

SplitKey: Two-party signing and decryption with extra features

MPTS 2026: NIST Workshop on Multi-Party Threshold Schemes 2026

Peeter Laud (Cybernetica AS, team SplitForge)

2002: Identity cards

- Create signatures, protecting the private keys
- User's computer must have a card reader
- There are more-or-less standard software components for browsers and other applications to access the card
- To activate the card, it must receive a PIN from the software component or the card reader
 - User enters that PIN



2007: Mobile-ID

- A SIM card with additional functionality
 - Creates signatures, protecting the private keys



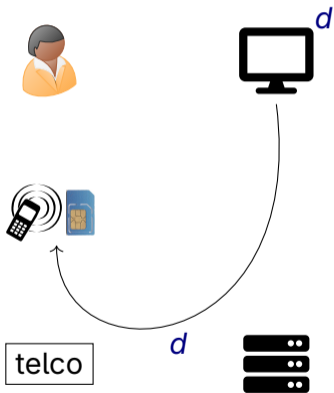
2007: Mobile-ID

- A SIM card with additional functionality
 - Creates signatures, protecting the private keys



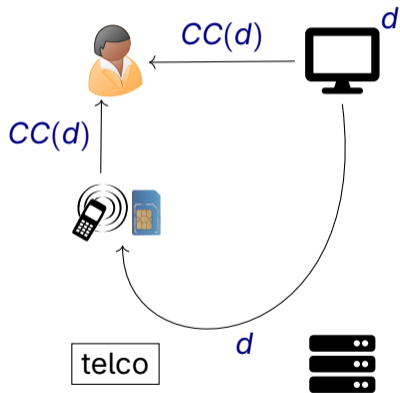
2007: Mobile-ID

- A SIM card with additional functionality
 - Creates signatures, protecting the private keys
 - Receives the to-be-signed value via OTA (Over-the-Air) and SMSC servers, and service SMS communication channel



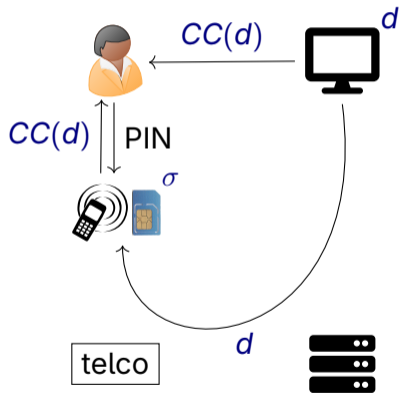
2007: Mobile-ID

- A SIM card with additional functionality
 - Creates signatures, protecting the private keys
 - Receives the to-be-signed value via OTA (Over-the-Air) and SMSC servers, and service SMS communication channel



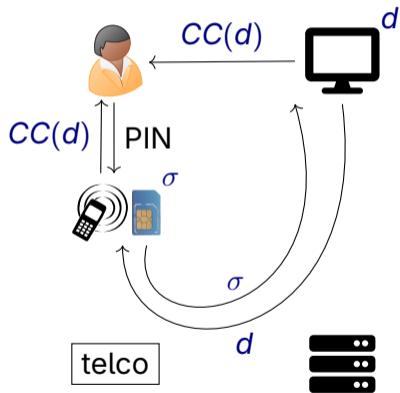
2007: Mobile-ID

- A SIM card with additional functionality
 - Creates signatures, protecting the private keys
 - Receives the to-be-signed value via OTA (Over-the-Air) and SMSC servers, and service SMS communication channel
 - Is activated by a PIN entered to phone



2007: Mobile-ID

- A SIM card with additional functionality
 - Creates signatures, protecting the private keys
 - Receives the to-be-signed value via OTA (Over-the-Air) and SMSC servers, and service SMS communication channel
 - Is activated by a PIN entered to phone



Certification

- eIDAS (EU 910/2014) defines QES and Q(E)SCD
 - A *Qualified Electronic Signature (QES)* has the same legal meaning as a handwritten signature
 - A *Qualified Signature Creation Device (QSCD)* can be used to create QES-s
- Implementing act EU 650/2016 establishes the use of Common Criteria for certification, and protection profiles for QSCD-s
- ID-cards (IDEMIA Cosmo, Thales MultiApp) have been certified as QSCD-s
- Mobile-ID SIMs were certified as a QSCD
 - Currently: SSCD

2014: expected takeover of embedded SIM-s

Smartphones: an emerging platform

- Put the private key into the phone?
 - In 2014, this did not seem to be a good idea

Smartphones: an emerging platform

- Put the private key into the phone?
 - In 2014, this did not seem to be a good idea
- An adversarial app can perhaps read the persistent memory of another app
- An adversarial app can perhaps also read the volatile memory of another app that is currently running

Smartphones: an emerging platform

- Put the private key into the phone?
 - In 2014, this did not seem to be a good idea
- An adversarial app can perhaps read the persistent memory of another app
- An adversarial app can perhaps also read the volatile memory of another app that is currently running
- Threshold signing, 2 - out - of - 2. Phone knows, what to sign. How does assisting server learn it?
 - Phone tells the server, what to sign...
 - Server tries to detect adversarial activities...

2016: SplitKey (key generation)



2016: SplitKey (key generation)

Client



$$p_1, q_1 \leftarrow \$ \mathbb{P}$$

$$n_1 \leftarrow p_1 q_1$$

$$d_1 \leftarrow e^{-1} \pmod{\phi(n_1)}$$

Server

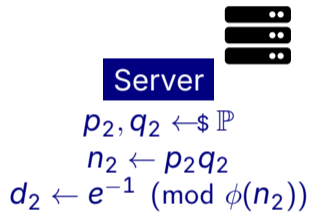
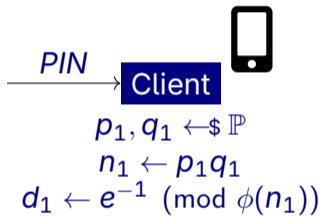


$$p_2, q_2 \leftarrow \$ \mathbb{P}$$


$$n_2 \leftarrow p_2 q_2$$

$$d_2 \leftarrow e^{-1} \pmod{\phi(n_2)}$$


2016: SplitKey (key generation)



2016: SplitKey (key generation)

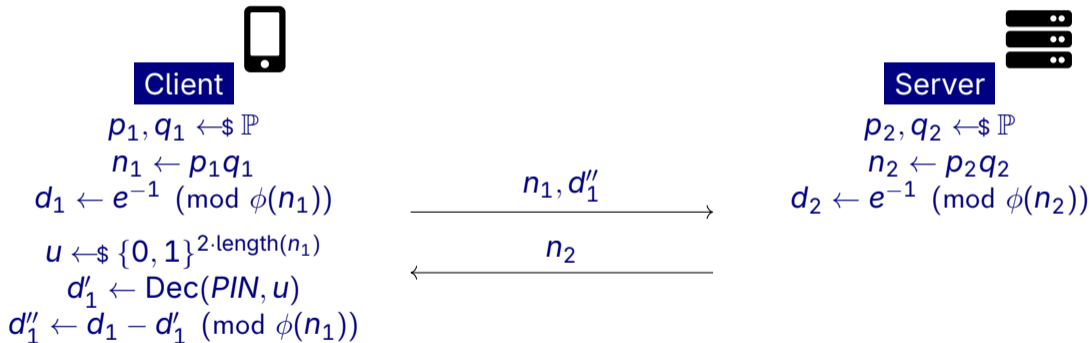
PIN → **Client** 

$$p_1, q_1 \leftarrow \$ \mathbb{P}$$
$$n_1 \leftarrow p_1 q_1$$
$$d_1 \leftarrow e^{-1} \pmod{\phi(n_1)}$$
$$u \leftarrow \$ \{0, 1\}^{2 \cdot \text{length}(n_1)}$$
$$d'_1 \leftarrow \text{Dec}(PIN, u)$$
$$d''_1 \leftarrow d_1 - d'_1 \pmod{\phi(n_1)}$$

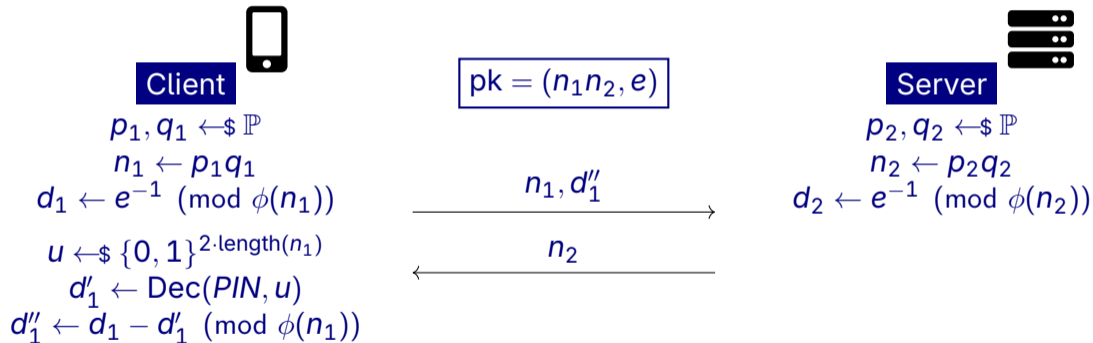
Server 

$$p_2, q_2 \leftarrow \$ \mathbb{P}$$
$$n_2 \leftarrow p_2 q_2$$
$$d_2 \leftarrow e^{-1} \pmod{\phi(n_2)}$$

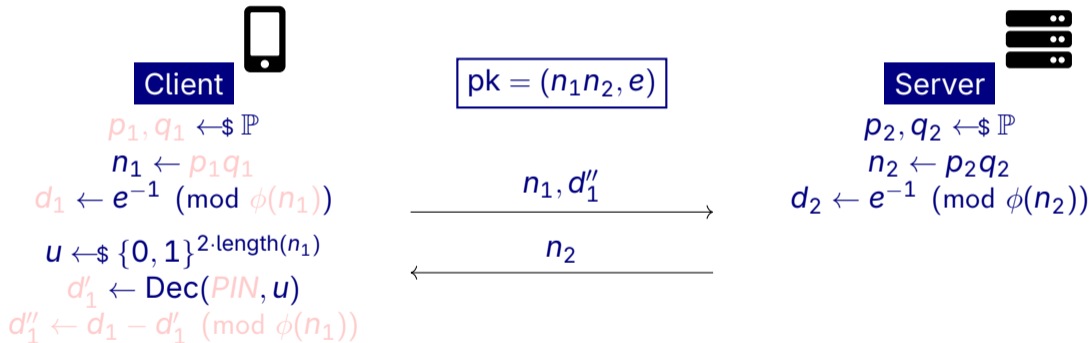
2016: SplitKey (key generation)



2016: SplitKey (key generation)



2016: SplitKey (key generation)



2016: SplitKey (signing)

$$\begin{aligned} \text{pk} &= (n_1 n_2, e) \quad n_i = p_i q_i \quad d_i \cdot e \equiv 1 \pmod{\varphi(n_i)} \\ d'_1 + d''_1 &\equiv d_1 \pmod{\varphi(n_1)} \quad u = \text{Enc}(\text{PIN}, d'_1) \end{aligned}$$

Client

u, n_1, n_2

Server

d''_1, d_2, n_1, n_2

2016: SplitKey (signing)

$$\begin{aligned} \text{pk} &= (n_1 n_2, e) \quad n_i = p_i q_i \quad d_i \cdot e \equiv 1 \pmod{\varphi(n_i)} \\ d'_1 + d''_1 &\equiv d_1 \pmod{\varphi(n_1)} \quad u = \text{Enc}(\text{PIN}, d'_1) \end{aligned}$$



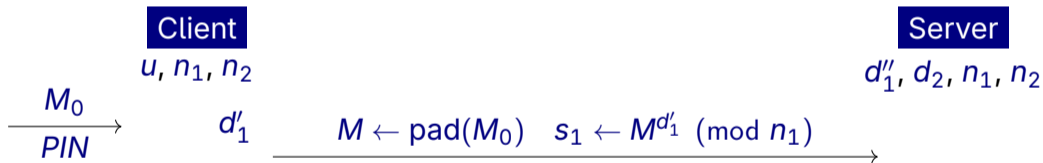
2016: SplitKey (signing)

$$\begin{aligned} \text{pk} &= (n_1 n_2, e) \quad n_i = p_i q_i \quad d_i \cdot e \equiv 1 \pmod{\varphi(n_i)} \\ d'_1 + d''_1 &\equiv d_1 \pmod{\varphi(n_1)} \quad u = \text{Enc}(\text{PIN}, d'_1) \end{aligned}$$



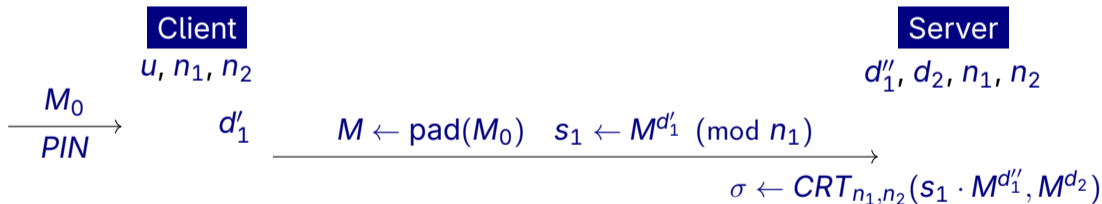
2016: SplitKey (signing)

$$\begin{aligned} \text{pk} &= (n_1 n_2, e) \quad n_i = p_i q_i \quad d_i \cdot e \equiv 1 \pmod{\varphi(n_i)} \\ d'_1 + d''_1 &\equiv d_1 \pmod{\varphi(n_1)} \quad u = \text{Enc}(\text{PIN}, d'_1) \end{aligned}$$



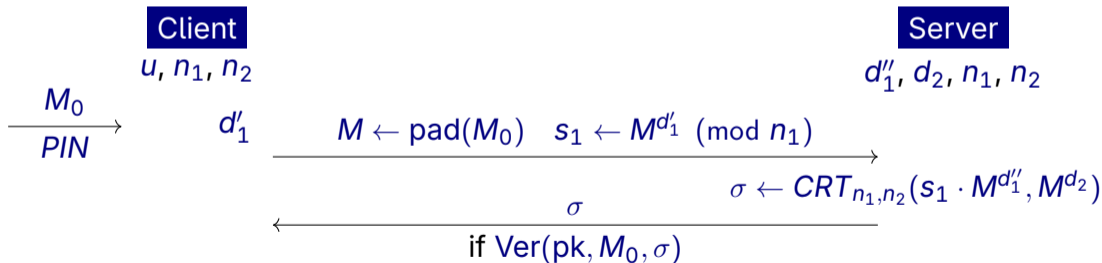
2016: SplitKey (signing)

$$\begin{aligned} \text{pk} &= (n_1 n_2, e) \quad n_i = p_i q_i \quad d_i \cdot e \equiv 1 \pmod{\varphi(n_i)} \\ d'_1 + d''_1 &\equiv d_1 \pmod{\varphi(n_1)} \quad u = \text{Enc}(\text{PIN}, d'_1) \end{aligned}$$



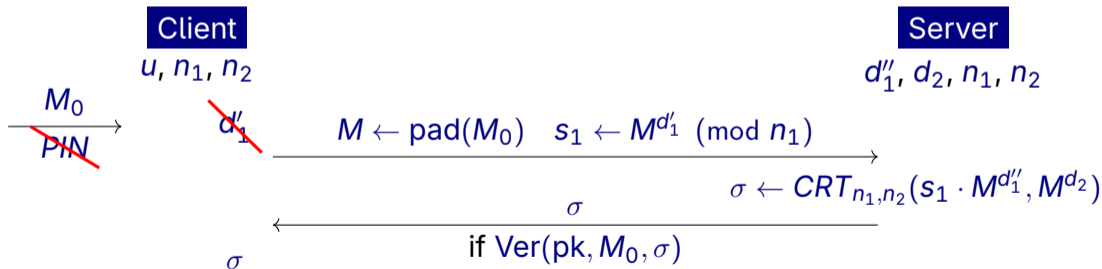
2016: SplitKey (signing)

$$\begin{aligned} \text{pk} &= (n_1 n_2, e) \quad n_i = p_i q_i \quad d_i \cdot e \equiv 1 \pmod{\varphi(n_i)} \\ d'_1 + d''_1 &\equiv d_1 \pmod{\varphi(n_1)} \quad u = \text{Enc}(\text{PIN}, d'_1) \end{aligned}$$



2016: SplitKey (signing)

$$\text{pk} = (n_1 n_2, e) \quad n_i = p_i q_i \quad d_i \cdot e \equiv 1 \pmod{\varphi(n_i)}$$
$$d'_1 + d''_1 \equiv d_1 \pmod{\varphi(n_1)} \quad u = \text{Enc}(\text{PIN}, d'_1)$$



Clone detection

- Additionally, Client and Server share a random bitstring r
- During signing, Client also sends r to Server
- Server checks that the value of r is as expected
 - If not, Server disqualifies this Client / this pk
- Server generates a new r and sends it back, together with σ

How SplitKey addresses the challenges

- "Leak of clients encrypted memory" u, n_1, n_2
 - Adversary tries to brute-force PIN and find d'_1 . Cannot verify its guess without contacting Server
 - Server maintains "wrong PIN counter"
- "Leak of clients unencrypted memory" d'_1, u, n_1, n_2
 - Adversary can issue signing requests until Client also issues a request
- "Clone detection" allows Server to reset "wrong PIN counter" in case of successful signing
- If Adversary takes over Client, then it can sign whatever it likes
 - *Decision to sign* a message is made by Client alone

Two-party RSA modulus (non)-generation

- Do not want to have the private key in one place, ever
- Protocols for RSA modulus / private key generation are expensive
- Widely adopted standards accepted and continue to accept RSA signatures of all sizes
 - We have *functional interchangeability*

Certification of SplitKey

- Smart-ID service built on SplitKey technology has been certified as a Remote QSCD (RQSCD)
 - TÜV Nord, Certificate IDs 9265.23, 9266.24, 9803.23
 - (different protection profile (PP), compared to QSCD)

A SplitKey signature has (may have) the same evidentiary value as a handwritten signature

Certification of SplitKey

- Smart-ID service built on SplitKey technology has been certified as a Remote QSCD (RQSCD)
 - TÜV Nord, Certificate IDs 9265.23, 9266.24, 9803.23
 - (different protection profile (PP), compared to QSCD)
- This PP requires the private key to be protected by hardware
 - SplitKey deployments use HSMs that do RSA signatures
 - ...and are certified

A SplitKey signature has (may have) the same evidentiary value as a handwritten signature

Split-ECDSA

$$\mathbb{G} = (E(\mathbb{F}_p), G, q) \quad pk_i = sk_i \cdot G \quad pk = pk_1 + pk_2 \quad u = \text{Enc}(PIN, sk_1)$$

Client

u, pk

Server

sk_2, pk_1, pk_2

Split-ECDSA

$$\mathbb{G} = (E(\mathbb{F}_p), G, q) \quad pk_i = sk_i \cdot G \quad pk = pk_1 + pk_2 \quad u = \text{Enc}(PIN, sk_1)$$



Split-ECDSA

$$\mathbb{G} = (E(\mathbb{F}_p), G, q) \quad pk_i = sk_i \cdot G \quad pk = pk_1 + pk_2 \quad u = \text{Enc}(PIN, sk_1)$$

Client

$$\begin{aligned} &u, pk \\ &\hat{sk}_1 \leftarrow \text{Dec}(\widehat{PIN}, u) \\ &\hat{pk}_1 \leftarrow \hat{sk}_1 \cdot G \end{aligned}$$

$$\begin{array}{c} M_0 \\ \xrightarrow{\widehat{PIN}} \end{array}$$

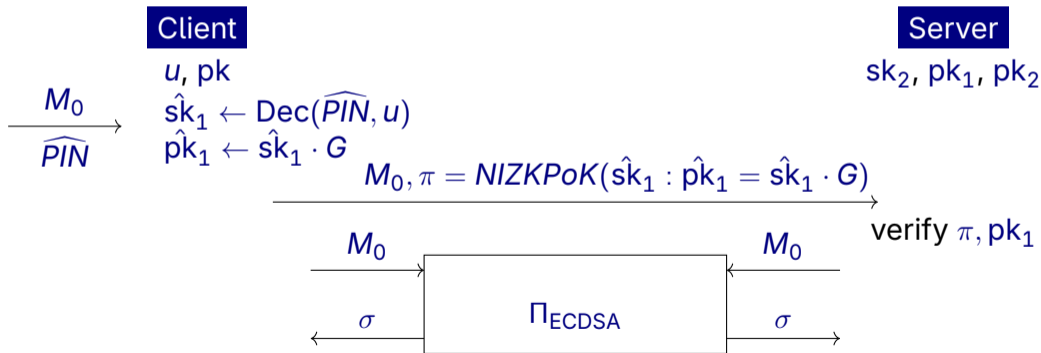
Server

$$sk_2, pk_1, pk_2$$



Split-ECDSA

$$\mathbb{G} = (E(\mathbb{F}_p), G, q) \quad pk_i = sk_i \cdot G \quad pk = pk_1 + pk_2 \quad u = \text{Enc}(PIN, sk_1)$$



Split - decryption

- We have signing. What about threshold decryption?
- Use - case: storing private credentials on the phone
 - Decrypting them before using them

Split - decryption

- We have signing. What about threshold decryption?
- Use - case: storing private credentials on the phone
 - Decrypting them before using them
- Π_{decr} similar to Π_{ECDSA} . Client sends ciphertext c and NIZKPoK for sk_1 .
 - Basis: Shoup - Gennaro: ElGamal with simulatable PoK of the ephemeral key

Split - decryption

- We have signing. What about threshold decryption?
- Use - case: storing private credentials on the phone
 - Decrypting them before using them
- Π_{decr} similar to Π_{ECDSA} . Client sends ciphertext c and NIZKPoK for sk_1 .
 - Basis: Shoup - Gennaro: ElGamal with simulatable PoK of the ephemeral key
- Issue: privacy / unlinkability against the server
 - (same issue also exists in signing. But less)

Unlinkable decryption requests

- $pk \in \mathbb{G}$. Ciphertext c contains $U = r \cdot G$, $V = Enc_{H(r \cdot pk)}(M)$, PoK π_1 of r

Unlinkable decryption requests

- $pk \in \mathbb{G}$. Ciphertext c contains $U = r \cdot G$, $V = Enc_{H(r.pk)}(M)$, PoK π_1 of r
- When Client sends U to Server, it must be blinded. E.g. $z \leftarrow \$ \mathbb{Z}_q$, $U' \leftarrow z \cdot U$
- Must also blind π_1 . Do not know how to do it for Chaum-Pedersen proofs
 - Perhaps Groth-Sahai proofs can be adapted. Use bilinear pairings

Unlinkable decryption requests

- $pk \in \mathbb{G}$. Ciphertext c contains $U = r \cdot G$, $V = Enc_{H(r \cdot pk)}(M)$, PoK π_1 of r
- When Client sends U to Server, it must be blinded. E.g. $z \leftarrow \$ \mathbb{Z}_q$, $U' \leftarrow z \cdot U$
- Must also blind π_1 . Do not know how to do it for Chaum-Pedersen proofs
 - Perhaps Groth-Sahai proofs can be adapted. Use bilinear pairings
- Certain designated-verifier NIZK proofs are blindable as needed
 - Add such proof π_2 to the ciphertext c
 - Server is the designated verifier. There are more keypairs involved

Unlinkable decryption requests

- $pk \in \mathbb{G}$. Ciphertext c contains $U = r \cdot G$, $V = Enc_{H(r \cdot pk)}(M)$, PoK π_1 of r
- When Client sends U to Server, it must be blinded. E.g. $z \leftarrow \$ \mathbb{Z}_q$, $U' \leftarrow z \cdot U$
- Must also blind π_1 . Do not know how to do it for Chaum-Pedersen proofs
 - Perhaps Groth-Sahai proofs can be adapted. Use bilinear pairings
- Certain designated-verifier NIZK proofs are blindable as needed
 - Add such proof π_2 to the ciphertext c
 - Server is the designated verifier. There are more keypairs involved
- Add also a proof π_3 that convinces Client that π_2 would convince Server
- During decryption, Client verifies π_1, π_3 , sends blinded U and π_2 to Server

Formalization of security properties

- Complex / complicated security requirements. Easy to get wrong
- We have given ideal functionalities, capturing the details
 - Various corruption levels of Client
 - Guessing the PIN by the adversary
- In real functionality, $\mathcal{M}_{\text{client}}$ must support adversarial commands "leak encrypted memory" and "leak unencrypted memory", besides "full corruption"

On ideal functionality for Split-decryption

- We found no *universally composable* treatment of threshold encryption in the literature
- The design of the ideal functionality $\mathcal{F}_{\text{thr-enc}}$ revealed some interesting details that had to be there:
 - If a privileged coalition has not been corrupted, but a corrupted party is part of the coalition currently decrypting c , then \mathcal{A} can ask \mathcal{F} for the corresponding plaintext m

On ideal functionality for Split-decryption

- We found no *universally composable* treatment of threshold encryption in the literature
- The design of the ideal functionality $\mathcal{F}_{\text{thr-enc}}$ revealed some interesting details that had to be there:
 - If a privileged coalition has not been corrupted, but a corrupted party is part of the coalition currently decrypting c , then \mathcal{A} can ask \mathcal{F} for the corresponding plaintext m
- In case of $\mathcal{F}_{\text{split-decr}}$, corrupt Client, honest Server, \mathcal{A} can ask for mappings $c \mapsto m$ at any time
 - Rate-limited by the number of decryption queries

2026: post-quantum SplitKey-like signing?

- Pick a standard scheme, e.g. ML-DSA
- Design a two-party protocol first, then add SplitKey-like properties

Recall steps of an ML-DSA signing **attempt**

- Generate ephemeral secret $\mathbf{y} \leftarrow \text{ExpandMask}(\text{seed})$
- Compute commitment $\mathbf{w}^H \leftarrow \text{HighBits}(\mathbf{A} \cdot \mathbf{y})$
- Compute challenge $c \leftarrow H(\mathbf{w}^H, M)$
- Compute response $\mathbf{z} = c \cdot \text{sk} + \mathbf{y}$ and perform *rejection check*

Trilithium

- Based on techniques of Secure Multiparty Computation (MPC)
- Two parties. And a “trusted” party (CRP) for providing correlated randomness

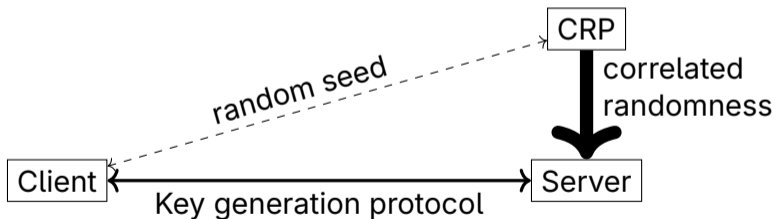
Trilithium

- Based on techniques of Secure Multiparty Computation (MPC)
- Two parties. And a "trusted" party (CRP) for providing correlated randomness



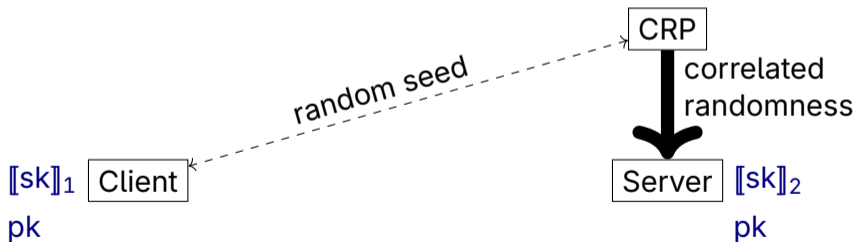
Trilithium

- Based on techniques of Secure Multiparty Computation (MPC)
- Two parties. And a "trusted" party (CRP) for providing correlated randomness



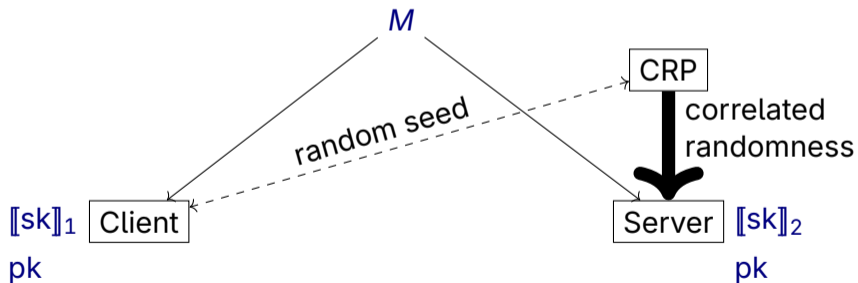
Trilithium

- Based on techniques of Secure Multiparty Computation (MPC)
- Two parties. And a "trusted" party (CRP) for providing correlated randomness



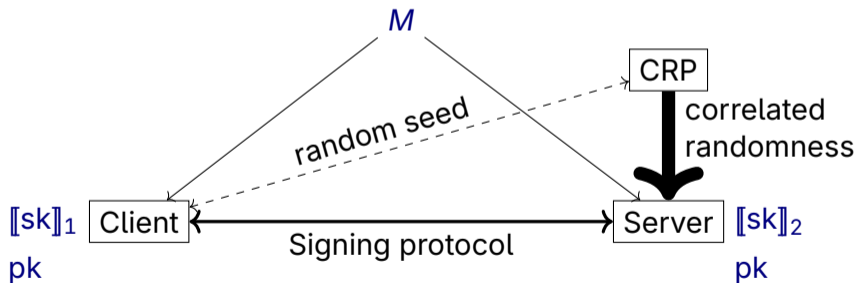
Trilithium

- Based on techniques of Secure Multiparty Computation (MPC)
- Two parties. And a "trusted" party (CRP) for providing correlated randomness



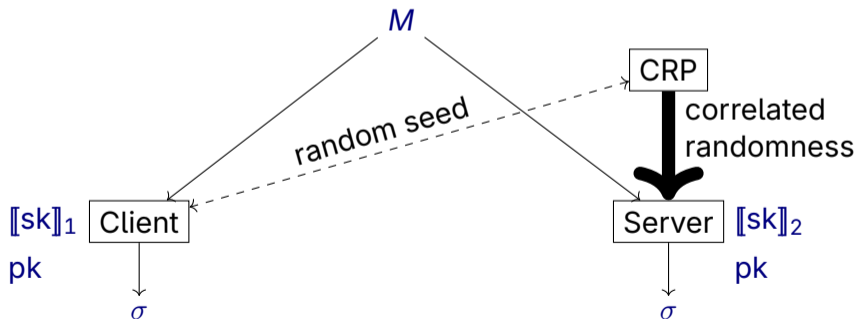
Trilithium

- Based on techniques of Secure Multiparty Computation (MPC)
- Two parties. And a "trusted" party (CRP) for providing correlated randomness



Trilithium

- Based on techniques of Secure Multiparty Computation (MPC)
- Two parties. And a "trusted" party (CRP) for providing correlated randomness



Network traffic

Key generation: traffic volume

ML - DSA -	P↔S	CRP→S
-44	920.52 KiB	1.42 MiB
-65	1760.22 KiB	2.66 MiB
-87	3704.24 KiB	4.44 MiB

Signing **attempt**: traffic volume

ML - DSA -	P↔S	CRP→S
-44	186.72 KiB	55.05 MiB
-65	244.86 KiB	71.23 MiB
-87	328.50 KiB	95.95 MiB

Network traffic

Key generation: traffic volume

ML-DSA-	P↔S	CRP→S
-44	920.52 KiB	1.42 MiB
-65	1760.22 KiB	2.66 MiB
-87	3704.24 KiB	4.44 MiB

Signing **attempt**: traffic volume

ML-DSA-	P↔S	CRP→S
-44	186.72 KiB	55.05 MiB
-65	244.86 KiB	71.23 MiB
-87	328.50 KiB	95.95 MiB

Key generation: num. of rounds

3

Signing **attempt**: num. of rounds

14

Security model

As two - party protocol

- (output to CRP: nothing)
- Secure against one malicious party
- (additionally, a trusted CRP)

As three - party protocol

- (output to CRP: signature or rejection)
- Secure against one malicious party
- A *selective failure attack* is available for the CRP

Trilithium: a few details

- Commitment **y** generation: standard MPC techniques, no [ExpandMask](#)

Trilithium: a few details

- Commitment y generation: standard MPC techniques, no [ExpandMask](#)
- Computing w^H : standard MPC techniques, applied to an optimized circuit for [HighBits](#)
 - Different from Quorus (tomorrow); a combination may be fruitful

Trilithium: a few details

- Commitment y generation: standard MPC techniques, no [ExpandMask](#)
- Computing w^H : standard MPC techniques, applied to an optimized circuit for [HighBits](#)
 - Different from Quorus (tomorrow); a combination may be fruitful
- Computing $c = H(w^H, M)$: in the clear
 - We argue that it is safe to open w^H . Quorus gives an alternative approach
 - Can evaluate H with garbled circuits. [Mohassel et al., CCS 2015] fits nicely

Trilithium: a few details

- Commitment y generation: standard MPC techniques, no [ExpandMask](#)
- Computing w^H : standard MPC techniques, applied to an optimized circuit for [HighBits](#)
 - Different from Quorus (tomorrow); a combination may be fruitful
- Computing $c = H(w^H, M)$: in the clear
 - We argue that it is safe to open w^H . Quorus gives an alternative approach
 - Can evaluate H with garbled circuits. [Mohassel et al., CCS 2015] fits nicely
- Rejection check: adapt the two-party passively-secure protocol of [Attrapadung et al., ASIACCS 2022]
 - This part of protocol is secure against active Client, passive Server only
 - Provides only *privacy* against active Server
 - But the whole Trilithium protocol has security against active Server

What is “(functional) interchangability” for a protocol Π and a signature algorithm Sig ?

Possible options

- $\text{Sig.verify}(\text{pk}, \sigma, \mu) = 1$ for $\Pi.\text{sign}(\mu)$?
- $\Pi.\text{keygen}().\text{pk} \approx \text{Sig.keygen}().\text{pk}$ and $\text{Sig.sign}(\text{sk}, \mu) \approx \Pi.\text{sign}(\mu)$?
- Π securely implements $\mathcal{F}_{\text{thresholdized-Sig}}$?
- Π can be made to pass KATs (or other tests that certification labs may use) for (plain) Sig ?

(Also, Π has to be a *secure* threshold signature protocol)

Participating in an ecosystem

- Need to get the approval from the gatekeepers of the ecosystem
- This call can help to obtain that approval

Ideal functionality for thresholdized Sig

Key generation

On input (keygen) from a privileged coalition P_{i_1}, \dots, P_{i_k} , ① run $(pk, sk) \leftarrow \text{Sig.keygen}()$, ② store sk , ③ send pk to \mathcal{A} and honest P_{i_1}, \dots, P_{i_k}

Signing

On input (sign, sid, μ) from a privileged coalition P_{i_1}, \dots, P_{i_k} , ① run $\sigma \leftarrow \text{Sig.sign}(sk, \mu)$, ② send (sign, sid, μ, σ) to \mathcal{A} , ③ get (proceed, sid) from \mathcal{A} , ④ send (result, sid, σ) to honest P_{i_1}, \dots, P_{i_k}

The team

- Name: SplitForge
- Intention to submit: Split-RSA, Split-ECDSA, Split-decryption, Trilithium
- Members: Aivo Kalu, Alisa Pankova, Burak Can Kus, Jelizaveta Vakarjuk*, Mart Oruaas, Nikita Snetkov*, Peeter Laud, Petr Muzikant, Raul-Martin Rebane, Semjon/Sona Kravtšenko
- Everybody works in Cybernetica AS
 - * also PhD student in Tallinn University of Technology



Thank you!

- Questions?



[cybernetica](#)



[Cybernetica](#)



[cybernetica_ee](#)



[Cybernetica](#)