Lightning Talks (Thursday)

Speaker	Institution	Title
Caroline Sandsbråten	NTNU	Zero-Knowledge Proofs in Applications
Audhild Høgåsen	Swiss Post	ZK-Proofs in the current and future Swiss Post E-Voting System
Hans Heum	NTNU	Quantum secure proof of shuffle
Emil August Hovd Olaisen	NTNU	Distributed Decryption Derived Verifiable Decryption
Artem Grigor	UCL	State of ZKP on mobile devices
Mahdi Sedaghat	COSIC, KU Leuven	zklogin: Privacy-preserving blockchain authentication with existing credentials
Jayamine Alupotha	University of Bern	Account-based Untraceable Payments: Defeating Graph Analysis with Small Decoy Sets
Thomas den Hollander	Universität der Bundeswehr München	A Crack in the Firmament Restoring Soundness of the Orion Proof System

NTNU | Norwegian University of Science and Technology

ZERO-KNOWLEDGE PROOFS IN APPLICATIONS

Foundations and Applications of Zero-Knowledge Proofs Workshop

Caroline Sandsbråten

05.09.2024

Caroline Sandsbråten

- PhD student in Cryptology at NTNU
- Researching lattice-based cryptography in distributed systems
- Also interested in PQ anonymous SSO and anonymous credentials

---

carosa.no

ntnu.edu/employees/caroline.sandsbraten

Contents

Caveats

ZKPs in E-Voting

ZKPs in Distributed Key Generation

ZKPs in Threshold Signatures

Summary

Contents

Caveats

ZKPs in E-Voting

ZKPs in Distributed Key Generation

ZKPs in Threshold Signatures

Summary

- My own work focus mostly on applications of lattice based protocols.
- Most of these applications of zero-knowledge are therefore from the perspective of general lattice-based applications.
- I have tried to make it applicable to everyone not necessarily interested in lattices as well, but some parts will include lattice-specific proof requirements.

Contents

Caveats

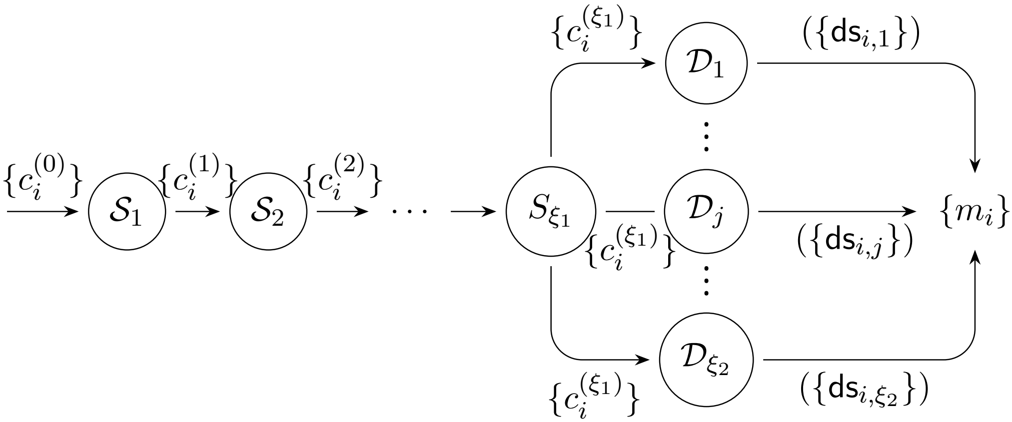
ZKPs in E-Voting

ZKPs in Distributed Key Generation

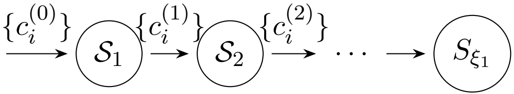
ZKPs in Threshold Signatures

Summary

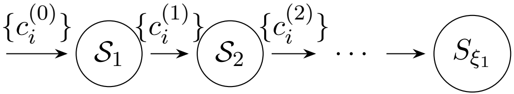
ZKPs in E-Voting



Zooming in on shuffling

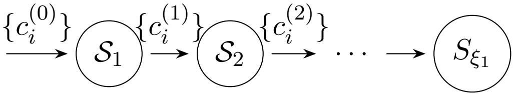


Zooming in on shuffling



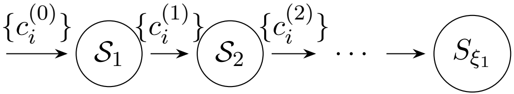
- Input-output ciphertexts correspondence must be obscured.

Zooming in on shuffling



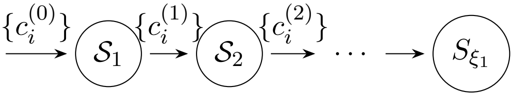
- Input-output ciphertexts correspondence must be obscured.
- The set of output ciphertexts must decrypt to the same set of plaintexts.

Zooming in on shuffling



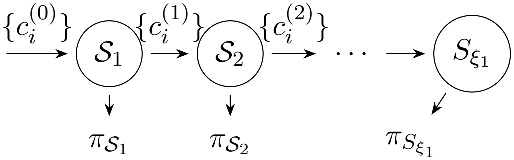
- Input-output ciphertexts correspondence must be obscured.
- The set of output ciphertexts must decrypt to the same set of plaintexts.
- Ciphertext noise must be bounded.

Zooming in on shuffling



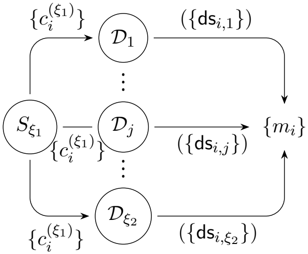
- Input-output ciphertexts correspondence must be obscured.
- The set of output ciphertexts must decrypt to the same set of plaintexts.
- Ciphertext noise must be bounded.
- Anyone should be able to verify this.

Zooming in on shuffling

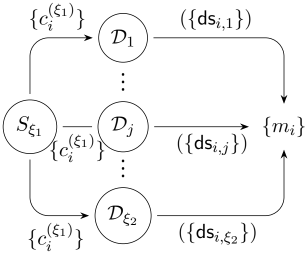


- Input-output ciphertexts correspondence must be obscured.
- The set of output ciphertexts must decrypt to the same set of plaintexts.
- Ciphertext noise must be bounded.
- Anyone should be able to verify this.

Zooming in on decryption

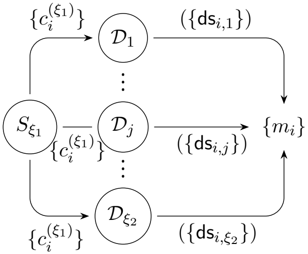


Zooming in on decryption



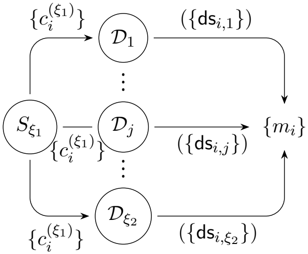
- Encryption of the plaintexts must be correct.

Zooming in on decryption



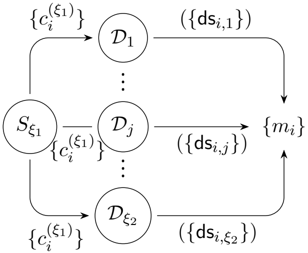
- Encryption of the plaintexts must be correct.
- Avoiding leaking $\rightarrow$ requires noise drowning.

Zooming in on decryption



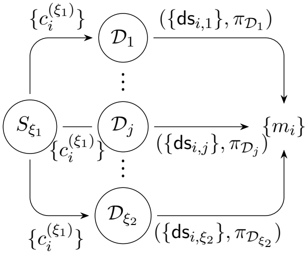
- Encryption of the plaintexts must be correct.
- Avoiding leaking $\rightarrow$ requires noise drowning.
- Need a subsequent proof that noise drowning has been applied.

Zooming in on decryption



- Encryption of the plaintexts must be correct.
- Avoiding leaking $\rightarrow$ requires noise drowning.
- Need a subsequent proof that noise drowning has been applied.
- Need to prove the well-formedness of $\mathsf{ds}_{i,j}$.

Zooming in on decryption



- Encryption of the plaintexts must be correct.
- Avoiding leaking $\rightarrow$ requires noise drowning.
- Need a subsequent proof that noise drowning has been applied.
- Need to prove the well-formedness of $\mathsf{ds}_{i,j}$.

Contents

Caveats

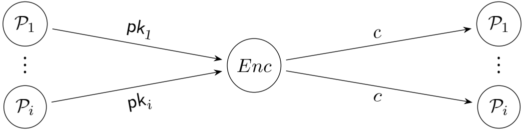
ZKPs in E-Voting

ZKPs in Distributed Key Generation

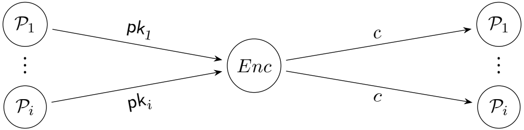
ZKPs in Threshold Signatures

Summary

ZKPs in Distributed Key Generation

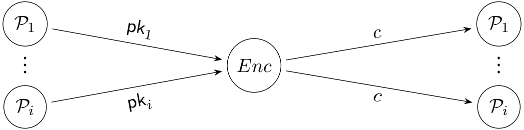


ZKPs in Distributed Key Generation



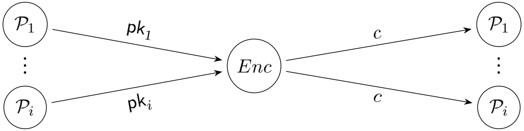
- $\mathcal{P}_i$ needs to prove that the public key is well-formed and satisfy some norm bound.

ZKPs in Distributed Key Generation



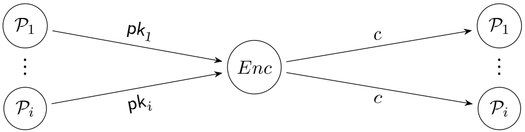
- $\mathcal{P}_i$ needs to prove that the public key is well-formed and satisfy some norm bound.
- Each party should idealy be able to abort if the key pair of any other party is not generated correctly.

ZKPs in Distributed Key Generation



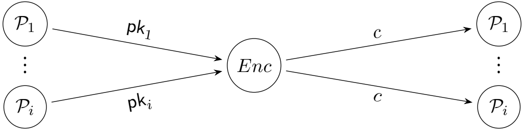
- $\mathcal{P}_i$ needs to prove that the public key is well-formed and satisfy some norm bound.
- Each party should idealy be able to abort if the key pair of any other party is not generated correctly.
- The encryptor needs to be able to verify this.

ZKPs in Distributed Key Generation



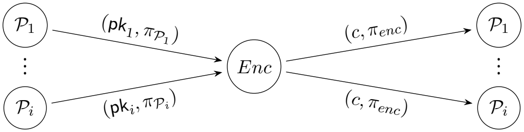
- $\mathcal{P}_i$ needs to prove that the public key is well-formed and satisfy some norm bound.
- Each party should idealy be able to abort if the key pair of any other party is not generated correctly.
- The encryptor needs to be able to verify this.
- The encryptor then needs to prove that the ciphertext is well-formed.

ZKPs in Distributed Key Generation



- $\mathcal{P}_i$ needs to prove that the public key is well-formed and satisfy some norm bound.
- Each party should idealy be able to abort if the key pair of any other party is not generated correctly.
- The encryptor needs to be able to verify this.
- The encryptor then needs to prove that the ciphertext is well-formed.
- Again, we would ideally like anyone to be able to verify this.

ZKPs in Distributed Key Generation



- $\mathcal{P}_i$ needs to prove that the public key is well-formed and satisfy some norm bound.
- Each party should idealy be able to abort if the key pair of any other party is not generated correctly.
- The encryptor needs to be able to verify this.
- The encryptor then needs to prove that the ciphertext is well-formed.
- Again, we would ideally like anyone to be able to verify this.

Contents

Caveats

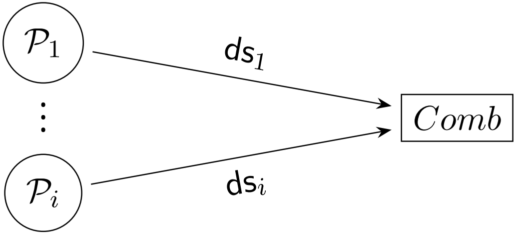
ZKPs in E-Voting

ZKPs in Distributed Key Generation

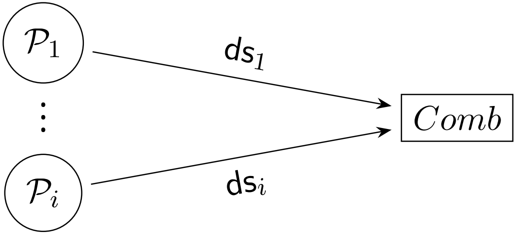
ZKPs in Threshold Signatures

Summary

ZKPs in Threshold Signatures

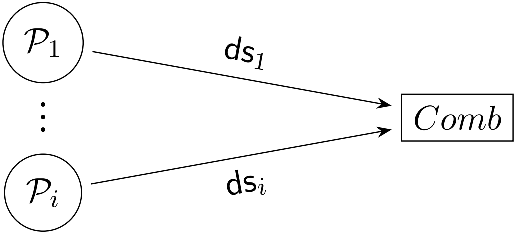


ZKPs in Threshold Signatures



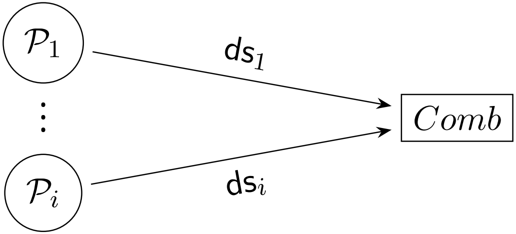
- Prove that the correct randomness is used.

ZKPs in Threshold Signatures



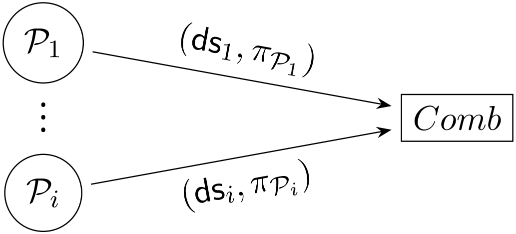
- Prove that the correct randomness is used.
- Prove that the partial signatures are well-formed.

ZKPs in Threshold Signatures



- Prove that the correct randomness is used.
- Prove that the partial signatures are well-formed.
- The underlying keypairs should also be proven to be generated correctly.

ZKPs in Threshold Signatures



- Prove that the correct randomness is used.
- Prove that the partial signatures are well-formed.
- The underlying keypairs should also be proven to be generated correctly.

Contents

Caveats

ZKPs in E-Voting

ZKPs in Distributed Key Generation

ZKPs in Threshold Signatures

Summary

Summary

Summary

- Zero-knowledge proofs are a powerful tool in the cryptographic toolbox for applications in distributed systems, including but not limited to distributed key generation, threshold signatures and electronic voting.

Summary

- Zero-knowledge proofs are a powerful tool in the cryptographic toolbox for applications in distributed systems, including but not limited to distributed key generation, threshold signatures and electronic voting.

- They can be used to prove the correctness of key-generation generation (and more).

Summary

- Zero-knowledge proofs are a powerful tool in the cryptographic toolbox for applications in distributed systems, including but not limited to distributed key generation, threshold signatures and electronic voting.

- They can be used to prove the correctness of key-generation generation (and more).

- They can be used to prove well-formedness.

Summary

- Zero-knowledge proofs are a powerful tool in the cryptographic toolbox for applications in distributed systems, including but not limited to distributed key generation, threshold signatures and electronic voting.

- They can be used to prove the correctness of key-generation generation (and more).

- They can be used to prove well-formedness.

- They can be used to prove certain properties needed to ensure security in distributed systems.

Summary

- Zero-knowledge proofs are a powerful tool in the cryptographic toolbox for applications in distributed systems, including but not limited to distributed key generation, threshold signatures and electronic voting.
- They can be used to prove the correctness of key-generation generation (and more).
- They can be used to prove well-formedness.
- They can be used to prove certain properties needed to ensure security in distributed systems.
- They can be used to prove some party has performed some operation in the excpected way defined by a protocol.

Questions?

ZK-Proofs in the Current and Future Swiss Post Voting System

Audhild Høgåsen
audhild.hoegaasen@post.ch

2015-2022 Master’s Degree Mathematics

Norwegian University of Science and Technology (NTNU)
University of Innsbruck
University of Bern

2022 Master’s thesis: *Return Codes from Lattice Assumptions*
Supervisors: Kristian Gjøsteen and Tjerand Silde

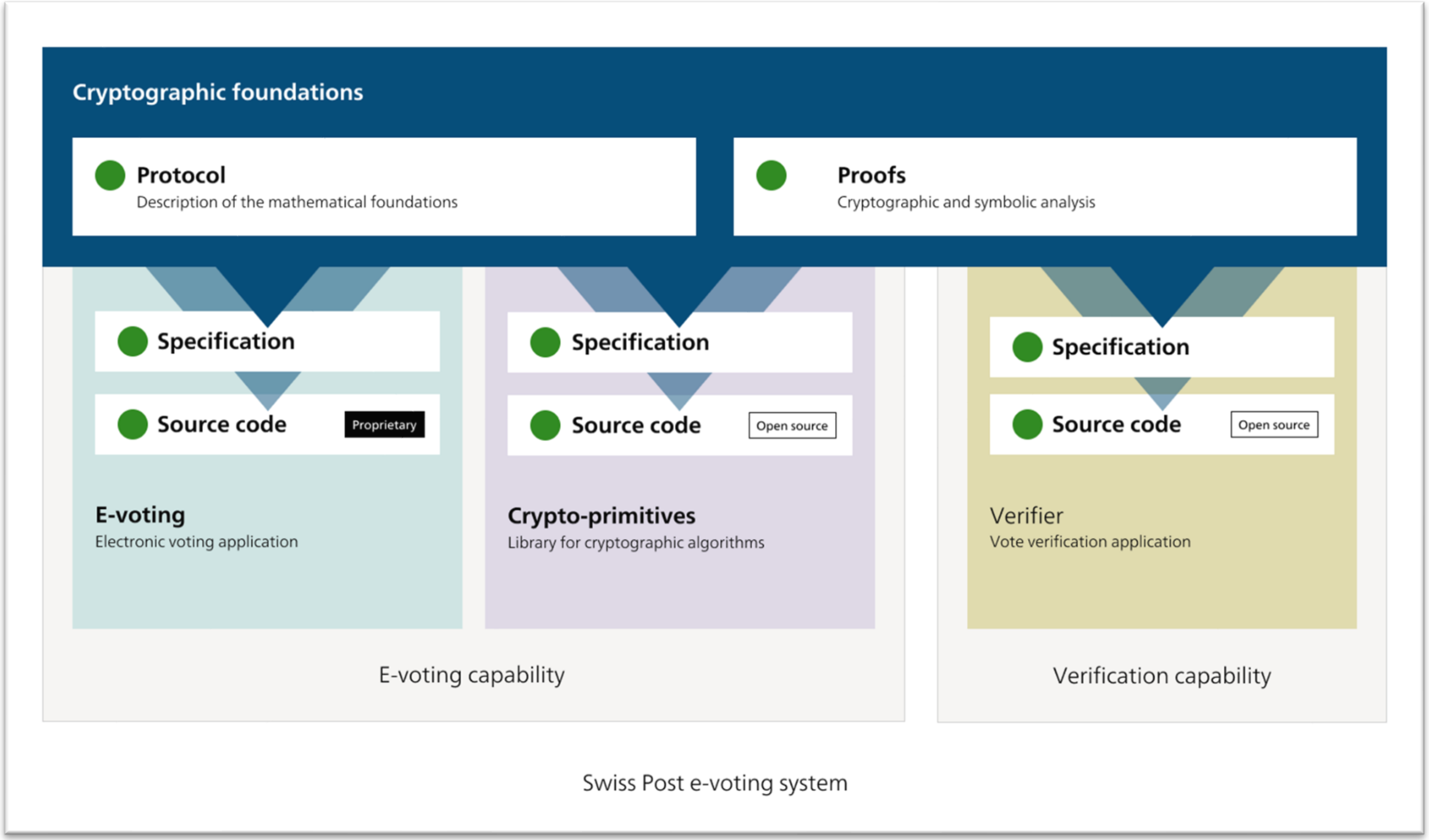
* Short paper: *Return Codes from Lattice Assumptions*, E-Vote-ID Conference 2022. Joint work with Tjerand Silde.

2022 - current Team E-Voting at Swiss Post

Bern, Switzerland

* Paper: *Improving the Swiss Post Voting System: Practical Experiences from the Independent Examination and First Productive Election Event,* E-Vote-ID Conference 2023.
* Co-supervision of two NTNU-students (2023-2024) for the Master’s thesis *Next Generation Electronic Voting in Switzerland*. Main supervisor: Tjerand Silde.

Swiss Post Voting System



The Swiss Post Voting System is an electronic voting system in use in national and cantonal elections in Switzerland.

Ca 4 elections per year (direct democracy). E-voting as additional (optional) voting channel. (Most people in Switzerland vote by postal voting.)

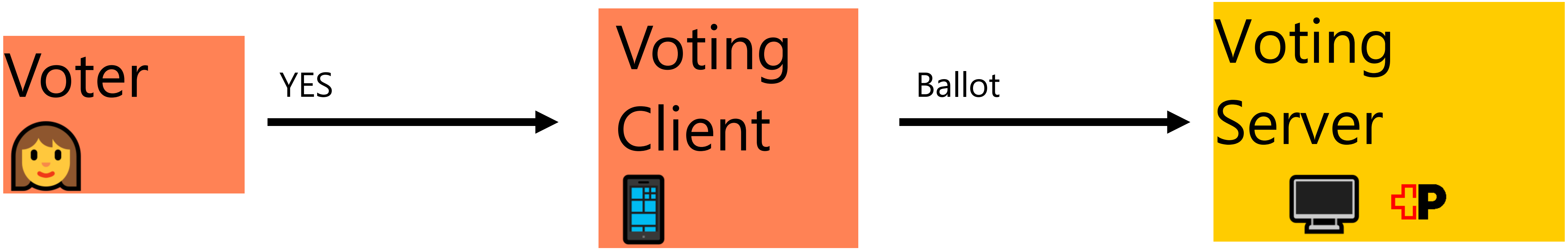
All documentation published on GitLab.

In the e-voting setting, NIZK-Proofs play an important role to ensure vote secrecy and verifiability.

Where in the Swiss Post Voting System are NIZK-Proofs used?

Voting phase: Creation of the ballot

includes ZK-proofs of correct creation



Create ballot
* Ciphertext = Encrypt(YES)
* **ZK-Proofs**

Voting phase: Creation of the ballot

pseudocode for generating ZK-Proofs of the ballot

Swiss Post Voting System
System Specification

© 2024 Swiss Post Ltd.
Version 1.4.1

Algorithm 5.4 CreateVote

Context:

Group modulus $p \in \mathbb{P}$
Group cardinality $q \in \mathbb{P}$ s.t. $p = 2q + 1$
Group generator $g \in \mathbb{G}_q$
Election event ID $\mathsf{ee} \in (\mathbb{A}_{Base16})^{l_{\mathsf{ID}}}$
Verification card set ID $\mathsf{vcs} \in (\mathbb{A}_{Base16})^{l_{\mathsf{ID}}}$
Verification card ID $\mathsf{vc_{id}} \in (\mathbb{A}_{Base16})^{l_{\mathsf{ID}}}$
Primes mapping table $\mathsf{pTable} \in (\mathcal{T}_1^{50} \times ((\mathbb{G}_q \cap \mathbb{P}) \setminus g) \times \mathbb{A}_{UCS}^* \times \mathcal{T}_1^{50})^n$ ▷ pTable is of the form $((\mathsf{v}_0, \tilde{\mathsf{p}}_0, \sigma_0, \tau_0), \dots, (\mathsf{v}_{n-1}, \tilde{\mathsf{p}}_{n-1}, \sigma_{n-1}, \tau_{n-1}))$ ▷ ψ and δ can be derived from pTable using algorithms 3.9 and 3.10
Election public key $\mathsf{EL_{pk}} = (\mathsf{EL_{pk,0}}, \dots, \mathsf{EL_{pk,\delta_{max}-1}}) \in \mathbb{G}_q^{\delta_{max}}$
Choice Return Codes encryption public key $\mathbf{pk}_{\mathsf{CCR}} \in \mathbb{G}_q^{\psi_{max}}$

Input:

Selected actual voting options $\hat{\mathbf{v}}_{\mathsf{id}} = (\hat{\mathsf{v}}_0, \dots, \hat{\mathsf{v}}_{\psi-1}) \in \left(\mathcal{T}_1^{50}\right)^{\psi}$ ▷ See section 3.5
Selected write-ins $\hat{\mathbf{s}}_{\mathsf{id}} = (\hat{\mathsf{s}}_0, \dots, \hat{\mathsf{s}}_{k-1}) \in \left((\mathbb{A}_{\mathsf{latin}} \setminus \#)^*\right)^k$ ▷ See section 3.7
Verification card secret key $\mathsf{k_{id}} \in \mathbb{Z}_q$

Require: GetBlankCorrectnessInformation() = GetCorrectnessInformation($\hat{\mathbf{v}}_{\mathsf{id}}$) ▷ See algorithms 3.6 and 3.7.
The algorithm 3.6 ensures $\hat{\mathbf{v}}_{\mathsf{id}}$ is a subset of $\tilde{\mathbf{v}}$ and contains no duplicates.
Require: $k \le \delta - 1$ ▷ $\delta = 1$, if the ballot box does not have any write-in candidates.
Require: $|\hat{\mathsf{s}}_i| < \mathsf{l_w}, \forall i \in [0, k)$ ▷ where $|\hat{\mathsf{s}}_i|$ is the character length of $\hat{\mathsf{s}}_i$

Operation: ▷ For all algorithms see the crypto primitives specification

$(\hat{\mathsf{p}}_0, \dots, \hat{\mathsf{p}}_{\psi-1}) \leftarrow$ GetEncodedVotingOptions($\hat{\mathbf{v}}_{\mathsf{id}}$) ▷ See algorithm 3.3
$(\mathsf{w_{id,0}}, \dots, \mathsf{w_{id,\delta-2}}) \leftarrow$ EncodeWriteIns($\hat{\mathbf{s}}_{\mathsf{id}}$) ▷ See algorithm 3.19
$\rho \leftarrow \prod_{i=0}^{\psi-1} \hat{\mathsf{p}}_i \bmod p$
$r \leftarrow$ GenRandomInteger(q)
$\mathsf{E1} = (\gamma_1, \phi_{1,0}, \dots, \phi_{1,\delta-1}) \leftarrow$ GetCiphertext($(\rho, \mathsf{w_{id,0}}, \dots, \mathsf{w_{id,\delta-2}}), r, \mathsf{EL_{pk}}$)
for $i \in [0, \psi)$ **do**
    $\mathsf{pCC}_{\mathsf{id},i} \leftarrow \hat{\mathsf{p}}_i^{\mathsf{k_{id}}} \bmod p$
end for
$\mathbf{pCC}_{\mathsf{id}} = (\mathsf{pCC}_{\mathsf{id},0}, \dots, \mathsf{pCC}_{\mathsf{id},\psi-1})$
$r' \leftarrow$ GenRandomInteger(q)
$\mathsf{E2} = (\gamma_2, \phi_{2,0}, \dots, \phi_{2,\psi-1}) \leftarrow$ GetCiphertext($\mathbf{pCC}_{\mathsf{id}}, r', \mathbf{pk}_{\mathsf{CCR}}$)
$\widetilde{\mathsf{E1}} \leftarrow$ GetCiphertextExponentiation($(\gamma_1, \phi_{1,0}), \mathsf{k_{id}}$)
$\widetilde{\mathsf{E2}} \leftarrow (\gamma_2, \prod_{i=0}^{\psi-1} \phi_{2,i} \bmod p)$
$\mathsf{K_{id}} \leftarrow g^{\mathsf{k_{id}}} \bmod p$
$\mathbf{i}_{\mathsf{aux}} \leftarrow$ ("CreateVote", $\mathsf{vc_{id}}$, GetHashContext()) ▷ See algorithm 3.11
$\mathbf{i}_{\mathsf{aux}} \leftarrow (\mathbf{i}_{\mathsf{aux}}$, IntegerToString($\gamma_1$), IntegerToString($\phi_{1,0}$), …, IntegerToString($\phi_{1,\delta-1}$))
$\mathbf{i}_{\mathsf{aux}} \leftarrow (\mathbf{i}_{\mathsf{aux}}$, IntegerToString($\gamma_2$), IntegerToString($\phi_{2,0}$), …, IntegerToString($\phi_{2,\psi-1}$))
$\pi_{\mathsf{Exp}} \leftarrow$ GenExponentiationProof($(g, \gamma_1, \phi_{1,0}), \mathsf{k_{id}}, (\mathsf{K_{id}}, \gamma_1^{\mathsf{k_{id}}}, \phi_{1,0}^{\mathsf{k_{id}}}), \mathbf{i}_{\mathsf{aux}}$)
$\widetilde{\mathsf{pk}}_{\mathsf{CCR}} \leftarrow \prod_{i=0}^{\psi-1} \mathsf{pk}_{\mathsf{CCR},i} \bmod p$
$\pi_{\mathsf{EqEnc}} \leftarrow$ GenPlaintextEqualityProof($\widetilde{\mathsf{E1}}, \widetilde{\mathsf{E2}}, \mathsf{EL_{pk,0}}, \widetilde{\mathsf{pk}}_{\mathsf{CCR}}, (r \cdot \mathsf{k_{id}}, r'), \mathbf{i}_{\mathsf{aux}}$)

Output:

Encrypted vote $\mathsf{E1} = (\gamma_1, \phi_{1,0}, \dots, \phi_{1,\delta-1}) \in \mathbb{G}_q^{\delta+1}$
Encrypted partial Choice Return Codes $\mathsf{E2} = (\gamma_2, \phi_{2,0}, \dots, \phi_{2,\psi-1}) \in \mathbb{G}_q^{\psi+1}$
Exponentiated encrypted vote $\widetilde{\mathsf{E1}} \in \mathbb{G}_q^2$
Exponentiation proof $\pi_{\mathsf{Exp}} \in \mathbb{Z}_q \times \mathbb{Z}_q$
Plaintext equality proof $\pi_{\mathsf{EqEnc}} \in \mathbb{Z}_q \times \mathbb{Z}_q^2$

Cryptographic Primitives of the Swiss Post Voting System
Pseudocode Specification

© 2024 Swiss Post Ltd.
Version 1.4.1

Generating and verifying exponentiation proofs The algorithms below are the adaptations of the general case presented in section 10.1, with explicit domains and operations. Our phi-function defined in algorithm 10.7 has domain $(\mathbb{Z}_q, +)$ and co-domain $(\mathbb{G}_q^n, \times)$. Therefore the operations given as $\star$ will be replaced with addition (modulo q), and the "exponentiation" used in the computation of z is a multiplication; whereas the operation denoted by $\otimes$ is multiplication (modulo p) and the exponentiation used in the computation of c' is a modular exponentiation in $\mathbb{G}_q$.

Algorithm 10.8 GenExponentiationProof: Generate a proof of validity for the provided exponentiation

Context:

Group modulus $p \in \mathbb{P}$
Group cardinality $q \in \mathbb{P}$ s.t. $p = 2 \cdot q + 1$

Input:

A vector of bases $\mathbf{g} = (g_0, \dots, g_{n-1}) \in \mathbb{G}_q^n$ s.t. $n \in \mathbb{N}^+$
The witness – a secret exponent $x \in \mathbb{Z}_q$
The statement – a vector of exponentiations $\mathbf{y} = (y_0, \dots, y_{n-1}) \in \mathbb{G}_q^n$ s.t. $y_i = g_i^x$
An array of optional additional information $\mathbf{i}_{\mathsf{aux}} \in (\mathbb{A}_{UCS}^*)^s, s \in \mathbb{N}$

Operation:

$b \leftarrow$ GenRandomInteger(q) ▷ See algorithm 5.1
$\mathbf{c} \leftarrow$ ComputePhiExponentiation($b, \mathbf{g}$) ▷ See algorithm 10.7
$\mathbf{f} \leftarrow (p, q, \mathbf{g})$
$\mathbf{h}_{\mathsf{aux}} \leftarrow$ ("ExponentiationProof", $\mathbf{i}_{\mathsf{aux}}$) ▷ If $\mathbf{i}_{\mathsf{aux}}$ is empty, we omit it
$e \leftarrow$ ByteArrayToInteger(RecursiveHash($\mathbf{f}, \mathbf{y}, \mathbf{c}, \mathbf{h}_{\mathsf{aux}}$)) ▷ See algorithms 3.8 and 5.5
$z \leftarrow b + e \cdot x \bmod q$

Output:

Proof $(e, z) \in \mathbb{Z}_q \times \mathbb{Z}_q$

Cryptographic Primitives of the Swiss Post Voting System
Pseudocode Specification

© 2024 Swiss Post Ltd.
Version 1.4.1

Generating and verifying plaintext equality proofs The algorithms below are the adaptations of the general case presented in section 10.1, with explicit domains and operations. Our phi-function defined in algorithm 10.10 has domain $(\mathbb{Z}_q^2, +)$ and co-domain $(\mathbb{G}_q^3, \times)$. Therefore the operations given as $\star$ will be replaced with addition (modulo q), and the "exponentiation" used in the computation of z is a multiplication; whereas the operation denoted by $\otimes$ is multiplication (modulo p) and the exponentiation used in the computation of c' is a modular exponentiation in $\mathbb{G}_q$.

Algorithm 10.11 GenPlaintextEqualityProof: Generate a proof of equality of the plaintext corresponding to the two provided encryptions

Context:

Group modulus $p \in \mathbb{P}$
Group cardinality $q \in \mathbb{P}$ s.t. $p = 2 \cdot q + 1$
Group generator $g \in \mathbb{G}_q$

Input:

The first ciphertext $\mathbf{C} = (c_0, c_1) \in \mathbb{G}_q^2$
The second ciphertext $\mathbf{C}' = (c'_0, c'_1) \in \mathbb{G}_q^2$
The first public key $h \in \mathbb{G}_q$
The second public key $h' \in \mathbb{G}_q$
The witness—the randomness used in the encryptions—$(r, r') \in \mathbb{Z}_q^2$
An array of optional additional information $\mathbf{i}_{\mathsf{aux}} \in (\mathbb{A}_{UCS}^*)^s, s \in \mathbb{N}$

Operation:

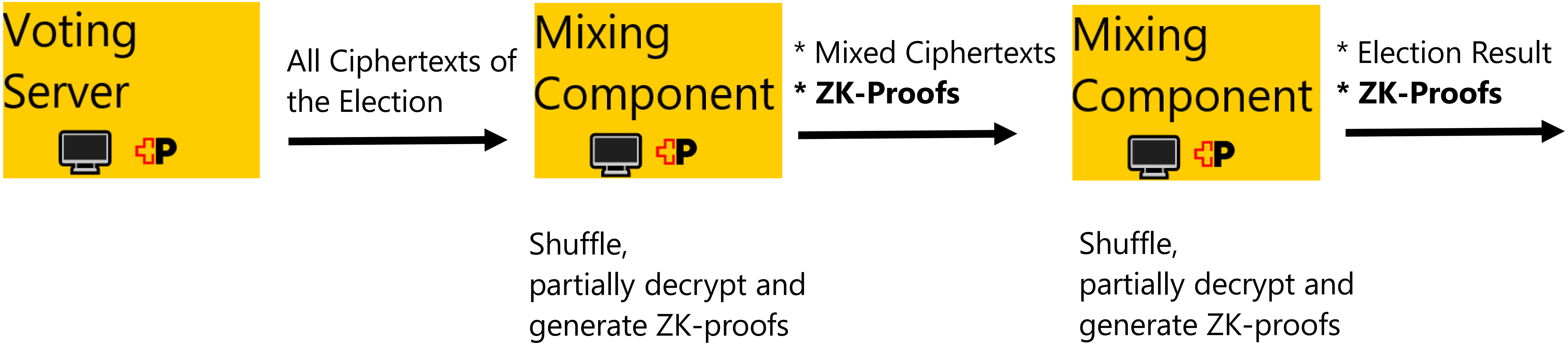
$(b_1, b_2) \leftarrow$ GenRandomVector($q, 2$) ▷ See algorithm 5.2
$\mathbf{c} \leftarrow$ ComputePhiPlaintextEquality($(b_1, b_2), h, h'$) ▷ See algorithm 10.10
$\mathbf{f} \leftarrow (p, q, g, h, h')$
$\mathbf{y} \leftarrow (c_0, c'_0, \frac{c_1}{c'_1})$
$\mathbf{h}_{\mathsf{aux}} \leftarrow$ ("PlaintextEqualityProof", $c_1, c'_1, \mathbf{i}_{\mathsf{aux}}$) ▷ If $\mathbf{i}_{\mathsf{aux}}$ is empty, we omit it
$e \leftarrow$ ByteArrayToInteger(RecursiveHash($\mathbf{f}, \mathbf{y}, \mathbf{c}, \mathbf{h}_{\mathsf{aux}}$)) ▷ See algorithms 3.8 and 5.5
$\mathbf{z} \leftarrow (b_1 + e \cdot r, b_2 + e \cdot r')$

Output:

Proof $(e, \mathbf{z}) \in \mathbb{Z}_q \times \mathbb{Z}_q^2$

Tally phase: Mix net

includes ZK-Proofs of correct shuffle and correct partial decryption



Tally phase: Mix net

sequence diagram and pseudocode for the mixing process

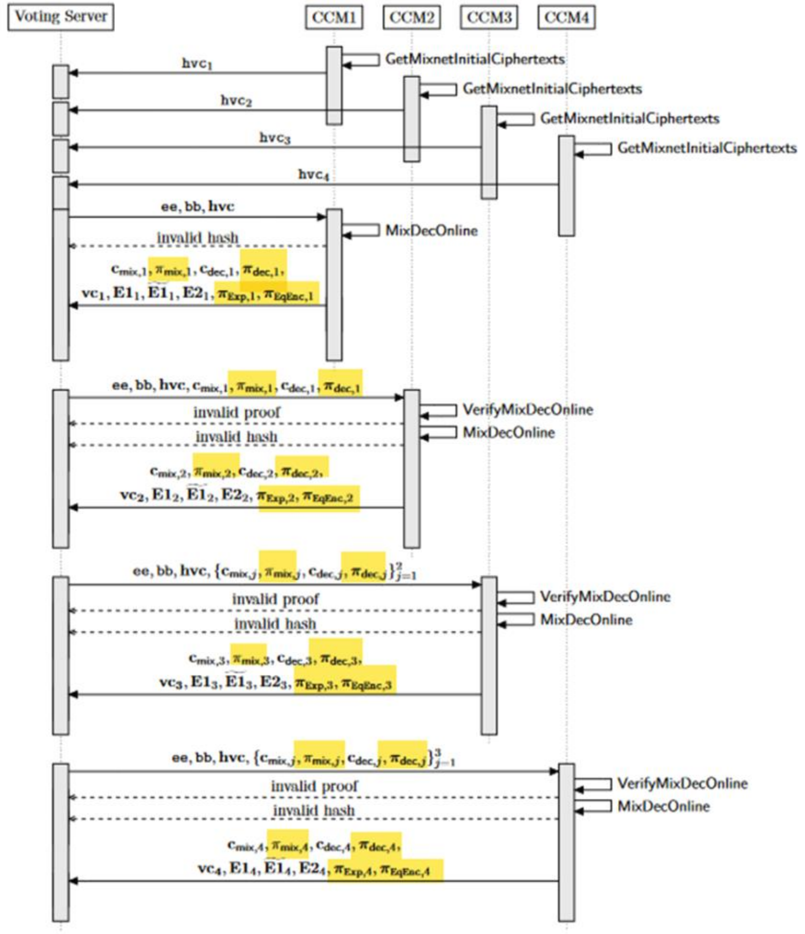


Fig. 12: Sequence diagram of the MixOnline protocol.

Swiss Post Voting System
System Specification
© 2024 Swiss Post Ltd.
Version 1.4.1

Algorithm 6.3 MixDecOnline

Context:

Group modulus $p \in \mathbb{P}$
Group cardinality $q \in \mathbb{P}$ s.t. $p = 2q + 1$
Group generator $g \in \mathbb{G}_q$
Control component index $j \in [1, 4]$
Election event ID $\mathtt{ee} \in (\mathbb{A}_{Base16})^{l_{\mathrm{ID}}}$
Ballot box ID $\mathtt{bb} \in (\mathbb{A}_{Base16})^{l_{\mathrm{ID}}}$
Number of allowed write-ins + 1 for this specific ballot box $\delta \in [1, \delta_{\mathsf{sup}}]$ ▷ Can be derived from pTable using algorithm 3.10
CCM election public keys $(\mathtt{EL}_{\mathtt{pk},1}, \mathtt{EL}_{\mathtt{pk},2}, \mathtt{EL}_{\mathtt{pk},3}, \mathtt{EL}_{\mathtt{pk},4}) \in \left(\mathbb{G}_q^{\delta_{\mathsf{max}}}\right)^4$
Electoral board public key $\mathtt{EB}_{\mathtt{pk}} \in \mathbb{G}_q^{\delta_{\mathsf{max}}}$

Stateful Lists and Maps:

List of bb of the shuffled and decrypted ballot boxes $\mathtt{L}_{\mathtt{bb},j}$

Input:

Partially decrypted votes $\mathbf{c}_{\mathsf{dec},j-1} \in (\mathbb{G}_q^{\delta+1})^{\hat{N}_c}$ ▷ CCM_1 uses $\mathbf{c}_{\mathsf{init},1}$ from internal view
CCM_j election secret key $\mathtt{EL}_{\mathtt{sk},j} \in \mathbb{Z}_q^{\delta_{\mathsf{max}}}$
CCM_j hash of the encrypted, confirmed votes $\mathtt{hvc}_j \in \mathbb{A}_{Base64}{}^{l_{\mathrm{HB64}}}$ ▷ From internal view
CCM hashes of the encrypted, confirmed votes $\mathbf{hvc} = (\mathtt{hvc}_1, \mathtt{hvc}_2, \mathtt{hvc}_3, \mathtt{hvc}_4) \in (\mathbb{A}_{Base64}{}^{l_{\mathrm{HB64}}})^4$

Require: $\mathtt{hvc}_j = \mathtt{hvc}_1 = \mathtt{hvc}_2 = \mathtt{hvc}_3 = \mathtt{hvc}_4$ ▷ The view of the initial ciphertexts must be the same for all CCs before mixing begins
Require: $\hat{N}_c \geq 2$ ▷ The algorithm runs with at least two votes
Require: $\mathtt{bb} \notin \mathtt{L}_{\mathtt{bb},j}$

Operation: ▷ For all algorithms see the crypto primitives specification

1: $\overline{\mathtt{EL}}_{\mathtt{pk}} \leftarrow \mathsf{CombinePublicKeys}\big((\mathtt{EL}_{\mathtt{pk},j}, \ldots, \mathtt{EL}_{\mathtt{pk},4}, \mathtt{EB}_{\mathtt{pk}})\big)$
2: $\mathtt{i}_{\mathsf{aux}} \leftarrow (\mathtt{ee}, \mathtt{bb}, \text{“}\mathtt{MixDecOnline}\text{”}, \mathsf{IntegerToString}(j))$
3: $(\mathbf{c}_{\mathsf{mix},j}, \pi_{\mathsf{mix},j}) \leftarrow \mathsf{GenVerifiableShuffle}(\mathbf{c}_{\mathsf{dec},j-1}, \overline{\mathtt{EL}}_{\mathtt{pk}})$
4: $(\mathbf{c}_{\mathsf{dec},j}, \pi_{\mathsf{dec},j}) \leftarrow \mathsf{GenVerifiableDecryptions}(\mathbf{c}_{\mathsf{mix},j}, (\mathtt{EL}_{\mathtt{pk},j}, \mathtt{EL}_{\mathtt{sk},j}), \mathtt{i}_{\mathsf{aux}})$
5: $\mathtt{L}_{\mathtt{bb},j} \leftarrow \mathtt{L}_{\mathtt{bb},j} \cup \mathtt{bb}$

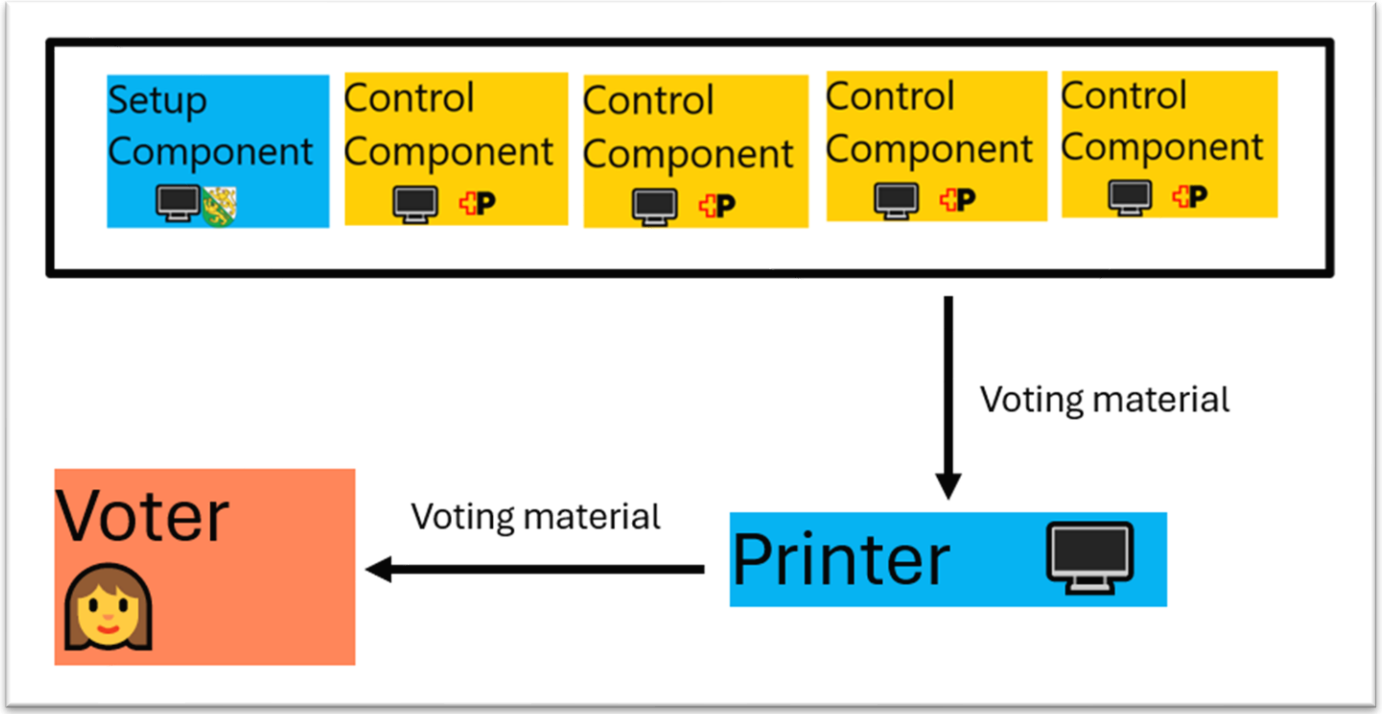
Output:

Shuffled votes $\mathbf{c}_{\mathsf{mix},j} \in (\mathbb{G}_q^{\delta+1})^{\hat{N}_c}$
Shuffle proof $\pi_{\mathsf{mix},j}$ ▷ See the domain of the shuffle argument in the crypto primitives specification
Partially decrypted votes $\mathbf{c}_{\mathsf{dec},j} \in (\mathbb{G}_q^{\delta+1})^{\hat{N}_c}$
Decryption proofs $\pi_{\mathsf{dec},j} \in (\mathbb{Z}_q \times \mathbb{Z}_q^{\delta})^{\hat{N}_c}$

Future enhanced protocol

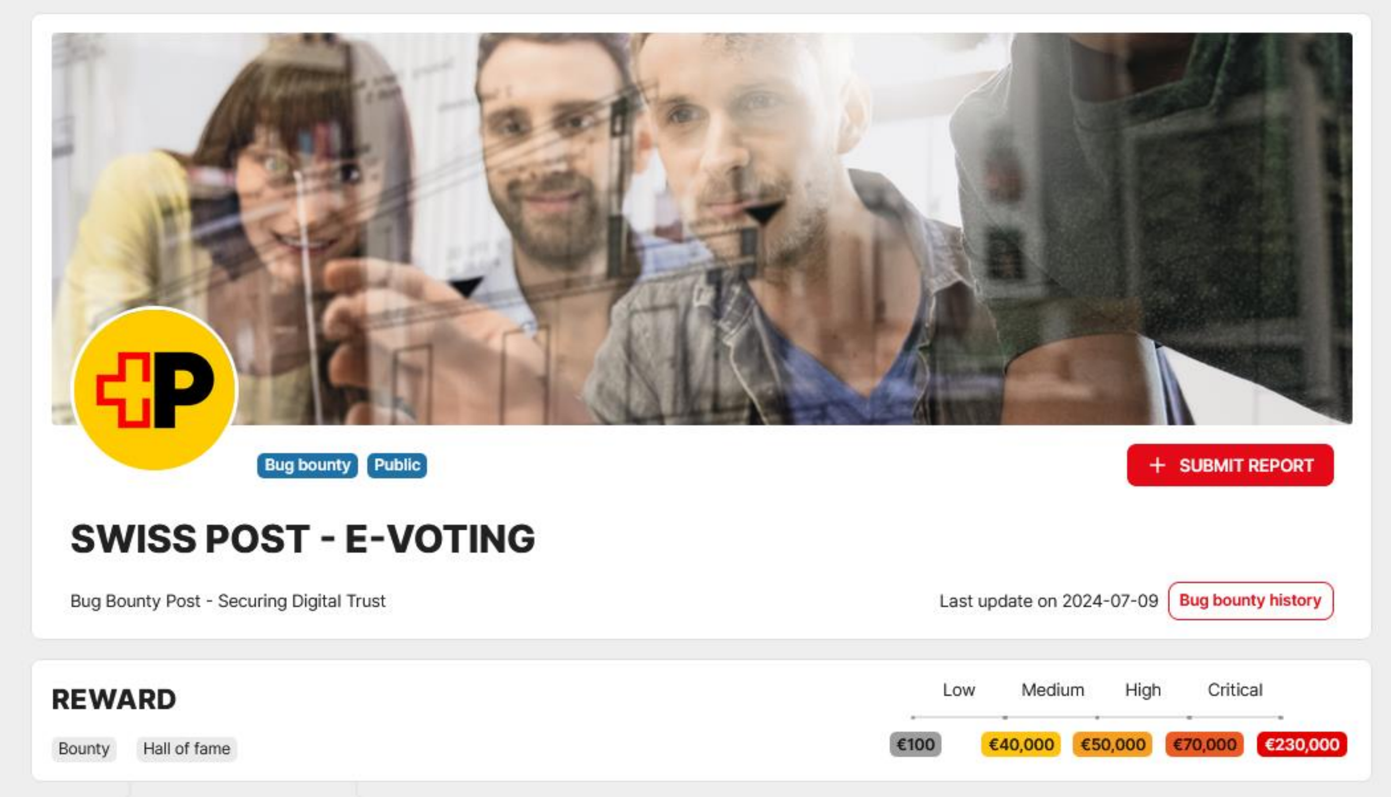
further ZK-Proofs needed in the setup phase and voting phase

- Swiss Post is working on an asymmetric distributed protocol for weakening the trust assumptions on the Setup Component;
- Currently, a trustworthy Setup Component is assumed for vote secrecy and individual verifiability;
- In the enhanced protocol, one offline and multiple online components generate the codes of the system in a distributed way;
- The enhanced protocol might include
  (additional to the primitives already present in current protocol)
  - Mix net in the setup phase
  - ZK-proof of same permutation used in two different shuffles
  - Plaintext Equality Tests (PET)



Do you want to know more about the Swiss Post Voting System?

- Find more information about the system and how to contribute on gitlab.com/swisspost-evoting;
- See also *Improving the Swiss Post Voting System: Practical Experiences from the Independent Examination and First Productive Election Event,* E-Vote-ID Conference 2023



Community programme current status (02.08.2024)

Since 2021…

- Total reports: 360
- Findings of "critical" severity: 0
- Findings of "high" severity: 5
- Total rewards paid out: € 198 450

Hans Heum

NTNU

NTNU | Norwegian University of Science and Technology

DISTRIBUTED DECRYPTION DERIVED VERIFIABLE DECRYPTION

Emil August Hovd Olaisen

August 22, 2024

Verifiable Decryption

- A system that enables a prover with the secret key to demonstrate that a ciphertext decrypts to a given message using that key

Verifiable Decryption

- A system that enables a prover with the secret key to demonstrate that a ciphertext decrypts to a given message using that key
- Showing that a message encrypts to a ciphertext is something anyone can do using the public key

Verifiable Decryption

- A system that enables a prover with the secret key to demonstrate that a ciphertext decrypts to a given message using that key
- Showing that a message encrypts to a ciphertext is something anyone can do using the public key
- We want this to be a zero-knowledge proof, it should not leak info about the secret key, nor be open to forgery

2-Party Distributed Decryption

Given a PKE with algorithms $\mathsf{KGen}, \mathsf{Enc}, \mathsf{Dec}$ we define the algorithms of 2-party distributed decryption:

The dealer algorithm ($\mathsf{Deal}(\mathsf{pk}, \mathsf{sk})$) outputs two secret key shares $\mathsf{sk}_0, \mathsf{sk}_1$ and additional auxiliary data aux

2-Party Distributed Decryption

Given a PKE with algorithms $\mathsf{KGen}, \mathsf{Enc}, \mathsf{Dec}$ we define the algorithms of 2-party distributed decryption:

The dealer algorithm ($\mathsf{Deal}(\mathsf{pk}, \mathsf{sk})$) outputs two secret key shares $\mathsf{sk}_0, \mathsf{sk}_1$ and additional auxiliary data aux

The verify algorithm ($\mathsf{Verify}(\mathsf{pk}, \mathsf{aux}, i, \mathsf{sk}_i)$) outputs either *yes* or *no*

2-Party Distributed Decryption

Given a PKE with algorithms $\mathsf{KGen}, \mathsf{Enc}, \mathsf{Dec}$ we define the algorithms of 2-party distributed decryption:

The dealer algorithm ($\mathsf{Deal}(\mathsf{pk}, \mathsf{sk})$) outputs two secret key shares $\mathsf{sk}_0, \mathsf{sk}_1$ and additional auxiliary data aux

The verify algorithm ($\mathsf{Verify}(\mathsf{pk}, \mathsf{aux}, i, \mathsf{sk}_i)$) outputs either *yes* or *no*

The player algorithm ($\mathsf{Play}(\mathsf{sk}_i, c)$) outputs a decryption share ds_i

2-Party Distributed Decryption

Given a PKE with algorithms $\mathsf{KGen}, \mathsf{Enc}, \mathsf{Dec}$ we define the algorithms of 2-party distributed decryption:

The dealer algorithm ($\mathsf{Deal}(\mathsf{pk}, \mathsf{sk})$) outputs two secret key shares $\mathsf{sk}_0, \mathsf{sk}_1$ and additional auxiliary data aux

The verify algorithm ($\mathsf{Verify}(\mathsf{pk}, \mathsf{aux}, i, \mathsf{sk}_i)$) outputs either *yes* or *no*

The player algorithm ($\mathsf{Play}(\mathsf{sk}_i, c)$) outputs a decryption share ds_i

The reconstruction algorithm ($\mathsf{Rec}(c, \mathsf{ds}_0, \mathsf{ds}_1)$) outputs either an error $\perp$ or a message m

2-Party Distributed Decryption

Given a PKE with algorithms $\mathsf{KGen}, \mathsf{Enc}, \mathsf{Dec}$ we define the algorithms of 2-party distributed decryption:

The dealer algorithm ($\mathsf{Deal}(\mathsf{pk}, \mathsf{sk})$) outputs two secret key shares $\mathsf{sk}_0, \mathsf{sk}_1$ and additional auxiliary data aux

The verify algorithm ($\mathsf{Verify}(\mathsf{pk}, \mathsf{aux}, i, \mathsf{sk}_i)$) outputs either *yes* or *no*

The player algorithm ($\mathsf{Play}(\mathsf{sk}_i, c)$) outputs a decryption share ds_i

The reconstruction algorithm ($\mathsf{Rec}(c, \mathsf{ds}_0, \mathsf{ds}_1)$) outputs either an error $\perp$ or a message m

Correctness

A distributed decryption protocol is **correct** if on input message m and pk, we have that all $(\mathsf{sk}_0, \mathsf{sk}_1, \mathsf{aux})$ generated by the dealer algorithm Deal satisfies $\mathsf{Verify}(\mathsf{pk}, \mathsf{aux}, i, \mathsf{sk}_i) = 1$ for $i = 0, 1$, and that

$$c = \mathsf{Enc}(\mathsf{pk}, m); \mathsf{Rec}(c, \mathsf{Play}(\mathsf{sk}_0, c), \mathsf{Play}(\mathsf{sk}_1, c)) = m$$

Verifiable Decryption from Distributed Decryption

How does verifiable decryption follow? Suppose we want to prove that $m = \mathsf{Dec}(c, \mathsf{sk})$

Verifiable Decryption from Distributed Decryption

How does verifiable decryption follow? Suppose we want to prove that $m = \mathsf{Dec}(c, \mathsf{sk})$

1. The prover runs Deal α times to create the key shares $\mathsf{sk}_{0,k}, \mathsf{sk}_{1,k}, \mathsf{aux}_k$ for $1 \leq k \leq \alpha$, they commit to these shares. They also generate $\mathsf{ds}_{0,j} = \mathsf{Play}(\mathsf{sk}_{0,k}, c), \mathsf{ds}_{1,k} = \mathsf{Play}(\mathsf{sk}_{1,k}, c)$ and send the commitments, decryption share and auxiliary data

Verifiable Decryption from Distributed Decryption

How does verifiable decryption follow? Suppose we want to prove that $m = \mathsf{Dec}(c, \mathsf{sk})$

1. The prover runs Deal α times to create the key shares $\mathsf{sk}_{0,k}, \mathsf{sk}_{1,k}, \mathsf{aux}_k$ for $1 \leq k \leq \alpha$, they commit to these shares. They also generate $\mathsf{ds}_{0,j} = \mathsf{Play}(\mathsf{sk}_{0,k}, c), \mathsf{ds}_{1,k} = \mathsf{Play}(\mathsf{sk}_{1,k}, c)$ and send the commitments, decryption share and auxiliary data
2. The verifier sends back a vector $\phi \in \{0, 1\}^\alpha$

Verifiable Decryption from Distributed Decryption

How does verifiable decryption follow? Suppose we want to prove that $m = \mathsf{Dec}(c, \mathsf{sk})$

1. The prover runs Deal α times to create the key shares $\mathsf{sk}_{0,k}, \mathsf{sk}_{1,k}, \mathsf{aux}_k$ for $1 \leq k \leq \alpha$, they commit to these shares. They also generate $\mathsf{ds}_{0,j} = \mathsf{Play}(\mathsf{sk}_{0,k}, c), \mathsf{ds}_{1,k} = \mathsf{Play}(\mathsf{sk}_{1,k}, c)$ and send the commitments, decryption share and auxiliary data
2. The verifier sends back a vector $\phi \in \{0,1\}^\alpha$
3. The prover sends the secret key shares $\mathsf{sk}_{\phi[k],k}$

Verifiable Decryption from Distributed Decryption

How does verifiable decryption follow? Suppose we want to prove that $m = \mathsf{Dec}(c, \mathsf{sk})$

1. The prover runs Deal α times to create the key shares $\mathsf{sk}_{0,k}, \mathsf{sk}_{1,k}, \mathsf{aux}_k$ for $1 \leq k \leq \alpha$, they commit to these shares. They also generate $\mathsf{ds}_{0,j} = \mathsf{Play}(\mathsf{sk}_{0,k}, c), \mathsf{ds}_{1,k} = \mathsf{Play}(\mathsf{sk}_{1,k}, c)$ and send the commitments, decryption share and auxiliary data
2. The verifier sends back a vector $\phi \in \{0,1\}^\alpha$
3. The prover sends the secret key shares $\mathsf{sk}_{\phi[k],k}$
4. For all $1 \leq k \leq \alpha$ the verifier checks if $\mathsf{Rec}(c, \mathsf{ds}_{0,k}, \mathsf{ds}_{1,k}) = m, \mathsf{Play}(\mathsf{sk}_{\phi[k],k}, c) = \mathsf{ds}_{\phi[k],k}$ and if $\mathsf{Verify}(\mathsf{pk}, \mathsf{aux}_k, \phi[k], \mathsf{sk}_{\phi[k],k})$ holds true

Contributions

Verifiable decryption scheme	Encryption scheme	Ciphertext size	Plaintext size	Amortized proof size
Gjøsteen et al. [1]	BGV	28.2 KB	2048 bits	$(4883/\tau + 1.8)$ MB
Our protocol Π_2	BGV	28.2 KB	2048 bits	$(2691/\tau + 32.8)$ KB
Lyubashevsky et al. [2]	Kyber-512	0.8 KB	256 bits	43.6 KB
Our protocol Π_2	$\mathrm{M-LWE}$	19.9 KB	256 bits	$(3181/\tau + 4.1)$ KB

Table: Amortized comparison between verifiable decryption schemes for $\lambda = 128$.

References

K. Gjøsteen, T. Haines, J. Müller, P. B. Rønne, and T. Silde.
Verifiable decryption in the head.
pages 355–374, 2022.

V. Lyubashevsky, N. K. Nguyen, and G. Seiler.
Shorter lattice-based zero-knowledge proofs via one-time commitments.
pages 215–241, 2021.

State of Zero-Knowledge Proofs on Mobile

Artem Grigor

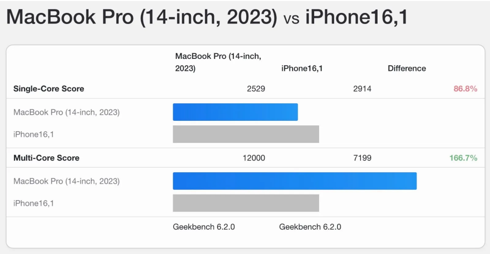
University College London (UCL)

05/09/2024

Why ZKP on Mobile?

Performance overall has increased

1. Recent Advances in ZKP Implementation
   - Software optimizations, particularly in Multi-Scalar Multiplication (MSM).
   - Better developer friendly tooling.
2. Mobile devices now rival or exceed modern PCs in power, enabling practical ZKP implementations.



Why ZKP on Mobile?

Mobile Phones are now the most used platform

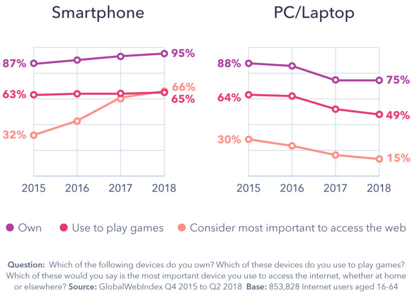


Figure: https://blog.gwi.com/trends/device-usage-2019/

ZKP in Action: Identity Verification

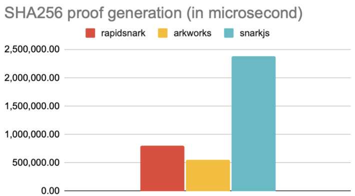
- **KYC**: Anon Aadhaar, Myna Wallet.
- **Voting**:
- **Proof of Humanity on Blockchain**: zkPassport, Proof-of-Passport.

ZKP in Action: Data Provenance

- **Proof of Funds**: Verida.
- **Proof of Payment**: zkP2P.
- **General Proof of Data/Attribute Provenance**: zkTLS, zkEmail.

Why do Native ZKP on Mobile?

- Dominance of native ZKP implementation due to better performance and security.
- Full utilization of mobile hardware, optimized resource usage, OS-level security.



Benchmark result on mobile

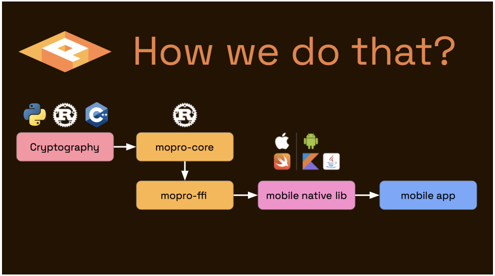
RSA circuit	witness calculation	proof generation
circom-witness-rs/ark-works	502 ms (~10x faster)	2096 ms (~6x faster)
witnesscalc/rapidsnark	228 ms (~20x faster)	2672 ms (~5x faster)
snarkjs	5440 ms	13376 ms

keccak256 circuit	witness calculation	proof generation
circom-witness-rs/ark-works	25 ms (~10x faster)	1177 ms (~10x faster)
witnesscalc/rapidsnark	161 ms (~1.7x faster)	2793 ms (~4x faster)
snarkjs	276 ms	11884 ms

Developer Ecosystem

- **Developer Tools**: *Mopro* as a framework to simplify ZKP development across platforms.

- **Technology Stack**: Mostly *Circom (R1CS + Groth16)*, considered alternatives include:
  1. *Noir DSL (Hyperplonk)*
  2. *Halo2 Rust library*



Live App Demonstration

Instructions: Scan the QR code to view the live demo or interact with the application. Focus on how the app implements ZKP efficiently on mobile.

Figure: Mopro Benchmark App link

Performance Benchmark

The benchmark results of running several circuits on iPhone 14 Pro

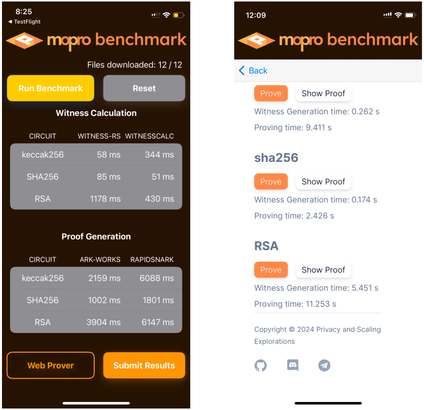


Figure: Comparison of Native and Browser Implementations

Challenges and Future Directions

Challenges:

- High computational cost of SNARKs.
- Cross-platform restrictions (e.g., iOS WebAssembly limitations).
- GPU optimization issues (e.g., MSM, I/O bottlenecks).

Future Directions:

- Exploring prover-efficient proof systems (e.g., STARKs).
- Potential of MPC and surrogate proofs.
- Further optimization and hardware acceleration.

Questions and Discussion

Thank you for your time!

Any Questions?

Soundness Labs

zkLogin: Onboarding the next billion users to web3

Mahdi Sedaghat

Jointly with Foteini Baldimitsi | Kostas Chalkias | Yan Ji | Jonas Lindstrøm | Deepak Maram | Ben Riva | Arnab Roy | Joy Wang

Foundations and Applications of Zero-Knowledge Proofs, Edinburgh, UK

There are around

100 million

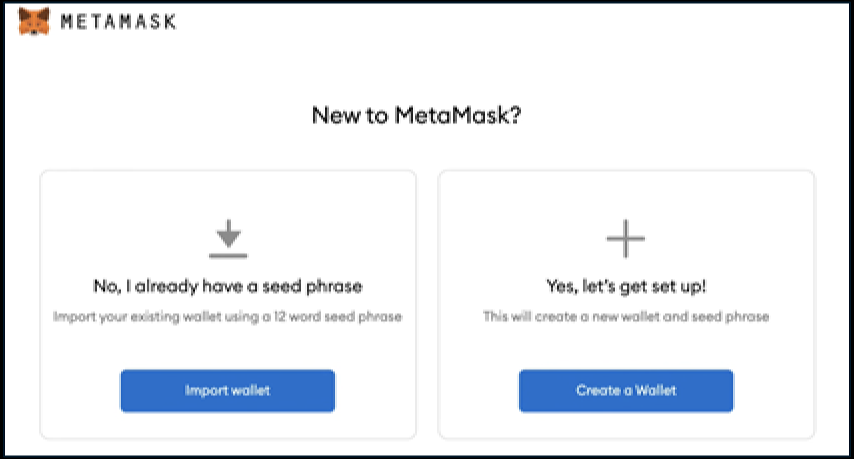
active crypto wallets

and there are several

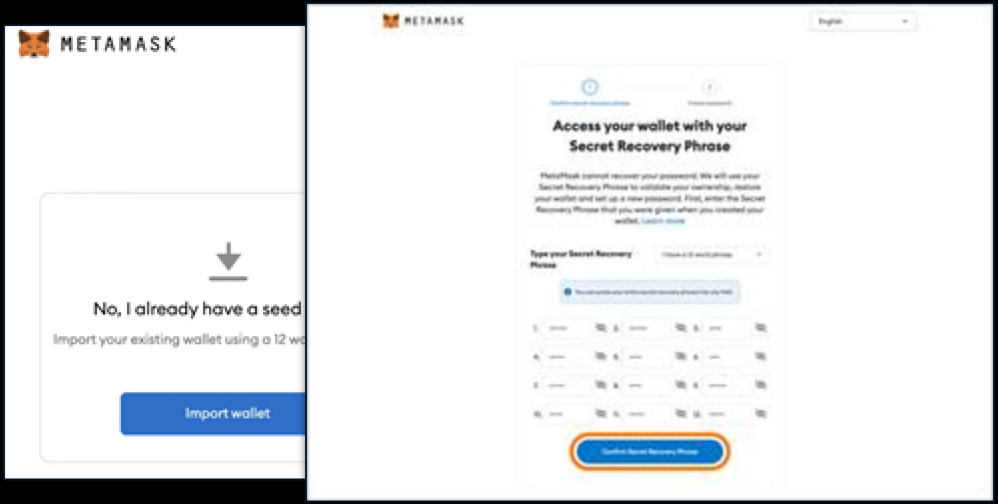
BILLIONS

of web2 accounts

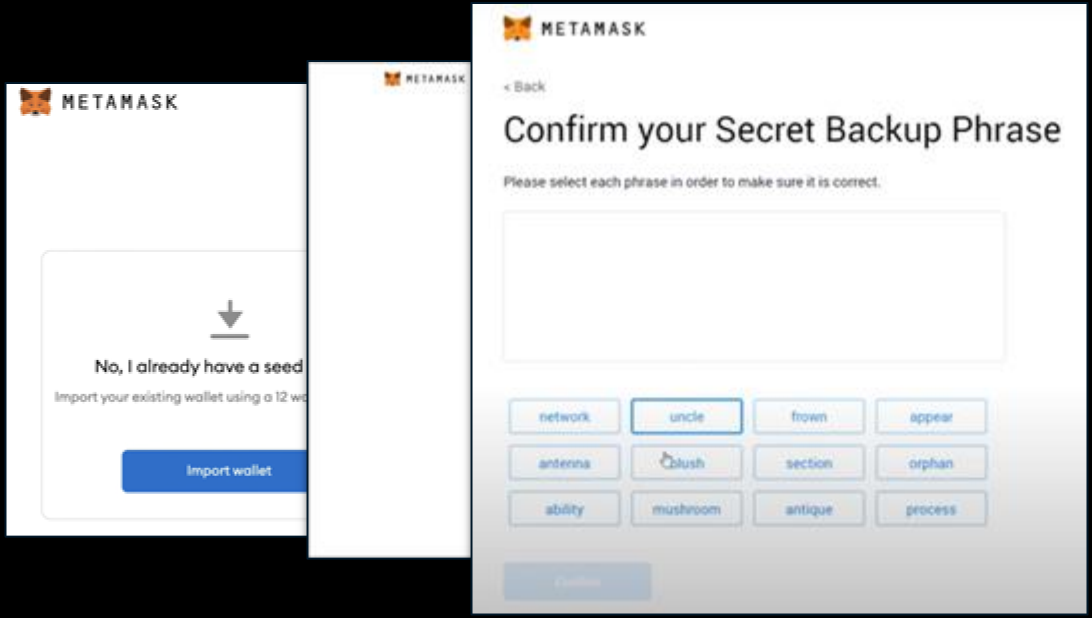
Web3 has an onboarding problem



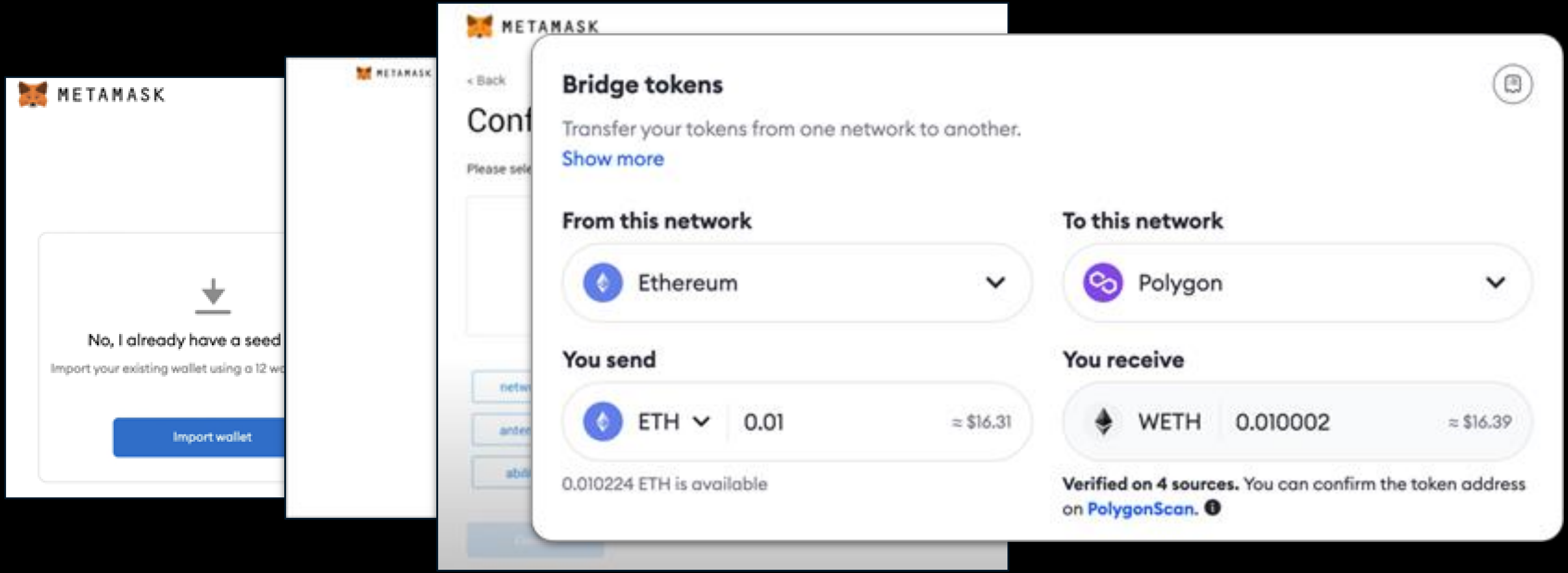
Web3 has an onboarding problem



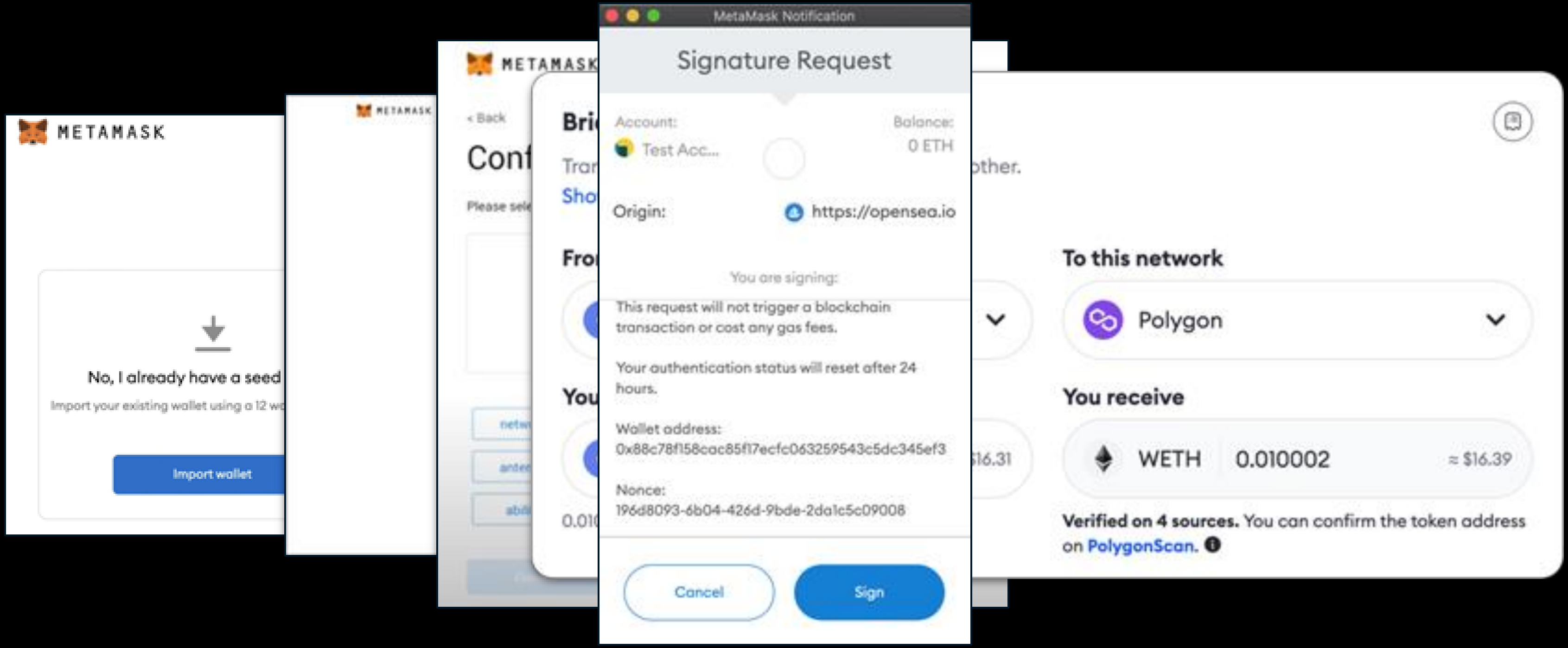
Web3 has an onboarding problem



Web3 has an onboarding problem



Web3 has an onboarding problem

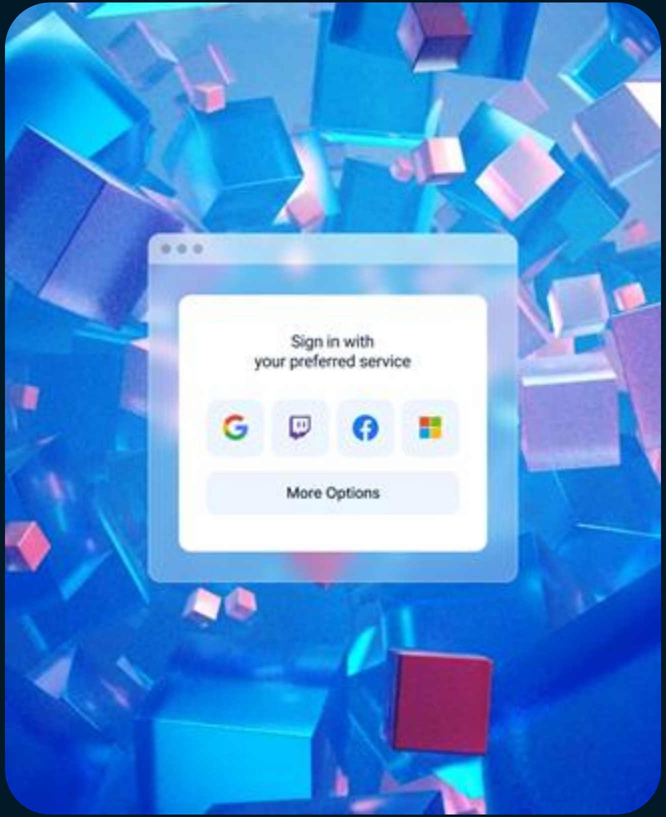


Mnemonics and keys are not going to get us mass adoption.

Complexity is the killer of adoption.
The ultimate killer dApp for blockchain, is accessibility.

Can we make it as easy as signing in with Google, Facebook and co?

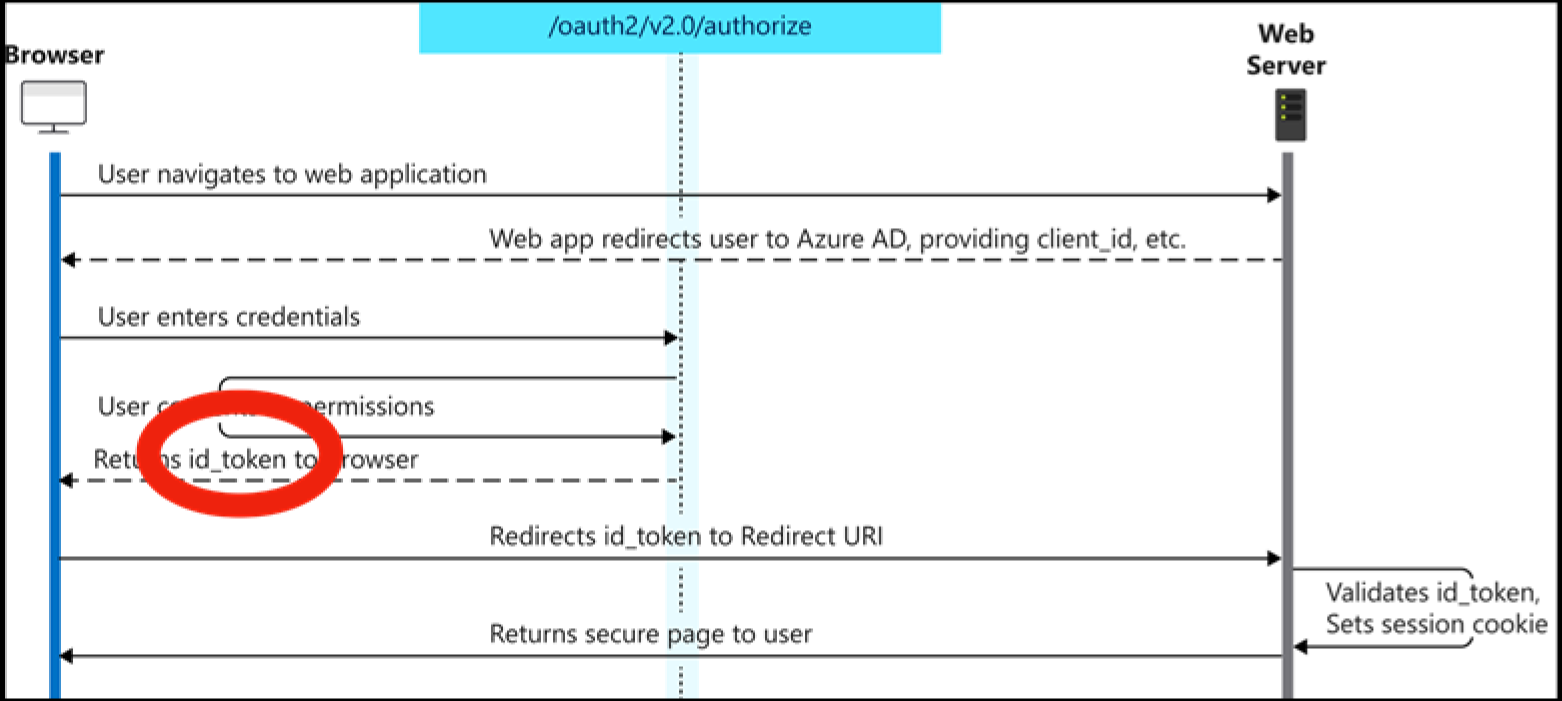
- People don’t want to use separate passwords for each and every app, each and every web2 service
- Extremely likely they already have a Google, Facebook, Amazon account
- Solution: use OAuth to leverage these already existing accounts



zkLogin:
OAuth + Zero Knowledge Proof

Non-custodial
User-friendly
Privacy-preserving

OpenID Connect (an extension of OAuth 2.0)



JWT: JSON Web Token

Base64-encoded, RSA-signed

JWT as an alternative to a private key?

Encoded PASTE A TOKEN HERE

eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiJ9.eyJzdWIiOiJwaGlsaXBwZUBwcmFnbWF0aWN3ZWJzZWN1cml0eS5jb20iLCJyb2xlIjoiYWRtaW4iLCJpc3MiOiJwcmFnbWF0aWN3ZWJzZWN1cml0eS5jb20ifQ.jW4cq__pkcq-r6H1Ebiq8toW-4Igstk1ibRgxECUhdExvZTzhvXqfrPewgtRHEApBWXpUqGqRY6LSj2Gklxt3O6kxUaky-VTl8jbL0OV5HEQVOnL3VVgPv65ddGRYaCOuyzYcf6M1fA4PeFme9lL2ZTNtjiE0OJjUR3LH1Dptm_u9_aQRtJ_IU8xiywctV1JLeQcMJFDXCS2N5oU0Gkatu oJNbjMdSTg3BsU5yUsGLyuPnJTeUWJajin5e0NuBA1Bc6oLee6KtPAM8-1ufhHr1fpT78iGyrSQLpiVd2naPA0CvUyZ6W_4arnmZDKRF9N9zOR_Jxyfv5xFMi4G67EhA

Decoded EDIT THE PAYLOAD AND SECRET

HEADER: ALGORITHM & TOKEN TYPE

```
{
  "typ": "JWT",
  "alg": "RS256"
}
```

PAYLOAD: DATA

```
  "sub": "philippe@pragmaticwebsecurity.com",
  "role": "admin",
  "iss": "pragmaticwebsecurity.com"
}
```

VERIFY SIGNATURE

```
RSASHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),
   Lg8ulqDgdXLFwS/lHXV/QKcBOXBrIp
   B4DWQ0c16zLZU7NTe657rWqKlwIDAQ
   AB
   -----END RSA PUBLIC KEY-----
```

A Google-issued JWT (decoded)

```
{
  "alg": "RS256",
  "kid": "96971808796829a972e79a9d1a9fff11cd61b1e3",
  "typ": "JWT"
}
```

Sign in with Google

```
{
  "iss": "https://accounts.google.com",
  "azp": "575519204237-msop9ep45u2uo98hapqmngv8d84qdc8k.apps.googleusercontent.com",
  "aud": "575519204237-msop9ep45u2uo98hapqmngv8d84qdc8k.apps.googleusercontent.com",
  "sub": "1104634521            ",
  "nonce": "16637918813908060261870528903994038721669799613803601616678155512181273289477",
  "iat": 1682002642,
  "exp": 1682006242,
  "jti": "a8a0728a3ffd5dc81ecfd0ea81d0d33d803eb830"
}
```

you can ask for email
and other personal info

zkLogin tricks

sample openID JWT token
signed by Google / FB

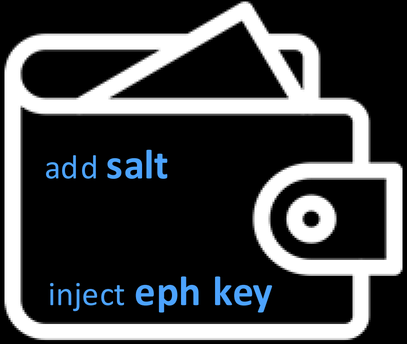
```
{
  "iss": "https://accounts.google.com",
  "azp": "575519204237-msop9ep45u2uo98hapqmngv8d84qdc8k.apps.googleusercontent.com",
  "aud": "575519204237-msop9ep45u2uo98hapqmngv8d84qdc8k.apps.googleusercontent.com",
  "sub": "1104634521          ",
  "nonce": "16637918813908060261870528903994038721669799613803601616678155512181273289477",
  "iat": 1682002642,
  "exp": 1682006242,
  "jti": "a8a0728a3ffd5dc81ecfd0ea81d0d33d803eb830"
}
```

aud = walletID
sub = userID

we could ask for email too

nonce = eph. pubKey + expiration

+



+ **ZK proof** =

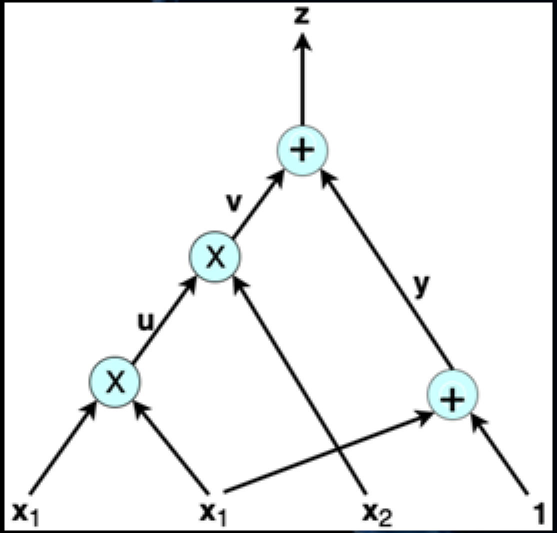
ADDRESS
~hash(providerID + zkhash(walletID + userID + zkhash(salt)))

&

verify ZKproof + **verify eph key sig**

Circuit details

- Implemented in circom: ~1M R1CS constraints
- Key operations
    - SHA-2 (66%)
    - RSA signature verification (14%) using tricks from [KPS18]
    - JSON parsing, Poseidon hashing, Base64, extra rules (20%)
- Prover based on rapidsnark
    - C++ and Assembly based



zkLogin latency

These numbers correspond only to the **first transaction of a session**

Operation	zkLogin	Ed25519
Fetch salt from salt service	0.2 s	NA
Fetch ZKP from ZK service	2.78 s	NA
Signature verification	2.04 ms	56.3 μs
E2E transaction confirmation	3.52 s	120.74 ms

Latency for most zkLogin transactions is **very similar** to traditional ones!

MystenLabs Sui COSIC Soundness Labs

Q & A

ZK for authentication

How to SNARK sign-in with Google, Apple & FB

Paper

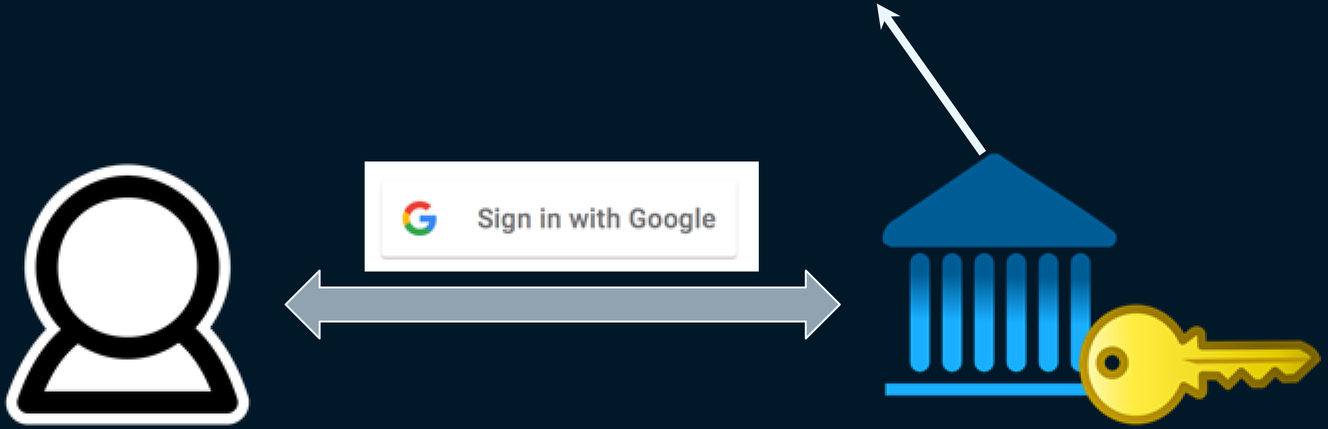
Sui docs

Demo

Contact: mahdi@soundness.xyz

Backup slides

Naive solution: OAuth + Custodian



Can we avoid the trusted custodian?

zkLogin goodies

Native auth, cheap

Not via smart contracts, same gas cost as regular sig verification.

ID-based wallets

Create email or phone number based accounts.

Can also reveal identity of an existing account (e.g., email) fully or partially (e.g., reveal a suffix like @xyz.edu)

Embedded wallet

Mobile apps or websites can natively integrate zkLogin without the need for a wallet popup!

2FA

Can do a 2-out-of-3 between Google, Facebook and Apple. Salt can also serve as a second factor.

Hard to lose!

Thanks to robust recovery paths of Google, Facebook.

ADDRESS

hash(providerID + zkhash(walletID + userID + zkhash(salt)))

+ ZK proof

Sui

zkLogin

single-click accounts w/

- Google
- Facebook
- Twitch
- Apple
- Slack
- Microsoft

native authenticator

non-custodial

*discoverable, claimable

invisible wallets

semi-portable, 2FA

Challenge 1: How to authorize a tx with a JWT?


```
{
  "alg": "RS256",
  "kid": "96971808796829a972e79a9d1a9fff11cd61b1e3",
  "typ": "JWT"
}
```

```
{
  "iss": "https://accounts.google.com",
  "azp": "575519204237-msop9ep45u2uo98hapqmngv8d84qdc8k.apps.googleusercontent.com",
  "aud": "575519204237-msop9ep45u2uo98hapqmngv8d84qdc8k.apps.googleusercontent.com",
  "sub": "1104634521              ",
  "nonce": "16637918813908060261870528903994038721669799613803601616678155512181273289477",
  "iat": 1682002642,
  "exp": 1682006242,
  "jti": "a8a0728a3ffd5dc81ecfd0ea81d0d33d803eb830"
}
```

Inject a fresh pub key into JWT!


```
{
  "alg": "RS256",
  "kid": "96971808796829a972e79a9d1a9fff11cd61b1e3",
  "typ": "JWT"
}

{
  "iss": "https://accounts.google.com",
  "azp": "57519204237-msop9ep45u2uo98hapqmngv8d84qdc8k.ap…ogleusercontent.com",
  "aud": "57519204237-msop9ep45u2uo98hapqmngv8d84qdc8k.a…s.googleusercontent.com",
  "sub": "1104634521          ",
 >"nonce": "16637918813908060261870528903994038721669799613803601616678155512181273289477",
  "iat": 1682002642,
  "exp": 1682006242,
  "jti": "a8a0728a3ffd5dc81ecfd0ea81d0d33d803eb830"
}
```


replace ***nonce*** with
user provided data:

ephemeral pub key +
expiration

We have a DIGITAL CERT over our fresh key + expiration

Challenge 2: How to identify the user without linking identities?


```
{
  "iss": "https://accounts.google.com",
  "azp": "575519204237-msop9ep45u2uo98hapqmngv8d84qdc8k.apps.googleusercontent.com",
  "aud": "575519204237-msop9ep45u2uo98hapqmngv8d84qdc8k.apps.googleusercontent.com",
  "sub": "1104634521          ",
  "nonce": "16637918813908060261870528903994038721669799613803601616678155512181273289477",
  "iat": 1682002642,
  "exp": 1682006242,
  "jti": "a8a0728a3ffd5dc81ecfd0ea81d0d33d803eb830"
}
```


aud = walletID
sub = userID

we could ask for email too

ADDRESS
???

Add a persistent randomizer: salt


```
{
  "iss": "https://accounts.google.com",
  "azp": "575519204237-msop9ep45u2uo98hapqmngv8d84qdc8k.apps.googleusercontent.com",
  "aud": "575519204237-msop9ep45u2uo98hapqmngv8d84qdc8k.apps.googleusercontent.com",
  "sub": "1104634521",
  "nonce": "16637918813908060261870528903994038721669799613803601616678155512181273289477",
  "iat": 1682002642,
  "exp": 1682006242,
  "jti": "a8a0728a3ffd5dc81ecfd0ea81d0d33d803eb830"
}
```


aud = walletID
sub = userID

we could ask for email too

ADDRESS
hash(providerID + walletID + userID + salt)

Salt: A persistent per-user secret for **unlinkability**

Who maintains the salt?

- Client-side on-device management
  - Edge cases, e.g., cross-device sync, device loss need handling

- Server-side management by a “salt service”
  - Each wallet can maintain their own service / delegate it
  - Privacy models: Store salt either in TEE / MPC / plaintext
  - Auth policies to the service: Either JWT or 2FA

ADDRESS

hash(providerID + walletID + userID + salt)

Salt: A persistent per-user secret for **unlinkability**

Challenge 3: How to hide the JWT? SNARKs to the rescue!


```
{
  "iss": "https://accounts.google.com",
  "azp": "575519204237-msop9ep45u2uo98hapqmngv8d84qdc8k.apps.googleusercontent.com",
  "aud": "575519204237-msop9ep45u2uo98hapqmngv8d84qdc8k.apps.googleusercontent.com",
  "sub": "1104634521",
  "nonce": "16637918813908060261870528903994038721669799613803601616678155512181273289477",
  "iat": 1682002642,
  "exp": 1682006242,
  "jti": "a8a0728a3ffd5dc81ecfd0ea81d0d33d803eb830"
}
```


aud = walletID
sub = userID

we could ask for email too

nonce = eph. pubKey + expiration

Goal: Prove you have a valid JWT + you know the salt + you injected the ephemeral key into JWT

- Verify JWT's signature using Google's public key
- Verify the ephemeral public key is injected into the JWT's nonce
- Verify that the address is derived correctly from the JWT's userID, walletID, providerID + user's salt

Yellow => private inputs
Blue => public inputs

Challenge 4: Prove + RTT in <3s

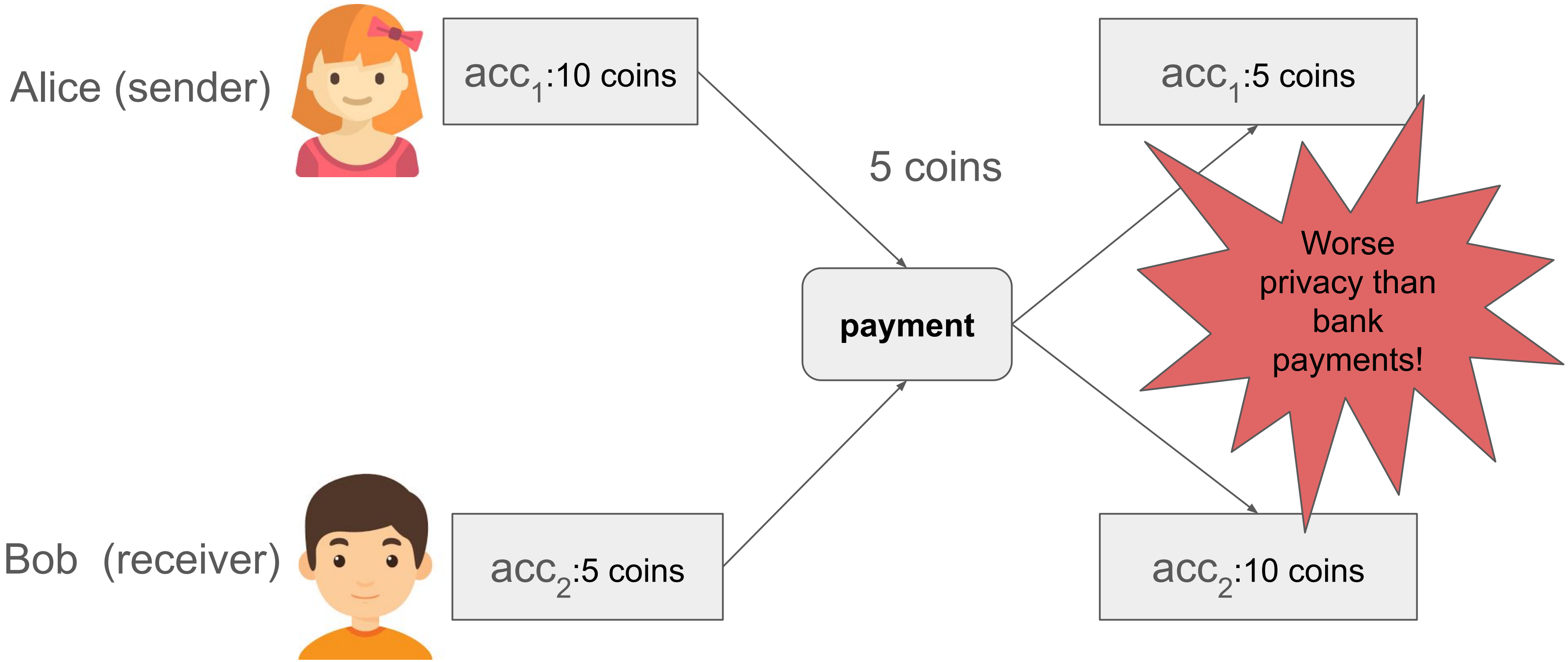
- We chose Groth16 due to its small proofs + rich ecosystem + fast prover
- But.. proofs are slow to generate on end-user devices
    - Make ZKP efficient: Hand-optimized circuit that selectively parses relevant parts of the JWT + string slicing tricks + …
    - Delegate proving to an untrusted ZKP service
        - Open problem: How to delegate with privacy?

Nopenena Untraceable Payments

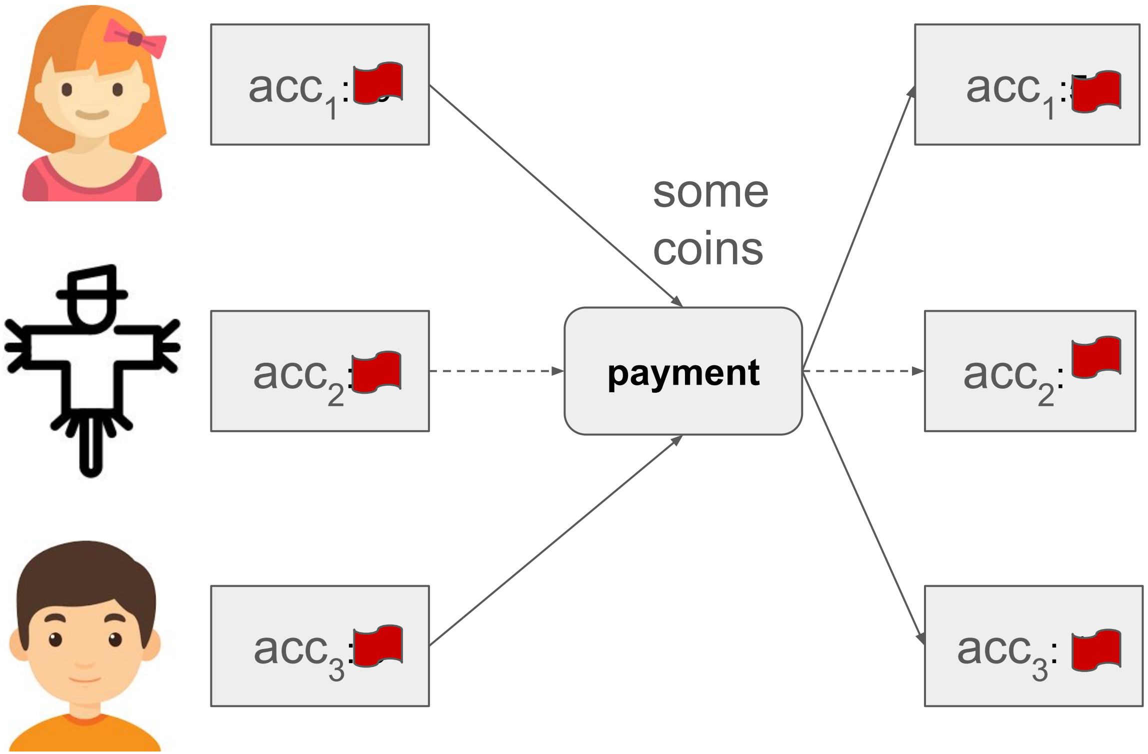
Defeating Graph Analysis with Small Decoy Sets

Jayamine Alupotha, Mathieu Gestin, and Christian Cachin

Classic Decentralized Payments



Decoy-based Confidential Payments



An example of account-based transactions

Confidentiality ✓

Untraceability ✓

Sender-Anonymity ✓

Bob only learns that Alice owns either acc_1 or acc_2.

Full decoy-sets vs. User-defined decoy-sets

Full decoy-set payments

Examples: Zerocoin, Zerocash, ZCash, Lelantus, and BlockMaze

Maximal untraceability

Higher transaction expiration, trusted setups, and high computational cost

User-defined decoy-set payments

Examples: Monero, RingCTv2, RingCTv3, Anonymous Zether, and QuisQuis

Better performance without trusted setups and no transaction expiration (!)

Untraceability within small sets (may be degrading)

Non-degrading Untraceability

- Monero, Ring CT v2, and Ring CT v3 (UTXO)
- Limited to an epoch: Anonymous Zether [1][2] and PriDe CT (Accounts)
- QuisQuis

[1] Bünz, Benedikt, et al. "Zether: Towards privacy in a smart contract world." *International Conference on Financial Cryptography and Data Security*. Cham: Springer International Publishing, 2020.

[2] Diamond, Benjamin E. "Many-out-of-many proofs and applications to anonymous zether." *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2021.

QuisQuis

- Non-degrading untraceability with small-decoy sets.
- Large cryptographic data for validity
- No “zero-knowledge contracts”

[3] Fauzi, Prastudy, et al. "Quisquis: A new design for anonymous cryptocurrencies." *Advances in Cryptology–ASIACRYPT 2019: 25th International Conference on the Theory and Application of Cryptology and Information Security, Kobe, Japan, December 8–12, 2019, Proceedings, Part I 25*. Springer International Publishing, 2019.

Nopenena (“cannot see”)

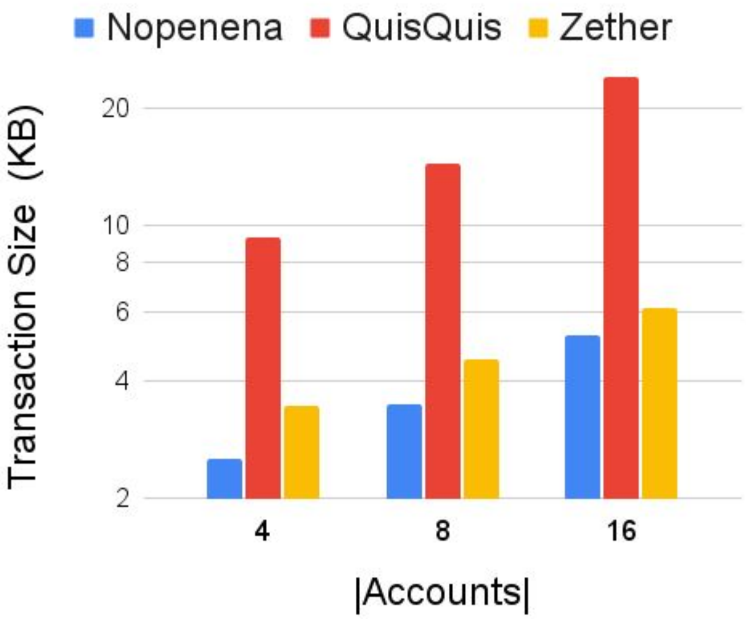
- Non-degrading untraceability with small-decoy sets.
- Zero-knowledge contract compatibility
- ~80% smaller than QuisQuis

How do we reduce transaction sizes and verification times?

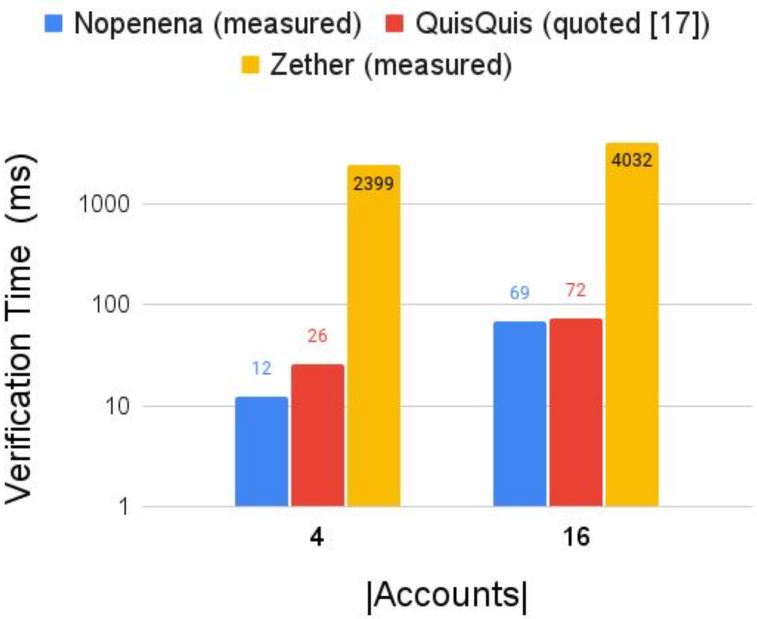
By replacing the entire cryptographic protocol!

https://eprint.iacr.org/2024/903

Performance Comparison



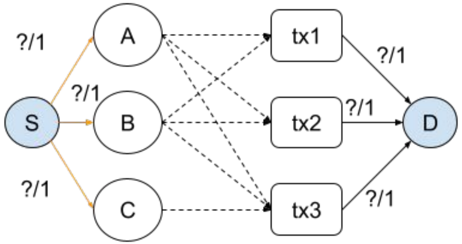
An apple-to-apple comparison!



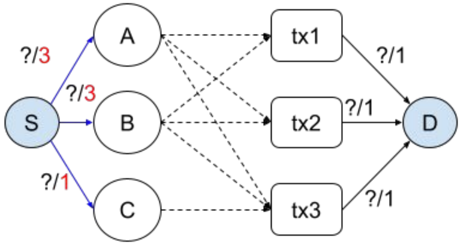
Not an apple-to-apple comparison!

Maximal Matching Problem

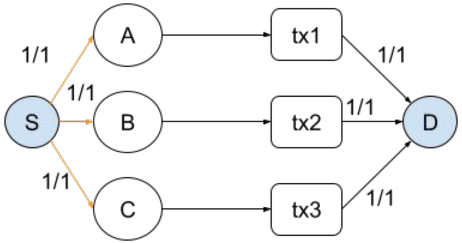
tx1: (A, B)
tx2: (A, B)
tx3: (A, B, C)



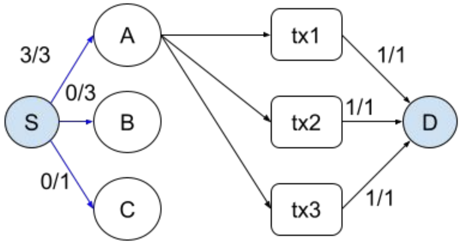
Maximal Matching Problem in Monero/Zether



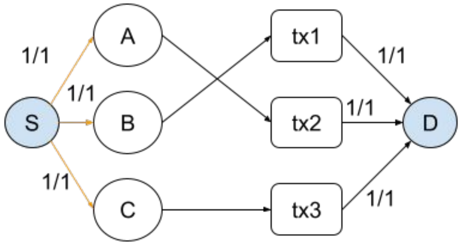
Maximal Matching Problem in Nopenena and QuisQuis



First solution for the problem in Monero/Zether



One of the possible solutions in Nopenena/QuisQuis that is impossible in Zether



Second solution for the problem in Monero/Zether

Each → has capacity of 1   Each → has capacity of 1 or 3

Each - - - → is a potential link between parties and transaction of capacity 1.

Protocol	Untraceable	Confidential	Expiring Probability	No Trusted Setup	DoS Attack Resistance	Graph Analysis Resistance	Non-monotonic Set of Assets	Contract Support
Zerocoin [42]	Maximal	○	High	○	○	●	●	○
ZCash [29]	Maximal	●	High	○	○	●	○	○
Lelantus [31]	Maximal	●	High	●	○	●	○	○
Mimblewimble [30]	No	●	Zero	●	-	-	●	○
Monero [46], [34, 60]	Degrading	●	Zero	●	●	○	○	○
Ring CT v.2 [54]	Degrading	●	Zero	○	●	○	○	○
Zether [9, 15]	Degrading (epoch)	●	High	●	○	◑	●	●
QuisQuis [19]	Non-degrading	●	Low	●	○	●	●	○
PriDe CT [26]	Degrading (epoch)	●	High	●	○	◑	●	●
PriFHEte [39]	Maximal	●	High	●	○	●	●	○
Nopenena (this paper)	Non-degrading	●	Low	●	●	●	●	●

Table 1: A Comparison of Related Work. Here, expiring probability means the probability of a transaction expiring due to epochs or updated assets. We use ◑ to denote DM-decomposition limited to epochs.

RI CODE
Research Institute
Cyber Defence
Universität der Bundeswehr München

A CRACK IN THE FIRMAMENT

Restoring Soundness of the Orion Proof System

Thomas den Hollander, Daniel Slamanig

05-09-2024

THE ORION ZK-SNARK [XZS22]

- zk-SNARK based on Brakedown

THE ORION ZK-SNARK [XZS22]

- zk-SNARK based on Brakedown
- Outer SNARK to prove Brakedown relation
  - $O(N)$ prover time overall
  - $\log^2(N)$ proof size and verifier time

THE ORION POLYNOMIAL COMMITMENT

- Polynomial as $n \times n$ matrix, encode rows, commit.

$$\begin{bmatrix} \mathrm{Enc}(x_{11}, \ldots, x_{1n}) \\ \mathrm{Enc}(x_{21}, \ldots, x_{2n}) \\ \ldots \\ \mathrm{Enc}(x_{n1}, \ldots, x_{nn}) \end{bmatrix}$$

THE ORION POLYNOMIAL COMMITMENT

- Polynomial as $n \times n$ matrix, encode rows, commit.
- Take random linear combination of rows.

$$\begin{bmatrix}\gamma_1 & \gamma_2 \dots \gamma_n\end{bmatrix}$$
$$\cdot$$
$$\begin{bmatrix}\mathrm{Enc}(x_{11}, \dots, x_{1n}) \\ \mathrm{Enc}(x_{21}, \dots, x_{2n}) \\ \dots \\ \mathrm{Enc}(x_{n1}, \dots, x_{nn})\end{bmatrix}$$
$$=$$
$$\begin{bmatrix}c_1 & \dots & c_n\end{bmatrix}$$

THE ORION POLYNOMIAL COMMITMENT

- Polynomial as $n \times n$ matrix, encode rows, commit.
- Take random linear combination of rows.
- Check
  1. $\vec{c} = \text{Enc}(y)$ for some y, inside outer SNARK.

$$\begin{bmatrix}\gamma_1 & \gamma_2 \dots \gamma_n\end{bmatrix}$$
$$\cdot$$
$$\begin{bmatrix}\text{Enc}(x_{11}, \dots, x_{1n}) \\ \text{Enc}(x_{21}, \dots, x_{2n}) \\ \dots \\ \text{Enc}(x_{n1}, \dots, x_{nn})\end{bmatrix}$$
$$=$$
$$\begin{bmatrix}c_1 & \dots & c_n\end{bmatrix}$$

THE ORION POLYNOMIAL COMMITMENT

- Polynomial as $n \times n$ matrix, encode rows, commit.
- Take random linear combination of rows.
- Check
    1. $\vec{c} = \text{Enc}(y)$ for some y, inside outer SNARK.
    2. Linear combination of commitment, for some columns at random.

$$\begin{bmatrix}\gamma_1 & \gamma_2 \dots \gamma_n\end{bmatrix}$$

$$\cdot$$

$$\begin{bmatrix}\text{Enc}(x_{11}, \dots, x_{1n}) \\ \text{Enc}(x_{21}, \dots, x_{2n}) \\ \dots \\ \text{Enc}(x_{n1}, \dots, x_{nn})\end{bmatrix}$$

$$=$$

$$\begin{bmatrix}c_1 & \dots & c_n\end{bmatrix}$$

THE ISSUE

- We need to commit to columns to check linear combination.

THE ISSUE

- We need to commit to columns to check linear combination.
- Succinctness $\Rightarrow$ Only commit to selected columns in outer SNARK
    - Or, non-linear prover time

THE ISSUE

- We need to commit to columns to check linear combination.
- Succinctness $\Rightarrow$ Only commit to selected columns in outer SNARK
  - Or, non-linear prover time
- Soundness $\Rightarrow$ Select columns at random after committing
  - Or, we can cheat by picking y s.t. $\mathsf{Enc}(y) = \vec{c}$ only at known columns!

THE ISSUE

- We need to commit to columns to check linear combination.
- Succinctness $\Rightarrow$ Only commit to selected columns in outer SNARK
  - Or, non-linear prover time
- Soundness $\Rightarrow$ Select columns at random after committing
  - Or, we can cheat by picking y s.t. $\mathsf{Enc}(y) = \vec{c}$ only at known columns!
- Our solution: use two different column sets

THE ISSUE

- We need to commit to columns to check linear combination.
- Succinctness $\Rightarrow$ Only commit to selected columns in outer SNARK
  - Or, non-linear prover time
- Soundness $\Rightarrow$ Select columns at random after committing
  - Or, we can cheat by picking y s.t. $\mathsf{Enc}(y) = \vec{c}$ only at known columns!
- Our solution: use two different column sets
- Breaks zero-knowledge: new randomization

THE ISSUE

- We need to commit to columns to check linear combination.
- Succinctness $\Rightarrow$ Only commit to selected columns in outer SNARK
  - Or, non-linear prover time
- Soundness $\Rightarrow$ Select columns at random after committing
  - Or, we can cheat by picking y s.t. $\mathsf{Enc}(y) = \vec{c}$ only at known columns!
- Our solution: use two different column sets
- Breaks zero-knowledge: new randomization
- Other improvements: better efficiency, fix simulator, ...

THANK YOU FOR LISTENING!

[HS] Thomas den Hollander and Daniel Slamanig. *A Crack in the Firmament: Restoring Soundness of the Orion Proof System and More*. URL: https://eprint.iacr.org/2024/1164.

[XZS22] Tiancheng Xie, Yupeng Zhang, and Dawn Song. "Orion: Zero Knowledge Proof with Linear Prover Time". In: *CRYPTO 2022, Part IV*. Ed. by Yevgeniy Dodis and Thomas Shrimpton. Vol. 13510. LNCS. Springer, Cham, Aug. 2022, pp. 299–328. DOI: 10.1007/978-3-031-15985-5_11.

Thomas den Hollander
Quantum Safe & Advanced Cryptography
Research Institute CODE
Universität der Bundeswehr München

thomasdh@unibw.de
https://www.unibw.de/crypto

Research Institute Cyber Defence
Universität der Bundeswehr München