Adding Schnorr’s blind signature in Taler - Defence
Gian Demarmels  Lucien Heuzeveldt

Advisor: Prof. Dr. Emmanuel Benoist
Expert: Elektronikingenieur HTL Daniel Voisard
Adding Schnorr's blind signature in Taler - Defence

Goals & Project Management
Preliminaries
Protocol Redesign
Specification & Implementation
Results
Goals & Project Management
Motivation

- Elliptic curve cryptography allows smaller keys
- Leads to huge performance benefits
- Cipher agility
- Recent topic
Our goal is to add support for Schnorr’s Blind Signature scheme to GNU Taler.

- 👍 Analyze current state of research
- 🔧 Redesign Taler’s protocols
- 🔰 Implementation of redesigned protocols
- 👀 Comparison with RSA Blind Signatures
Project Management:

- Waterfall vs. Agile
- Project Analysis
- ClickUp with Kanban-Boards and Gantt-Chart
- Git to manage code and deliverables
- Markdown notes for meetings, thoughts, etc.

Project Phases:

- Phase 1: Initiation
- Phase 2: Planning
- Phase 3: Execution
  - a) Design Phase
  - b) Specification Phase
  - c) Implementation Phase
- Phase 4: Discussion
- Phase 5: Closure
GNU Taler Overview
A privacy-preserving, fast and intuitive payment system

Taler Components

- 🏛️ Exchange
  Payment service provider between customer and merchant

- 🛒 Merchant
  Accepts payments with Taler in exchange for goods and services

- 📦 Wallet
  A customer holds coins in his electronic wallet

- 🕵️‍♀️ Auditor
  The auditors (financial regulators) monitor the exchanges behaviour

graphics source: https://taler.net/images/diagram-simple.png
GNU Taler properties

Properties:
- ⬇️ Free Software
- ⬇️ Buyer Privacy Protection
- ⬇️ Merchant Taxability
- ⚫️ Auditability - Income Transparency
- ⭕️ Prevent payment fraud
- 🌐 Privacy by design
- 👤 Easy to use
- ⚡ Efficient - Even more efficient with our improvements!
- ⚣ Fault-tolerant design
- ⛤️ Foster competition

More details on https://taler.net/en/principles.html
Abort-Idempotency

- **Idempotency**
  Idempotency ensures that the state of a system will not change, no matter how many times the same request was made.
  In other words: The same request will receive the same response.

- **Abort-Idempotency**
  Abort-Idempotency also ensures Idempotency in every abort scenario.
HKDF RFC5869
The HMAC-based Extract-and-Expand Key Derivation Function

- HKDF can be used as a pseudo-random function, a deterministic function whose output appears to be random
- follows the extract-then-expand paradigm
- A fixed-length high-entropy key $K$ is extracted from potentially weaker input keying material
- The key $K$ is then expanded to output a variable-length, pseudo-random key
Curve25519:

- Curve25519 is a Montgomery-Curve over prime field $2^{255} - 19$
- Provides 128 bits of security
- Well-known and trusted
- Good choice in terms of security & speed

Alternatives:

- Curve448-Goldilocks
- Secp256k1 ("Bitcoin curve")

Abbildung: Abbild der elliptischen Kurve

$$y^2 = x^3 + 486662x^2 + x$$

graphics source: https://heise.cloudimg.io/v7/_www-heise-de_/imgs/18/1/4/5/9/6/8/9/curve25519-5b8d94dd2448661c.png
The coin is a EdDSA keypair

- Uses Curve25519
- Public key is the planchet to be signed by the exchange
- The coin can be spent by signing a contract with the coin’s private key

graphics source: https://git.taler.net/marketing.git/plain/presentations/comprehensive/main.pdf
Adding Schnorr's blind signature in Taler - Defence

Customer:

\[ b \]

\[ \text{transmit} \]

Exchange:

\[ \text{Customer} \]

\[ \text{transmit} \]

Customer:

\[ b \]

---

graphics source: https://git.taler.net/marketing.git/plain/presentations/comprehensive/main.pdf

Berner Fachhochschule | Haute école spécialisée bernoise | Bern University of Applied Sciences
RSA Blind Signatures

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

$$f = FDH(m)$$

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

Bob
knows:
RSA keys $d_B, D_B$

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

unblind:
$$s = s' \cdot r^{-1}$$
Schnorr Signature Scheme

User knows:
- public key \( X \)

public parameters:
- \( \langle p, G, G, H \rangle \)

Signer knows:
- private signing key \( x \), \( X := xG \)

\[ r \leftarrow \text{random} \in \mathbb{Z}_p \]
\[ R := rG \]

\[ c := H(R, m) \]

\[ s := r + cx \mod p \]

check \( sG = R + cX \)
\[ \sigma := \langle R, s \rangle \]
The (broken) Blind Schnorr Signature Scheme

**User**

knows:

public key $X$

**Signer**

knows:

public parameters: $\langle p, \mathbb{G}, G, H \rangle$

private signing key $x$, $X := xG$

$r \leftarrow \text{random} \in \mathbb{Z}_p$

$R := rG$

$\alpha, \beta \leftarrow \text{random} \in \mathbb{Z}_p$

$R' := R + \alpha G + \beta X$

$c' := H(R', m)$

$c := c' + \beta \mod p$

$s := r + cx \mod p$

check $sG = R + cX$

$s' := s + \alpha \mod p$

$\sigma := \langle R', s' \rangle$
ROS problem - (informally)
Random inhomogeneities in an Overdetermined, Solvable system of linear equations

ROS problem:
- ROS depends on group order $p$, parameterized with integer $\ell$
- An adversary can produce $\ell + 1$ valid signatures after $\ell > \log_2(p)$ parallel sessions by solving a linear equation system
- $\sum_{j=1}^{\ell} \rho_{i,j} c_j = H_{ros}(\vec{p}_i), i \in [\ell + 1]$
- There exist a polynomial-time attack against $ROS_{\ell}$ when $\ell > \log_2(p)$

Modified ROS:
- Does not apply to the modified ROS problem
- Queries oracle with two vectors instead of one
- The signer returns a signature by randomly flipping a bit $b$
- Only the $c_b$ is signed and returned
- An adversary would need to commit to $c_b$ before learning about $b$

See: On the (in)security of ROS (https://eprint.iacr.org/2020/945)
Clause Blind Schnorr Signature Scheme

User knows:
public key $X$

Signer knows:
private signing key $x$, $X := xG$

Public parameters:
$\langle p, G, G, H \rangle$

$r_0, r_1 \leftarrow \text{random} \in \mathbb{Z}_p$
$R_0 := r_0G$
$R_1 := r_1G$

$r_0, r_1 \leftarrow \text{random} \in \mathbb{Z}_p$
$R'_0 := R_0 + \alpha_0G + \beta_0X$
$R'_1 := R_1 + \alpha_1G + \beta_1X$
$c'_0 := H(R'_0, m)$
$c'_1 := H(R'_1, m)$
$c_0 := c'_0 + \beta_0 \mod p$
$c_1 := c'_1 + \beta_1 \mod p$

$b \leftarrow \text{random} \in \{0, 1\}$
$s := r_b + c_b x \mod p$

Check $sG = R + cX$
$s' := s + \alpha_b \mod p$
$\sigma := \langle R'_b, s' \rangle$
Protocols:

- 🛒 Withdrawal
- 🔥 Refresh
- ⛑ Spend
- 🆕 Deposits
- 🍩 Tipping
- 👤 Payback
- 💸 Recoup

---

graphics source: https://taler.net/images/diagram-simple.png
Adding Schnorr’s blind signature in Taler - Defence

Customer
reserve keys \( w_s, W_p \)
denomination public key \( D_p = e, N \)

generate coin key pair:
\( c_s, C_p \leftarrow \text{Ed25519.KeyGen}() \)
blind:
\( r \leftarrow \text{random} \in \mathbb{Z}_N^* \)
\( m' = \text{FDH}(N, C_p) * r^e \mod N \)
sign with reserve private key:
\( \rho_W = D_p, m' \)
\( \sigma_W = \text{Ed25519.Sign}(w_s, \rho_W) \)

Exchange
reserve public key \( W_p \)
denomination keys \( d_s, D_p \)

\[ \rho = W_p \cdot \sigma_W \cdot \rho_W \]

verify if denomination public key is valid
check \( \text{Ed25519.Verify}(W_p, \rho_W, \sigma_W) \)
decrease balance if sufficient sign:
\( \sigma_c' = (m')^{d_s} \mod N \)

\[ \sigma_c = \sigma_c' \cdot r^{-1} \]
verify signature:
check \( \sigma_c^e = \text{FDH}(N, C_p) \)
resulting coin: \( c_s, C_p, \sigma_c, D_p \)
Refresh Protocol - DH-Lock

Diffie-Hellman Lock:
- keypairs \( C = cG \) and \( T = tG \)
- Both keys can unlock the lock: \( k = tC = cT \)

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

Graphics source: https://git.taler.net/marketing.git/plain/presentations/comprehensive/main.pdf
Refresh Protocol - Cut and Choose

- Customer sets up $k$ DH-Locks
- Exchange sends back random $\gamma \in \{1, \ldots, k\}$
- Customer reveals transfer private keys, except $t_\gamma$
- Exchange can detect fraud attempts with a probability of $1/k$

graphics source: https://git.taler.net/marketing.git/plain/presentations/comprehensive/main.pdf
Adding Schnorr’s blind signature in Taler - Defence

Customer creates $k$: RefreshDerives (DH-Locks)

Customer commits by calculating a commit hash

- $h_T := H(T_1, \ldots, T_k)$
- $h_m := H(m_1, \ldots, m_k)$
- $h_C := H(h_T, h_m)$

The exchange answers with a random $\gamma \in \{1, \ldots, k\}$

\[ h_T := H(T_1, \ldots, T_k) \]
\[ h_m := H(m_1, \ldots, m_k) \]
\[ h_C := H(h_T, h_m) \]

\[ \rho_{RC} := (h_C, D_{p(t)}, D_{p(0)}, C_p^{(0)}, \sigma_C^{(0)}) \]
\[ \sigma_{RC} := \text{Ed25519.Sign}(c^{(0)}_{\rho_{RC}}) \]

Persist refresh-request $(\rho_{RC}, \sigma_{RC})$

Check

- IsOverspending $(C_p^{(0)}, D_{p(0)}, v)$
- check $D_{p(t)} \in \{D_{p(0)}\}$
- check $\text{FDH}(N_0, C_p^{(0)}) \equiv N_0 (\sigma_C^{(0)}) e$

MarkFractionalSpend $(C_p^{(0)}, v)$

Persist refresh-record $(\rho_{RC}, \gamma)$

\[ \gamma := x \]

endif

\[ \gamma \leftarrow \{1, \ldots, k\} \]
Adding Schnorr’s blind signature in Taler - Defence

Customer reveals every transfer key (seed), except $t_γ$

The exchange now proves if the customer is honest by recalculating the RefreshDerives

If the check succeeds, the exchange returns the signature of the new coin

Fraud attempts are detected with probability of $1/k$
Threat: An evil customer sends the old coins private key to a third party.

The third party refreshes the coin and receives a new coin.

Solution: re-obtain refreshed coin with link protocol from $c_s(old)$
Challenges

- Two blinding factors
- Additional request
- Many calculations are done twice
- Many random elements - What about Abort-Idempotency?

How can we redesign Taler’s protocols to work with the Clause Blind Schnorr signature scheme while still preserving all properties?
Protocol Redesign
Protocol Redesign

- Analyze Taler protocols
- Integrate where blind signatures are used
- Proposal
- Rounds of Feedback
- Additional Request during signature creation
- Introduces complexity
- Challenge regarding abort-idempotency
- Vanilla Clause Blind Schnorr Signature Scheme:
  - $r_0 \leftarrow \text{random}$
  - $R_0 := rG$
- Our Changes:
  - Introduces Nonce $n$ used for Derivation
  - Derives $R$:
    - $r_0 := \text{HKDF}(256, n||d_s, "r0")$
    - $R_0 := r_0G$
  - Denomination private key as long-term secret
Withdraw Protocol

- Signature scheme related operations replaced
- Additional round-trip introduced
- Extensively uses HKDF to achieve abort-idempotency
- Randomness in CS replaced with derivation → unpredictable
Withdraw Protocol

Protocol Changes

- Withdraw Nonce (Wallet):
  \[
  c_s, C_p \leftarrow \text{Ed25519.KeyGen}()
  \]
  \[
  n_w := \text{HKDF}(256, c_s, "n")
  \]

- Request R

- Derive R (Exchange)

- Derive Blinding Secrets (Wallet):
  \[
  b_s := \text{HKDF}(256, c_s || R_0 || R_1, "b-seed")
  \]
  \[
  \alpha_0 := \text{HKDF}(256, b_s, "a0")
  \]
  \[
  \beta_1 := \text{HKDF}(256, b_s, "b1")
  \]
Withdraw Protocol

Protocol Changes

- Derive $b$ (exchange):
  $$b := \text{HKDF}(1, n_w || d_s, "b")$$
- Re-derive $r_b$
- Calculate signature scalar
- Unblind, construct signature $\langle R'_b, s' \rangle$

Customer knows:
- reserve keys $w_x, W_p$
- denomination public key $D_p$

Exchange knows:
- reserve public key $W_p$
- denomination keys $d_x, D_p$

Continuation of figure 4.1

sign with reserve private key:
$$\rho_W := \langle n_w, D_p, c_0, c_1 \rangle$$
$$\sigma_W := \text{Ed25519} \cdot \text{Sign}(w_x, \rho_W)$$

$$\langle n_w, D_p, c_0, c_1 \rangle := \rho_W$$
verify if $D_p$ is valid
check Ed25519Verify($W_p, \rho_W, \sigma_W$)
$$b := \text{HKDF}(1, n_w || d_s, "b")$$
s ← GetWithdraw($n_w, D_p$)
if $s = \perp$
$$r_b := \text{HKDF}(256, n_w || d_p, "rb")$$
s := $r_b + c_0d_s$ mod $p$
decrease balance if sufficient and persist $\langle n_w, D_p, s \rangle$
endif

verify signature:
check if $sG = R_b + c_0D_p$
unblind:
$$s' := s + c_0 \text{ mod } p$$
verify signature:
check if $s'G = R'_b + c_0D_p$
$$\sigma_C := \langle R'_b, s' \rangle$$
resulting coin: $c_x, C_p, \sigma_C, D_p$
Withdraw Protocol

Nonce Check

- Is this safe? (without nonce reuse check)
  \[ r_0 := \text{HKDF}(256, n||d_s, "r0") \]

- \( r_1 = r_2 \):
  \[ s_2 - s_1 = d_s(c'_1 - c'_2) - (r_1 - r_2) \]
  Allows private key recovery
  Happened before (Bitcoin, PlayStation 3)

- Prevent \( r \) reuse → do not allow nonce reuse (per denomination)

- Applies to withdraw AND refresh

Prevent nonce reuse check
Deposit Protocol

- Only coin signature verification changes:

\[ s' G = R' + c' D_p \]

\[ = R' + H(R', C_p)D_p \]
Refresh and Linking

- Integration similar to withdraw (additional round trip, derivation, etc.)
- Introduced new random refresh secret
  - Transfer secret
  - Refresh nonce
- Nonce check
- Two commit hashes instead of one

```
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, c'_s || R_0 || R_1, "b-seed")
α_0 := HKDF(256, b_s, "a0")
α_1 := HKDF(256, b_s, "a1")
β_0 := HKDF(256, b_s, "b0")
β_1 := HKDF(256, b_s, "b1")
R'_0 = R_0 + α_0 G + β_0 D_p
R'_1 = R_1 + α_1 G + β_1 D_p
c'_0 = H(R'_0, C'_p)
c'_1 = H(R'_1, C'_p)
\overline{c}_0 = c'_0 + β_0 \mod p
\overline{c}_1 = c'_1 + β_1 \mod p
return (T, c'_s, C'_p, \overline{c}_0, \overline{c}_1)
```
Tipping

- Wallet: same changes as Withdraw
- Merchant: Only message signed by merchant’s reserve private key changes
Payback Protocol

- Three different cases:
  - Revoked coin has never been seen by exchange: Adjust Withdraw Transcript
  - Coin partially spent: Invoke Refresh Protocol
  - Coin resulted from refresh, has never been seen: Adjust refresh transcript
Overview

Implemented & Tested:
- Cryptographic routines in GNUnet
- Cryptographic utilities in the Exchange
- Security Module for CS and crypto-helper
- Key Management
- New Endpoint to get $R_0, R_1$
- Withdraw protocol
- Deposit protocol

Not Implemented:
- Merchant (primarily Spend Protocol)
- Wallet support for two denomination types
- Tipping protocol
Testing

- Specification and test implementation hand in hand
- Cryptographic routines: unit tests, benchmark, test vectors
- Taler cryptographic utilities: unit tests
- CS security module: functionality tests, benchmark
- Exchange HTTP server: functionality tests (simulate wallet)
Implementation of cryptographic routines

Cryptographic routines in GNUnet

Cryptographic routines for Clause Blind Schnorr signatures:

- Programming language: C
- Implemented as free software in the GNUnet project
- Implemented on Curve25519
- Libsodium is used for group operations
- Implemented including testing, benchmarks and test-vector generator
- Other primitives from GNUnet reused
  - HKDF
  - KDF mod
  - Hash functions

graphics source: https://www.gnunet.org/images/gnunet-logo-dark-no-text.png
### Implementation of cryptographic routines

#### Implementation details

<table>
<thead>
<tr>
<th>Operation</th>
<th>API</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key Generation</td>
<td>GNUNET_CRYPTO_cs_private_key_generate()</td>
</tr>
<tr>
<td>Get public key</td>
<td>GNUNET_CRYPTO_cs_private_key_get_public()</td>
</tr>
<tr>
<td>Derive $r_0$, $r_1$</td>
<td>GNUNET_CRYPTO_cs_derive_r(nonce, lts, $r[2]$)</td>
</tr>
<tr>
<td>Get public $R$</td>
<td>GNUNET_CRYPTO_cs_r_get_public()</td>
</tr>
<tr>
<td>Derive blinding secrets (bs)</td>
<td>GNUNET_CRYPTO_cs_blinding_secrets_derive(seed)</td>
</tr>
<tr>
<td>Calculate blinded $c$</td>
<td>GNUNET_CRYPTO_cs_calc_blinded_c(bs, $R[2]$, $pk$, msg)</td>
</tr>
<tr>
<td>Unblind</td>
<td>GNUNET_CRYPTO_cs_unblind(blind_sig, $pk$, msg)</td>
</tr>
<tr>
<td>Verify</td>
<td>GNUNET_CRYPTO_cs_verify(sig, $pk$, msg)</td>
</tr>
</tbody>
</table>

- API designed to prevent misuse
- API includes “Clause” part
- Internal functionality: CS-FDH, clamping

<table>
<thead>
<tr>
<th>Values</th>
<th>Data Structure</th>
</tr>
</thead>
<tbody>
<tr>
<td>Curve25519 Scalar</td>
<td>GNUNET_CRYPTO_Cs25519Scalar</td>
</tr>
<tr>
<td>Curve25519 Point</td>
<td>GNUNET_CRYPTO_Cs25519Point</td>
</tr>
<tr>
<td>Private Key</td>
<td>GNUNET_CRYPTO_CsPrivateKey</td>
</tr>
<tr>
<td>Public Key</td>
<td>GNUNET_CRYPTO_CsPublicKey</td>
</tr>
<tr>
<td>$\alpha, \beta$</td>
<td>GNUNET_CRYPTO_CsBlindingSecret</td>
</tr>
<tr>
<td>$r$</td>
<td>GNUNET_CRYPTO_CsRSecret</td>
</tr>
<tr>
<td>$R$</td>
<td>GNUNET_CRYPTO_CsRPublic</td>
</tr>
<tr>
<td>$c$</td>
<td>GNUNET_CRYPTO_CsC</td>
</tr>
<tr>
<td>$s$</td>
<td>GNUNET_CRYPTO_CsBlindS</td>
</tr>
<tr>
<td>$s'$</td>
<td>GNUNET_CRYPTO_CsS</td>
</tr>
<tr>
<td>$\sigma := \langle s', R' \rangle$</td>
<td>GNUNET_CRYPTO_CsSignature</td>
</tr>
<tr>
<td>Nonce</td>
<td>GNUNET_CRYPTO_CsNonce</td>
</tr>
</tbody>
</table>
Adding Schnorr's blind signature in Taler - Defence

Exchange Architecture

secmod-eddsa <-> httpd <-> secmod-rsa

secmod-cs <-> Postgres <-> aggregator

transfer <-> Nexus <-> wirewatch

graphics source: https://git.taler.net/marketing.git/plain/presentations/comprehensive/main.pdf
Adding Schnorr’s blind signature in Taler - Defence

Berner Fachhochschule

Flag-checkered
Goals & Project Management
Preliminaries
Protocol Redesign
Specification & Implementation
Results

v1.0

Taler cryptographic utilities
Cryptographic utilities around crypto routines and planchets

Cryptographic utilities to use the crypto routines
- sign
- blind
- unblind
- key generation
- derive_r
- various utility functions

Utility functions around planchets
- derive/generate nonce
- blinding secrets
- planchet setup & prepare
- planchet to coin
- coin ev hash
CS Security Module:
- Standalone process
- The CS Security Module have sole access to the denomination private key
- All operations requiring the private key are done by the security module
  - Generate new keypair
  - Sign a message
  - Revoke keys
  - Derive private \( r \)
- API can use fixed-length structs (compared to RSA)

CS Crypto Helper:
- Talks to the security module for operations requiring the denominations private key
- Is part of the httpd service
- Unix Domain Sockets are used for Inter-Process Communication with the security module
Key Management

- Collect new denominations, security module public key from CS security module
- **GET /management/keys**: Offer future keys to exchange-offline
- **POST /management/keys**: Return signatures created with offline-signing key
- **GET /keys**: Make new denominations available for wallet:
- Currently requires both RSA and CS security modules to be running
Adding Schnorr’s blind signature in Taler - Defence

Endpoint for $R$

- New endpoint used for withdraw and refresh protocols
- Available under POST /csr
- Request:

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>nonce</td>
<td>String</td>
<td>32 Bytes encoded in Crockford base32 Hex</td>
</tr>
<tr>
<td>denom_pub_hash</td>
<td>String</td>
<td>Denomination Public Key encoded in Crockford base32 Hex</td>
</tr>
</tbody>
</table>

- Exchange checks denomination (including cipher type)
Endpoint for $R$

- Exchange derives $R$ based on supplied nonce and denomination
- Request passed down to security module
- No persistence necessary
- Response:

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>r_pub_0</td>
<td>String</td>
<td>32 Bytes encoded in Crockford base32 Hex</td>
</tr>
<tr>
<td>r_pub_1</td>
<td>String</td>
<td>32 Bytes encoded in Crockford base32 Hex</td>
</tr>
</tbody>
</table>
Withdraw Protocol

- Available under POST
  /reserves/[reserve]/withdraw

- Request data:

<table>
<thead>
<tr>
<th>Field</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>denom_pub_hash</td>
<td>Denomination Public Key</td>
</tr>
<tr>
<td>coin_ev</td>
<td>RSA blinded coin public key</td>
</tr>
<tr>
<td>reserve_sig</td>
<td>Signature over the request using the reserve's private key</td>
</tr>
</tbody>
</table>

- Adjusted coin_ev field (RSA):

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>cipher</td>
<td>Integer</td>
<td>Denomination cipher: 1 stands for RSA</td>
</tr>
<tr>
<td>rsa_blinded_planchet</td>
<td>String</td>
<td>RSA blinded coin public key</td>
</tr>
</tbody>
</table>

- CS coin_ev field:

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>cipher</td>
<td>Integer</td>
<td>Denomination cipher: 2 stands for CS</td>
</tr>
<tr>
<td>cs_nonce</td>
<td>String</td>
<td>32 Bytes encoded in Crockford base32 Hex</td>
</tr>
<tr>
<td>cs_blinded_c0</td>
<td>String</td>
<td>32 Bytes encoded in Crockford base32 Hex</td>
</tr>
<tr>
<td>cs_blinded_c1</td>
<td>String</td>
<td>32 Bytes encoded in Crockford base32 Hex</td>
</tr>
</tbody>
</table>

- Response:

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>cipher</td>
<td>Integer</td>
<td>Denomination cipher: 2 stands for CS</td>
</tr>
<tr>
<td>b</td>
<td>Integer</td>
<td>CS signature session identifier (either 0 or 1)</td>
</tr>
<tr>
<td>s</td>
<td>String</td>
<td>signature scalar (32 Bytes encoded in Crockford base32 Hex)</td>
</tr>
</tbody>
</table>
Withdraw Protocol

Implementation details

- Idempotency check - has the coin already been withdrawn?
  - RSA: Hash over message (blinded coin)
  - CS: Hash over nonce and denomination public key
- Additional denomination cipher check
- Various changes related to parsing, persistence and response
Minor Security Fix

- Recap: RSA idempotency check uses blinded coin hash
- Issue:
  - Wallet withdraws a coin
  - Withdraw same coin referencing different denomination
  - Exchange returns signature of first withdraw due to idempotency check
  - Invalid signature - open complaint at auditor
  - Auditor is able to disprove
- Solution: add denomination to coin hash
Deposit Protocol

- Available under POST /coins/[coin public key]/deposit
- Request: many fields, only coin_sig relevant for CS
- Content (RSA):

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>cipher</td>
<td>Integer</td>
<td>Denomination cipher: 1 stands for RSA</td>
</tr>
<tr>
<td>rsa_signature</td>
<td>String</td>
<td>Unblinded RSA signature</td>
</tr>
</tbody>
</table>

- coin_sig content for CS:

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>cipher</td>
<td>Integer</td>
<td>Denomination cipher: 2 stands for CS</td>
</tr>
<tr>
<td>cs_signature_r</td>
<td>String</td>
<td>Curve point $R'$ (32 Bytes encoded in Crockford base32 Hex)</td>
</tr>
<tr>
<td>cs_signature_s</td>
<td>String</td>
<td>Signature scalar (32 Bytes encoded in Crockford base32 Hex)</td>
</tr>
</tbody>
</table>

- Add denomination cipher check
- Signature verification (CS security module)
- Adjusted persistence
**New:** Wallet Cryptographic Routines

Wallet Implementation

- Programming language: Typescript
- libsodium.js for group operations
- cryptographic routines implemented
- tested with test vectors from C implementation

**Missing:**
- Add support for two denomination types (together with Taler team)
- integration test with exchange

*graphics source: https://taler.net/images/stock1s.jpg*
Results
Security Assumptions

RSA Blind Signature’s & Clause Blind Schnorr Signature’s

Scheme comparison:
- Number of blinding secrets
- Number of round trips
- CS signatures do most computations twice

Security assumptions
- Both Schemes are considered perfectly blind
- RSA depends on factoring large numbers being hard.
- Schnorr Signatures depends on computing the discrete logarithm being hard
- Clause Blind Schnorr Signatures additionally rely on the modified ROS problem being hard
- ROS is a recent research topic, and not as well researched
## CPU Performance

### Setup

<table>
<thead>
<tr>
<th></th>
<th>CPU: 8-core AMD Ryzen 7 PRO 5850U</th>
<th>OS: Ubuntu 21.10 Linux 5.13.0-25-generic</th>
</tr>
</thead>
</table>

### Operation | CS       | RSA 1024 bit | RSA 3072 bit |
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>key generation</td>
<td>0.204 ms</td>
<td>126 ms</td>
<td>2684 ms</td>
</tr>
<tr>
<td>blind</td>
<td>3.870 ms</td>
<td>1.282 ms</td>
<td>5 ms</td>
</tr>
<tr>
<td>signing</td>
<td>0.077 ms</td>
<td>7 ms</td>
<td>86 ms</td>
</tr>
<tr>
<td>unblinding</td>
<td>0.001 ms</td>
<td>2.991 ms</td>
<td>24 ms</td>
</tr>
<tr>
<td>verifying</td>
<td>1.358 ms</td>
<td>0.876 ms</td>
<td>3.075 ms</td>
</tr>
</tbody>
</table>
## Disk Space & Bandwidth

### Signatures: $\langle s, R \rangle$

<table>
<thead>
<tr>
<th>Signature Scheme</th>
<th>Disk Space</th>
<th>Factor</th>
<th>Disk Space 1M signatures</th>
</tr>
</thead>
<tbody>
<tr>
<td>CS 512 bits</td>
<td>1x</td>
<td>64 MB</td>
<td></td>
</tr>
<tr>
<td>RSA 1024 bit</td>
<td>2x</td>
<td>128 MB</td>
<td></td>
</tr>
<tr>
<td>RSA 2048 bit</td>
<td>4x</td>
<td>256 MB</td>
<td></td>
</tr>
<tr>
<td>RSA 3072 bit</td>
<td>6x</td>
<td>384 MB</td>
<td></td>
</tr>
<tr>
<td>RSA 4096 bit</td>
<td>8x</td>
<td>512 MB</td>
<td></td>
</tr>
</tbody>
</table>

### Wallet disk space: $\langle c_s, s, R_0, R_1, D_p \rangle$

<table>
<thead>
<tr>
<th>Signature Scheme</th>
<th>Disk Space</th>
<th>Factor</th>
<th>Disk Space 1M coins</th>
</tr>
</thead>
<tbody>
<tr>
<td>CS 256 bits</td>
<td>150 bytes</td>
<td>1x</td>
<td>160 MB</td>
</tr>
<tr>
<td>RSA 1024 bit</td>
<td>416 bytes</td>
<td>2.6x</td>
<td>416 MB</td>
</tr>
<tr>
<td>RSA 2048 bit</td>
<td>800 bits</td>
<td>5x</td>
<td>800 MB</td>
</tr>
<tr>
<td>RSA 3072 bit</td>
<td>1184 bits</td>
<td>7.4x</td>
<td>1184 MB</td>
</tr>
<tr>
<td>RSA 4096 bit</td>
<td>1568 bits</td>
<td>9.8x</td>
<td>1568 MB</td>
</tr>
</tbody>
</table>
Latency

- CS introduces an additional round trip
- A coin should not be spent immediately after withdrawal or refresh
- Additional round trip is therefore *negligible*
Comparison Conclusion

- ⚡ CS has overall better performance regarding speed, disk space and bandwidth
- 🔗 Additional round-trip is negligible
- 📊 CS has an additional, newer security assumption called ROS
- ⚡ Risk can be calculated and capped by denomination key lifetime
We would also like to thank Mr. Benoist and Mr. Voisard for the guidance during our thesis.
Future Work

- Refresh and other protocols (tipping, deposit, refund, etc.)
- Wallet
- Merchant
- Security Audit
- CS implementation on other curves
- Exchange API documentation
- Exchange operator guideline for when to use CS
Personal Conclusion

- From high-level down to code
- Challenging at times, pushed through with persistence
- Motivation grew with every completed step
- C:
  - Respect from it, but went well (cough macros cough)
  - Well designed APIs
  - Integrate new variables without RSA-counterpart
- Hope to pay with own code in the future!