Berner
Fachhochschule

Adding Schnorr’s blind signature in Taler - Defence

Gian Demarmels Lucien Heuzeveldt

Advisor: Prof. Dr. Emmanuel Benoist

Expert: Elektronikingenieur HTL Daniel Voisard

Content

- 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

Goals

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

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.

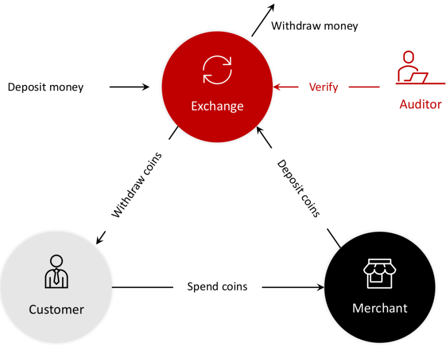
Project Phases:

- Phase 1: Initiation
- Phase 2: Planning
- Phase 3: Execution
  - a) Design Phase
  - b) Specification Phase
  - c) Implementation Phase
- Phase 4: Discussion
- Phase 5: Closure

Preliminaries

GNU Taler Overview

A privacy-preserving, fast and intuitive payment system



graphics source: `https://taler.net/images/diagram-simple.png`

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

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:

- 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”)

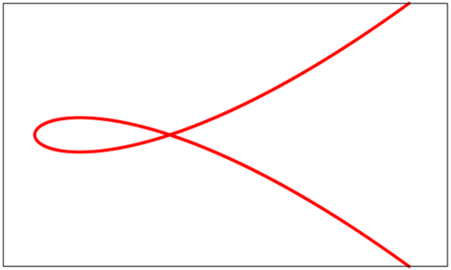
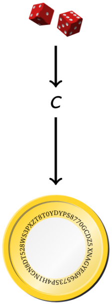


Abbildung: Abbild der elliptischen Kurve $y2 = x3 + 486662x2 + x$

graphics source: `https://heise.cloudimg.io/v7/_www-heise-de_/imgs/18/1/4/5/9/6/8/9/curve25519-5b8d94dd2448661c.png`

EdDSA

- 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

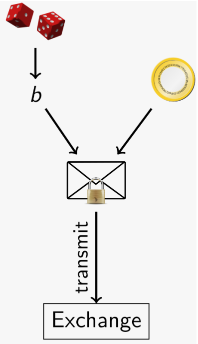


graphics source: `https://git.taler.net/marketing.git/plain/presentations/comprehensive/main.pdf`

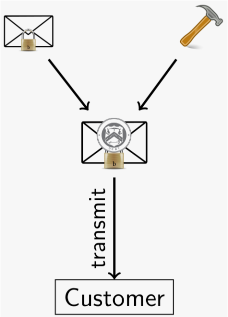
Blind Signatures

RSA Blind Signatures in Taler

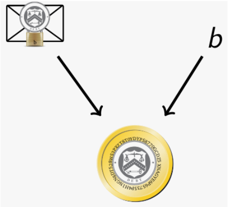
Customer:



Exchange:



Customer:



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

RSA Blind Signatures

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

$$f = FDH(m)$$

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

$$\xrightarrow{f'}$$

Bob
knows:
RSA keys d_B, D_B

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

$$\xleftarrow{s'}$$

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

Schnorr Signature Scheme

Berner Fachhochschule

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

v1.0

The (broken) Blind Schnorr Signature Scheme

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 $sG = R + cX$
$s' := s + \alpha \mod p$
$\sigma := \langle R', s' \rangle$

ROS problem - (informally)

Random inhomogeneities in an Overdetermined, Solvable system of linear equations

ROS problem:

- ROS depends on group order p, parameterized with integer ℓ
- An adversary can produce $\ell + 1$ valid signatures after $\ell > \log_2(p)$ parallel sessions by solving a linear equation system
- $\sum_{j=1}^{\ell} \rho_{i,j} c_j = H_{ros}(\overrightarrow{p}_i), i \in [\ell + 1]$
- There exist a polynomial-time attack against ROS_ℓ 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: Blind Schnorr Signatures and Signed ElGamal Encryption in the Algebraic Group Model (`https://eprint.iacr.org/2019/877.pdf`)

See: On the (in)security of ROS (`https://eprint.iacr.org/2020/945`)

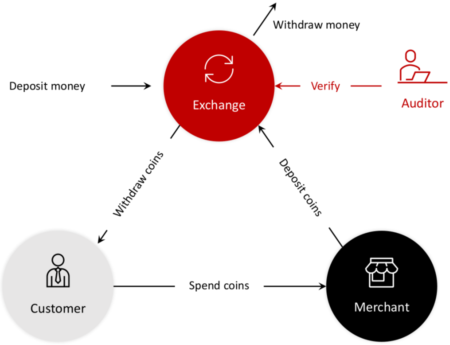
Clause Blind Schnorr Signature Scheme

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

Taler Protocols

Protocols:

- Withdrawal
- Refresh
- Spend
- Deposit
- Tipping
- Payback
- Recoup



graphics source: `https://taler.net/images/diagram-simple.png`

Withdrawal Protocol

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

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:
$\rho_W = D_p, m'$
$\sigma_W = \text{Ed25519.Sign}(w_s, \rho_W)$

Exchange
reserve public key W_p
denomination keys d_s, D_p

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

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

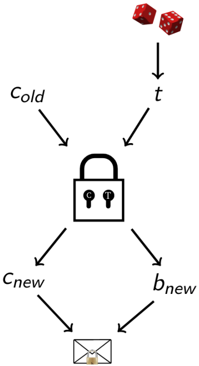
$\xleftarrow{\sigma_c'}$

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

Refresh Protocol - DH-Lock

Diffie-Hellman Lock:

- keypairs $C = cG$ and $T = tG$
- Both keys can unlock the lock: $k = tC = cT$



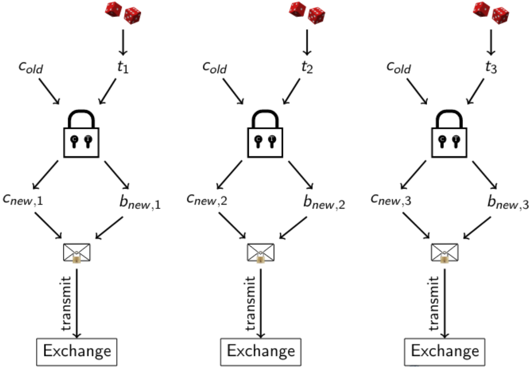
RefreshDerive$(s, \langle e, N \rangle, C_p)$

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

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_γ
- Exchange can detect fraud attempts with a probability of $1/k$



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

Refresh Protocol Commit Phase

- Customer creates k RefreshDerives (DH-Locks)
- Customer commits by calculating a commit hash
  $$h_T := H(T_1, \ldots, T_k)$$
  $$h_{\overline{m}} := H(\overline{m}_1, \ldots, \overline{m}_k)$$
  $$h_C := H(h_T, h_{\overline{m}})$$
- The exchange answers with a random $\gamma \in \{1, \ldots, k\}$

Customer

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

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

Exchange

denomination keys $d_{s(i)}, D_{p(i)}$
$(h_C, D_{p(t)}, D_{p(0)}, C_p^{(0)}, \sigma_C^{(0)} = \rho_{RC})$
checkEd25519.Verify$(C_p^{(0)}, \sigma_{RC}, \rho_{RC})$
$x \rightarrow$ GetOldRefresh(ρ_{RC})
Comment: GetOldRefresh
$(\rho_{RC} \mapsto \{\perp, \gamma\})$
if $x = \perp$
$v := D(D_{p(t)})$
$\langle e_0, N_0 \rangle := D_{p(0)}$
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$(C_p^{(0)}, v)$
$\gamma \leftarrow \{1, \ldots, \kappa\}$
Persist refresh-record $\langle \rho_{RC}, \gamma \rangle$
else
$\gamma := x$
endif

$\xleftarrow{\gamma}$

Refresh Protocol Reveal Phase

- 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 $1/k$

$\xleftarrow{\gamma}$

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$\langle D_{p(t)}, c_s^{(\gamma)}, C_p^{(\gamma)}, \sigma_C^{(\gamma)} \rangle$

Link Protocol

- 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 $c_{s(old)}$

Customer		Exchange
knows: coin$_0 = \langle D_{p(0)}, c_s^{(0)}, C_p^{(0)}, \sigma_C^{(0)} \rangle$		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$(c_s^{(0)}, T_\gamma^{(i)})$ $r_i :=$ SelectSeeded$(x_i, \mathbb{Z}^*_{N_t})$ $c_s^{(i)} :=$ HKDF$(256, x_i, "c")$ $C_p^{(i)} :=$ 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$\langle D_{p(t)}^{(i)}, c_s^{(i)}, C_p^{(i)}, \sigma_C^{(i)} \rangle$		

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

CS R

- Additional Request during signature creation
- Introduces complexity
- Challenge regarding abort-idempotency
- Vanilla Clause Blind Schnorr Signature Scheme:
  - $r_0 \leftarrow random$
  - $R_0 := rG$
- Our Changes:
  - Introduces Nonce n used for Derivation
  - Derives R:
    $r_0 := \text{HKDF}(256, n||d_s, \text{"r0"})$
    $R_0 := r_0 G$
  - 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 $\rightarrow$ unpredictable

Withdraw Protocol

Protocol Changes

- Withdraw Nonce (Wallet):
  $c_s, C_p \leftarrow \text{Ed25519.KeyGen}()$
  $n_w := \text{HKDF}(256, c_s, \text{"n"})$
- Request R
- Derive R (Exchange)
- Derive Blinding Secrets (Wallet):
  $b_s := \text{HKDF}(256, c_s || R_0 || R_1, \text{"b-seed"})$
  $\alpha_0 := \text{HKDF}(256, b_s, \text{"a0"})$
  $\dots$
  $\beta_1 := \text{HKDF}(256, b_s, \text{"b1"})$

Customer		Exchange
knows: reserve keys w_s, W_p denomination public key D_p		knows: reserve public key W_p denomination keys d_s, D_p
generate coin key pair: $c_s, C_p \leftarrow \text{Ed25519.KeyGen}()$ $n_w := \text{HKDF}(256, c_s, \text{"n"})$		
	$\xrightarrow{n_w, D_p}$	
		verify if D_p is valid $r_0 := \text{HKDF}(256, n_w \| d_s, \text{"r0"})$ $r_1 := \text{HKDF}(256, n_w \| d_s, \text{"r1"})$ $R_0 := r_0 G$ $R_1 := r_1 G$
	$\xleftarrow{R_0, R_1}$	
persist $\langle c_s, C_p, R_0, R_1 \rangle$ blind: $b_s := \text{HKDF}(256, c_s \| R_0 \| R_1, \text{"b-seed"})$ $\alpha_0 := \text{HKDF}(256, b_s, \text{"a0"})$ $\alpha_1 := \text{HKDF}(256, b_s, \text{"a1"})$ $\beta_0 := \text{HKDF}(256, b_s, \text{"b0"})$ $\beta_1 := \text{HKDF}(256, b_s, \text{"b1"})$ $R'_0 := R_0 + \alpha_0 G + \beta_0 D_p$ $R'_1 := R_1 + \alpha_1 G + \beta_1 D_p$ $c'_0 := H(R'_0, C_p)$ $c'_1 := H(R'_1, C_p)$ $c_0 := c'_0 + \beta_0 \mod p$ $c_1 := c'_1 + \beta_1 \mod p$		

Withdraw Protocol

Protocol Changes

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

Customer		Exchange
knows: reserve keys w_s, W_p; denomination public key D_p		knows: reserve public key W_p; denomination keys d_s, D_p
	Continuation of figure 4.1	
sign with reserve private key: $\rho_W := \langle n_w, D_p, c_0, c_1 \rangle$; $\sigma_W := \text{Ed25519.Sign}(w_s, \rho_W)$		
	$\xrightarrow{W_p, \sigma_W, \rho_W}$	
		$\langle n_w, D_p, c_0, c_1 \rangle := \rho_W$; verify if D_p is valid; check Ed25519.Verify(W_p, ρ_W, σ_W); $b := \text{HKDF}(1, n_w \| d_s, \text{"b"})$; $s \leftarrow \text{GetWithdraw}(n_w, D_p)$; **if** $s = \perp$; $r_b := \text{HKDF}(256, n_w \| d_s, \text{"rb"})$; $s := r_b + c_b d_s \mod p$; decrease balance if sufficient and persist $\langle n_w, D_p, s \rangle$; **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		

Withdraw Protocol

Nonce Check

- Is this safe? (without nonce reuse check)
  $r_0 := \text{HKDF}(256, n||d_s, \text{"r0"})$
- (Hint $\rightarrow$ no):
  - $s_2 - s_1 = d_s(c'_1 - c'_2) - (r_1 - r_2)$
  - if $r_1 = r_2$:
    $s_2 - s_1 = d_s(c'_1 - c'_2)$
  - Allows private key recovery
  - Happened before (Bitcoin, PlayStation 3)
- Prevent r reuse $\rightarrow$ do not allow nonce reuse (per denomination)
- Applies to withdraw AND refresh

Deposit Protocol

- Only coin signature verification changes:

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

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

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

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

</> Specification & Implementation

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

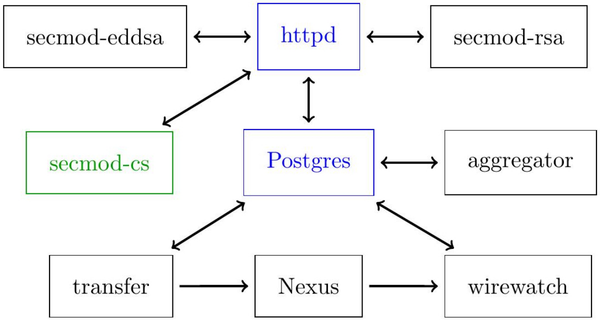
Implementation details

Operation	API
Key Generation	GNUNET_CRYPTO_cs_private_key_generate()
Get public key	GNUNET_CRYPTO_cs_private_key_get_public(sk)
Derive r_0, r_1	GNUNET_CRYPTO_cs_derive_r(nonce, lts, r[2])
Get public R	GNUNET_CRYPTO_cs_r_get_public(r)
Derive blinding secrets (bs)	GNUNET_CRYPTO_cs_blinding_secrets_derive(seed)
Calculate blinded c	GNUNET_CRYPTO_cs_calc_blinded_c(bs, R[2], pk, msg)
Sign and get b	GNUNET_CRYPTO_cs_sign_derive(sk, r[2], c[2], nonce)
Unblind	GNUNET_CRYPTO_cs_unblind(blind_sig, pk, msg)
Verify	GNUNET_CRYPTO_cs_verify(sig, pk, msg)

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

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

Exchange Architecture



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

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

CS Security Module & corresponding crypto helper

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 secuity 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

Berner Fachhochschule

v1.0

Endpoint for R

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

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

- 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:

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

Withdraw Protocol

- Available under `POST /reserves/[reserve]/withdraw`
- Request data:

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

- Adjusted coin_ev field (RSA):

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

- CS coin_ev field:

Field	Type	Value
cipher	Integer	Denomination cipher: 2 stands for CS
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

- Response:

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

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):

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

- coin_sig content for CS:

Field	Type	Value
cipher	Integer	Denomination cipher: 2 stands for CS
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)

- 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

CPU: 8-core AMD Ryzen 7 PRO 5850U
OS: Ubuntu 21.10 Linux 5.13.0-25-generic

Operation	CS	RSA 1024 bit	RSA 3072 bit
10x key generation	0.204 ms	126 ms	2684 ms
10x blind	3.870 ms	1.282 ms	5 ms
10x signing	0.077 ms	7 ms	86 ms
10x unblinding	0.001 ms	2.991 ms	24 ms
10x verifying	1.358 ms	0.876 ms	3.075 ms

Disk Space & Bandwidth

Signatures: $\langle s, R \rangle$

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

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

Signature Scheme	Disk Space	Factor	Disk Space 1M coins
CS 256 bits	150 bytes	1x	160 MB
RSA 1024 bit	416 bytes	2.6x	416 MB
RSA 2048 bit	800 bits	5x	800 MB
RSA 3072 bit	1184 bits	7.4x	1184 MB
RSA 4096 bit	1568 bits	9.8x	1568 MB

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

Acknowledgement

- Christian Grothoff
- Jeffrey Burdges
- Jacob Appelbaum
- Florian Dold

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!

Let's talk!


Java Is Just Coffee
Menu:
Espresso 5.20.–
Cafe Late........ 6.20.–
Cappuccino........ 6.20.–
Hot Chocolate........ 5.20.–
Hot Tea 4.20.–
Pay with:
TALER
RSA