of the RSA protocols with the same changes as in the Clause Blind Schnorr Signatures versions (where they can be applied).

C.1. Withdraw Protocol

The changes to the RSA witdhdraw protocol (see Figure C.1) are limited to the derivation of the coin and blinding factor.

C.2. Refresh Protocol

The changes to the refresh protocol are related to the derivation of transfer secrets and subsequent operations, see Figure C.2, Figure C.3 and Figure C.4.

C.3. Linking Protocol

The changes are described in Figure C.5.

Customer
knows:
reserve keys w_s, W_p
denomination public key $D_p = e, N$

generate withdraw secret:
$\omega := randombytes(32)$
persist $\langle \omega, D_p \rangle$
derive coin key pair:
$c_s :=$ HKDF$(256, \omega, \text{"cs"})$
$C_p :=$ Ed25519.GetPub(c_s)
blind:
$b_s :=$ HKDF$(256, \omega, \text{"b-seed"})$
$r :=$ FDH(b_s)
$m' :=$ FDH$(N, C_p) * r^e \mod N$
sign with reserve private key:
$\rho_W := \langle D_p, m' \rangle$
$\sigma_W :=$ Ed25519.Sign(w_s, ρ_W)

Exchange
knows:
reserve public key W_p
denomination keys d_s, D_p

$$\xrightarrow{\rho = W_p, \sigma_W, \rho_W}$$

$\langle D_p, m' \rangle := \rho_W$
verify if D_p is valid
check Ed25519.Verify(W_p, ρ_W, σ_W)
$\sigma'_c = (m')^{d_s} \mod N$
decrease balance if sufficient and
persist $\langle D_p, s \rangle$

$$\xleftarrow{\sigma'_c}$$

unblind:
$\sigma_c = \sigma'_c * r^{-1}$
verify signature:
check if $\sigma_c^e =$ FDH(N, C_p)

resulting coin: c_s, C_p, σ_c, D_p

implementation note: minimum of
persisted values is $\langle \omega, \sigma_c \rangle$

Figure C.1.: Redesigned RSA withdrawal process

```
RefreshDerive$(t, \langle e, N \rangle, C_p)$
$T :=$ Curve25519.GetPub$(t)$
$x :=$ ECDH-EC$(t, C_p)$
$b_s :=$ HKDF$(256, x,$ ”b-seed”$)$
$r :=$ FDH$(b_s)$
$c'_s :=$ HKDF$(256, x,$ "$c$"$)$
$C'_p :=$ Ed25519.GetPub$(c'_s)$
$\overline{m} := r^e * C'_p \mod N$
return $\langle T, c'_s, C'_p, \overline{m} \rangle$
```

Figure C.2.: Redesigned RSA RefreshDerive algorithm

```
Customer
knows:
denomination public key $D_{p(i)}$
$\text{coin}_0 = \langle D_{p(0)}, c_s^{(0)}, C_p^{(0)}, \sigma_c^{(0)} \rangle$
$\text{Select}\langle N_t, e_t \rangle := D_{p(t)} \in D_{p(i)}$
$\omega := randombytes(32)$
persist $\langle \omega, D_{p(t)} \rangle$
**for** $i = 1, \dots, \kappa :$
    $t_i := \text{HKDF}(256, \omega, \text{"t}i\text{"})$
    $X_i := \text{RefreshDerive}(t_i, D_{p(t)}, C_p^{(0)})$
    $(T_i, c_s^{(i)}, C_p^{(i)}, \overline{m}_i) := X_i$
**endfor**
$h_T := H(T_1, \dots, T_k)$
$h_{\overline{m}} := H(\overline{m}_1, \dots, \overline{m}_k)$
$h_C := H(h_t, h_{\overline{m}})$
$\rho_{RC} := \langle h_C, D_{p(t)}, D_{p(0)}, C_p^{(0)}, \sigma_C^{(0)} \rangle$
$\sigma_{RC} := \text{Ed25519.Sign}(c_s^{(0)}, \rho_{RC})$
Persist refresh-request$\langle \omega, \rho_{RC}, \sigma_{RC} \rangle$

                    $\xrightarrow{\rho_{RC}, \sigma_{RC}}$

Exchange
knows:
denomination keys $d_{s(i)}, D_{p(i)}$
$(h_C, D_{p(t)}, D_{p(0)}, C_p^{(0)}, \sigma_C^{(0)} = \rho_{RC})$
**check**Ed25519.Verify$(C_p^{(0)}, \sigma_{RC}, \rho_{RC})$
$x \rightarrow \text{GetOldRefresh}(\rho_{RC})$
**Comment:** GetOldRefresh$(\rho_{RC} \mapsto \{\bot, \gamma\})$
**if** $x = \bot$
    $v := \text{Denomination}(D_{p(t)})$
    $\langle e_0, N_0 \rangle := D_{p(0)}$
    **check** IsOverspending$(C_p^{(0)}, D_{p(0)}, v)$
    **check** $D_{p(t)} \in \{D_{p(i)}\}$
    **check** FDH$(N_0, C_p^{(0)}) \equiv_{N_0} (\sigma_0^{(0)})^{e_0}$
    MarkFractionalSpend$(C_p^{(0)}, v)$
    $\gamma \leftarrow \{1, \dots, \kappa\}$
    Persist refresh-record $\langle \rho_{RC}, \gamma \rangle$
**else**
    $\gamma := x$
**endif**

                    $\xleftarrow{\gamma}$
```

Continued in figure 2.16

Figure C.3.: Redesigned RSA refresh protocol (commit phase)

Customer Exchange

Continuation of figure 2.15

$$\xleftarrow{\quad\gamma\quad}$$

check IsConsistentChallenge(ρ_{RC}, γ)
Comment: IsConsistentChallenge
$(\rho_{RC}, \gamma) \mapsto \{\bot, \top\}$

Persist refresh-challenge$\langle \rho_{RC}, \gamma \rangle$
$S := \langle t_1, \ldots, t_{\gamma-1}, t_{\gamma+1}, \ldots, t_\kappa \rangle$
$\rho_L = \langle C_p^{(0)}, D_{p(t)}, T_\gamma, \overline{m}_\gamma \rangle$
$\rho_{RR} = \langle T_\gamma, \overline{m}_\gamma, S \rangle$
$\sigma_L =$ Ed25519.Sign$(c_s^{(0)}, \rho_L)$

$$\xrightarrow{\quad\rho_{RR},\rho_L,\sigma_L\quad}$$

$\langle T'_\gamma, \overline{m}'_\gamma, S \rangle := \rho_{RR}$
$\langle t_1, \ldots, t_{\gamma-1}, t_{\gamma+1}, \ldots, t_\kappa \rangle) := S$
check Ed25519.Verify$(C_p^{(0)}, \sigma_L, \rho_L)$
for$i = 1, \ldots, \gamma - 1, \gamma + 1, \ldots, \kappa$
$X_i :=$ RefreshDerive$(t_i, D_{p(t)}, C_p^{(0)})$
$\langle T_i, c_s^{(i)}, C_p^{(i)}, \overline{m}_i \rangle := X_i$
endfor
$h'_T = H(T_1, \ldots, T_{\gamma-1}, T'_\gamma, T_{\gamma+1}, \ldots, T_\kappa)$
$h'_{\overline{m}} = H(\overline{m}_1, \ldots, \overline{m}_{\gamma-1}, \overline{m}'_\gamma, \overline{m}_{\gamma+1}, \ldots, \overline{m}_\kappa)$
$h'_C = H(h'_T, h'_{\overline{m}})$
check $h_C = h'_C$
$\overline{\sigma}_C^{(\gamma)} := \overline{m}^{d_{s(t)}}$
persist $\langle \rho_L, \sigma_L, S \rangle$

$$\xleftarrow{\quad\overline{\sigma}_C^{(\gamma)}\quad}$$

$\sigma_C^{(\gamma)} := r^{-1}\overline{\sigma}_C^{(\gamma)}$
check if $(\sigma_C^{(\gamma)})^{e_t} \equiv_{N_t} C_p^{(\gamma)}$
Persist coin$\langle D_{p(t)}, c_s^{(\gamma)}, C_p^{(\gamma)}, \sigma_C^{(\gamma)} \rangle$

Figure C.4.: Redesigned RSA refresh protocol (reveal phase)

Customer knows:
coin$_0 = \langle D_{p(0)}, c_s^{(0)}, C_p^{(0)}, \sigma_C^{(0)} \rangle$

Exchange knows:

$$\xrightarrow{C_{p(0)}}$$

$L :=$ LookupLink$(C_{p(0)})$
Comment: LookupLink$(C_p) \mapsto \{\langle \rho_L^{(i)}, \sigma_L^{(i)}, \overline{\sigma}_C^{(i)} \rangle\}$

$$\xleftarrow{L}$$

for $\langle \rho_L^{(i)}, \overline{\sigma}_L^{(i)}, \sigma_C^{(i)} \rangle \in L$
$\langle \hat{C}_p^{(i)}, D_{p(t)}^{(i)}, T_\gamma^{(i)}, \overline{m}_\gamma^{(i)} \rangle := \rho_L^{(i)}$
$\langle e_t^{(i)}, N_t^{(i)} \rangle := D_{p(t)}^{(i)}$
check $\hat{C}_p^{(i)} \equiv C_p^{(0)}$
check Ed25519.Verify$(C_p^{(0)}, \rho_L^{(i)}, \sigma_L^{(i)})$
$x_i :=$ ECDH$(c_s^{(0)}, T_\gamma^{(i)})$
$c_s^{(i)} :=$ HKDF$(256, x_i, \text{"}c\text{"})$
$C_p^{(i)} :=$ Ed25519.GetPub$(c_s^{(i)})$
$b_s^{(i)} :=$ HKDF$(256, x_i, \text{"b-seed"})$
$r_i :=$ FDH$(b_s^{(i)})$
$\sigma_C^{(i)} := (r_i)^{-1} \cdot \overline{m}_\gamma^{(i)}$
check $(\sigma_C^{(i)})^{e_t^{(i)}} \equiv_{N_t^{(i)}} C_p^{(i)}$
(Re-)obtain coin$\langle D_{p(t)}^{(i)}, c_s^{(i)}, C_p^{(i)}, \sigma_C^{(i)} \rangle$

Figure C.5.: Redesigned RSA linking protocol