Cryptographic Vulnerabilities in Threshold Wallets

Omer Shlomovits

KZen Research

Outline

This talk is focused on the pitfalls of using threshold ECDSA in building new generation of SW wallets.

😇 Yay, Threshold Wallet!

😈 What can go wrong?

💩 I don't care. I want it!

A Wallet

- Client software used as a gateway and means of interaction with a blockchain

- Among other responsibilities, the Client software must play a critical cryptographic role of generating **digital signatures**

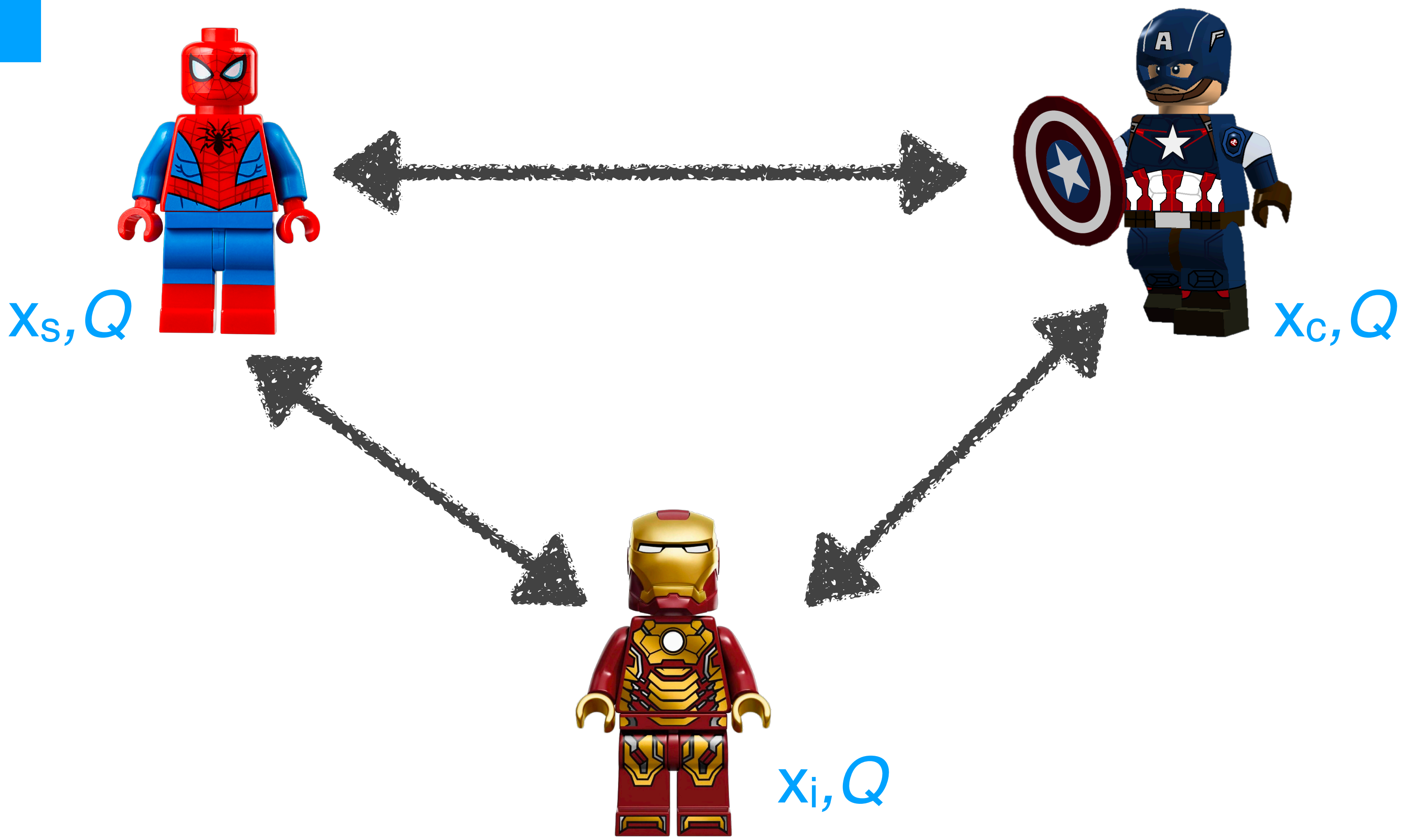
Threshold Signatures

(t,n)-threshold signature scheme distributes signing power to n parties such that any group of at least t parties can generate a signature

Threshold Signatures

(t,n)-threshold signature scheme distributes signing power to n parties such that any group of at least t parties can generate a signature

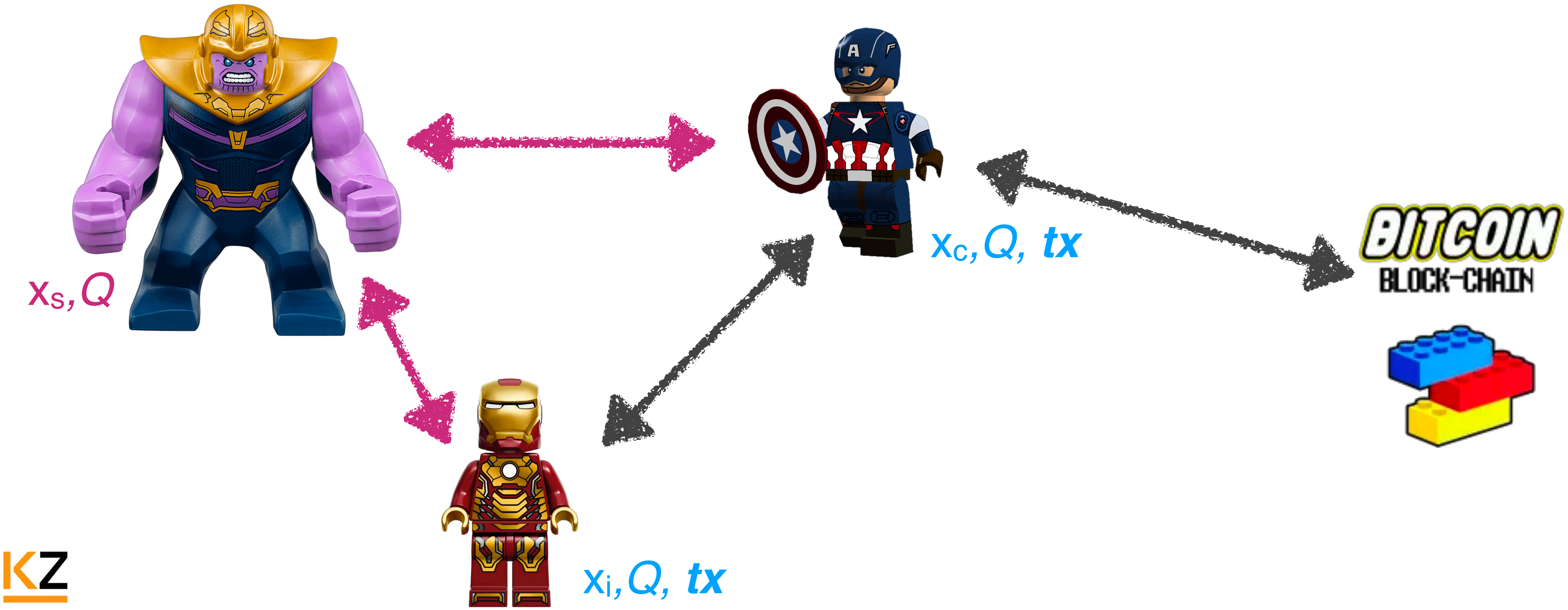
(2,3) - Keygen



Threshold Signatures

(t,n)-threshold signature scheme distributes signing power to n parties such that any group of at least t parties can generate a signature

(2,3) - Signing



Threshold ECDSA

https://github.com/KZen-networks/multi-party-ecdsa

	Assumptions	**KeyGen**	**Sign**
[L17]	ECDSA, Paillier	Seconds	milliseconds
[GG18]	ECDSA, Strong RSA	milliseconds	milliseconds
[DKLS18]	ECDSA	milliseconds	milliseconds
[LNR18]	ECDSA, DDH	Seconds	milliseconds

Threshold ECDSA

https://github.com/KZen-networks/multi-party-ecdsa

	Assumptions	**KeyGen**	**Sign**
[L17]	ECDSA, Paillier	Seconds	milliseconds
[GG18]	ECDSA, Strong RSA	milliseconds	milliseconds
[DKLS18]	ECDSA	milliseconds	milliseconds
[LNR18]	ECDSA, DDH	Seconds	milliseconds

- Annoyingly they all came out at the same time so none contains a comparison to the others

Threshold ECDSA

https://github.com/KZen-networks/multi-party-ecdsa

	Assumptions	KeyGen	Sign
[L17]	ECDSA, Paillier	Seconds	milliseconds
[GG18]	ECDSA, Strong RSA	milliseconds	milliseconds
[DKLS18]	ECDSA	milliseconds	milliseconds
[LNR18]	ECDSA, DDH	Seconds	milliseconds

- Annoyingly they all came out at the same time so none contains a comparison to the others
- Does efficient threshold signing ➡️ efficient threshold wallet ?

Threshold Wallet

Described by the following tuple of 5 algorithms

- Distributed key generation (DKG)
- Distributed Signing
- Secret Share Recovery
- Deterministic Child Address Derivation
- Rotation

Ref: https://github.com/KZen-networks/gotham-city

ThresholdSig > MultiSig?

A MultiSig is an Emulation of ThresholdSig

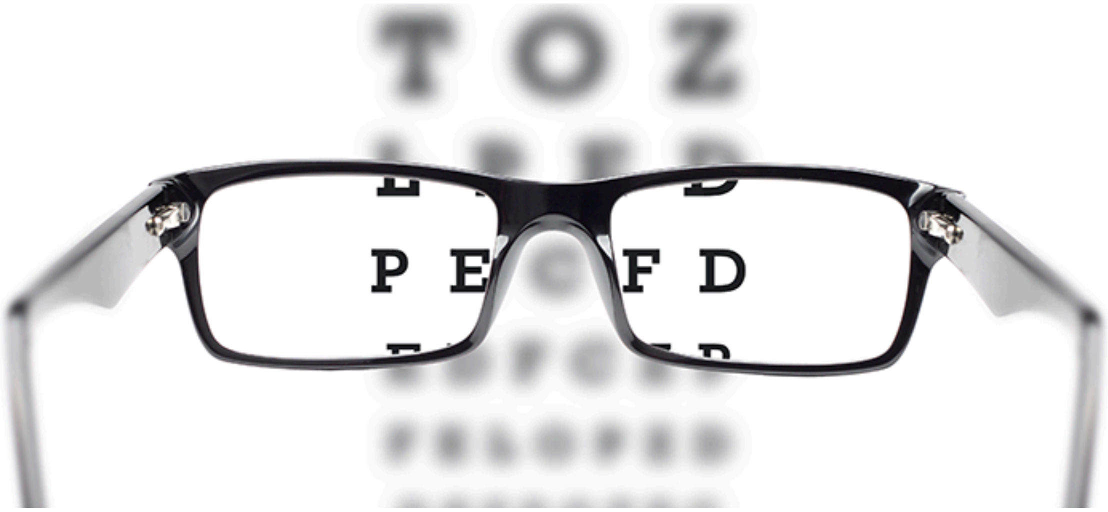
- Access policy privacy
- Secret refreshment
- Low cost
- Max number of parties

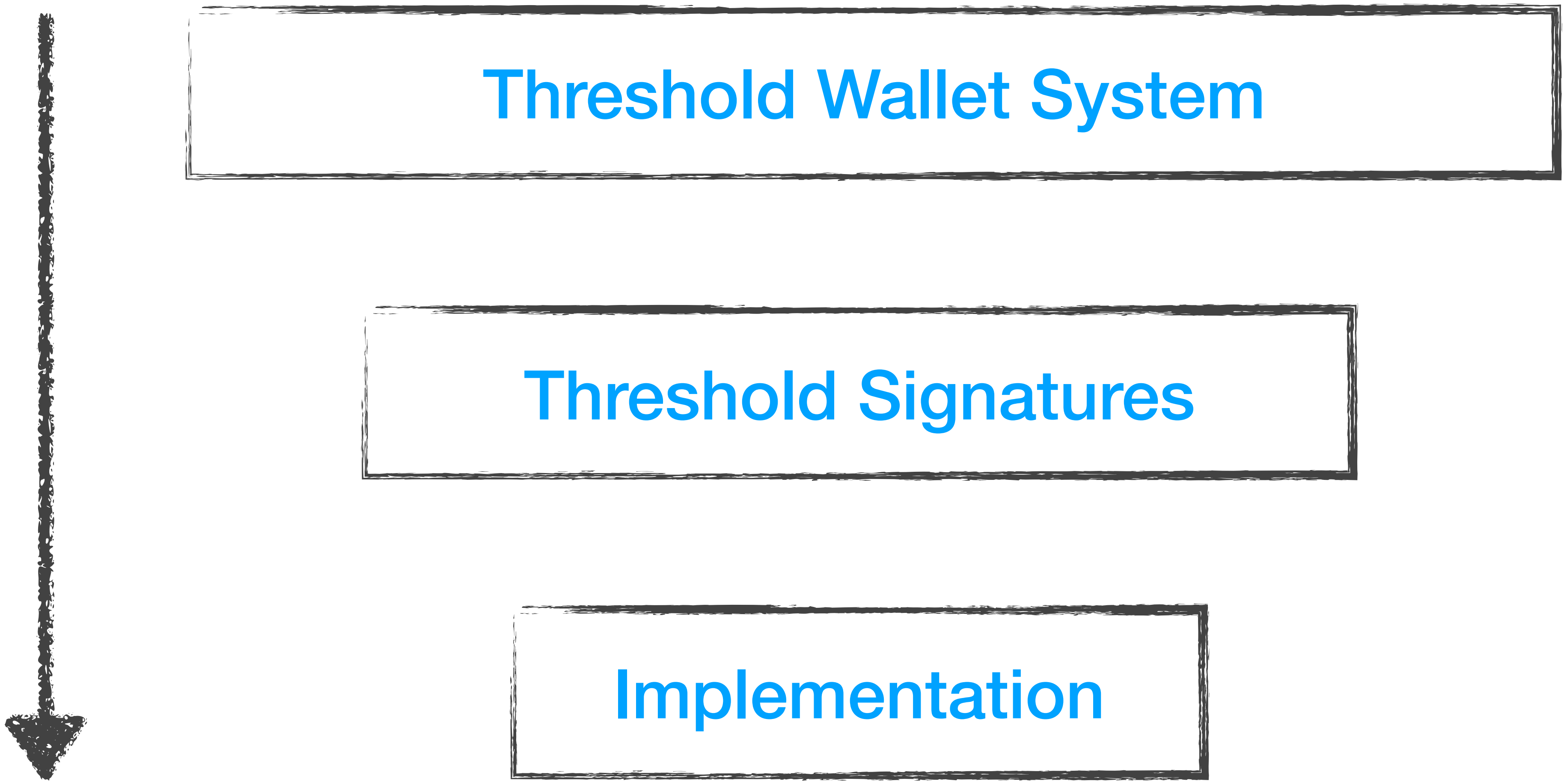
- Other differences:
  - Number of rounds
  - Chain support

Threshold Wallet - System

- Distributed network layer
- Who are the n parties?
- Do all parties have to run a full node?
- Can we obtain privacy between signing parties?
- Can we translate BIP32 to multi-party ?
- Single key system and multi-party KMS cannot co-exist

Focus



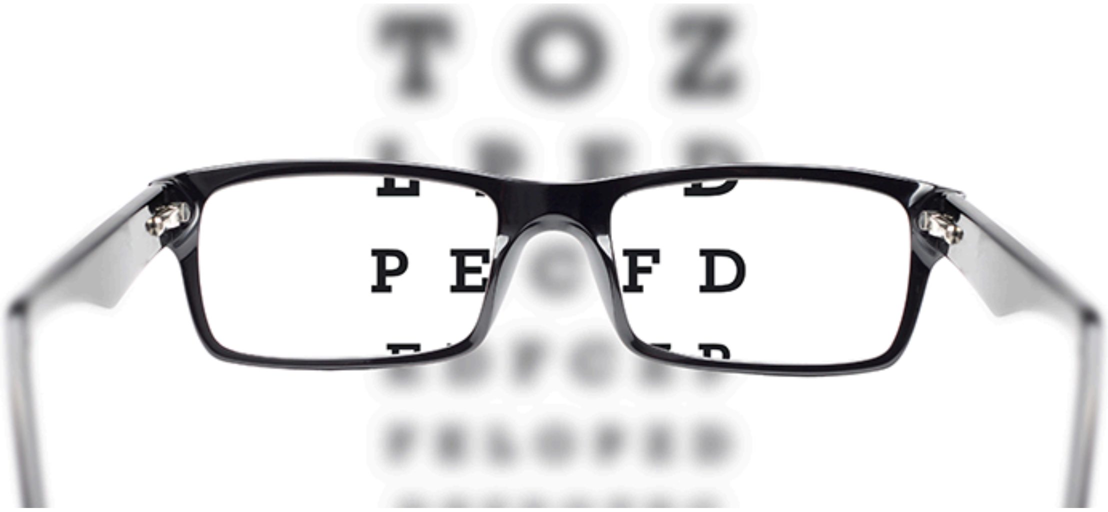


Threshold Wallet System

Threshold Signatures

Implementation

Focus



Threshold Wallet System

Threshold Signatures

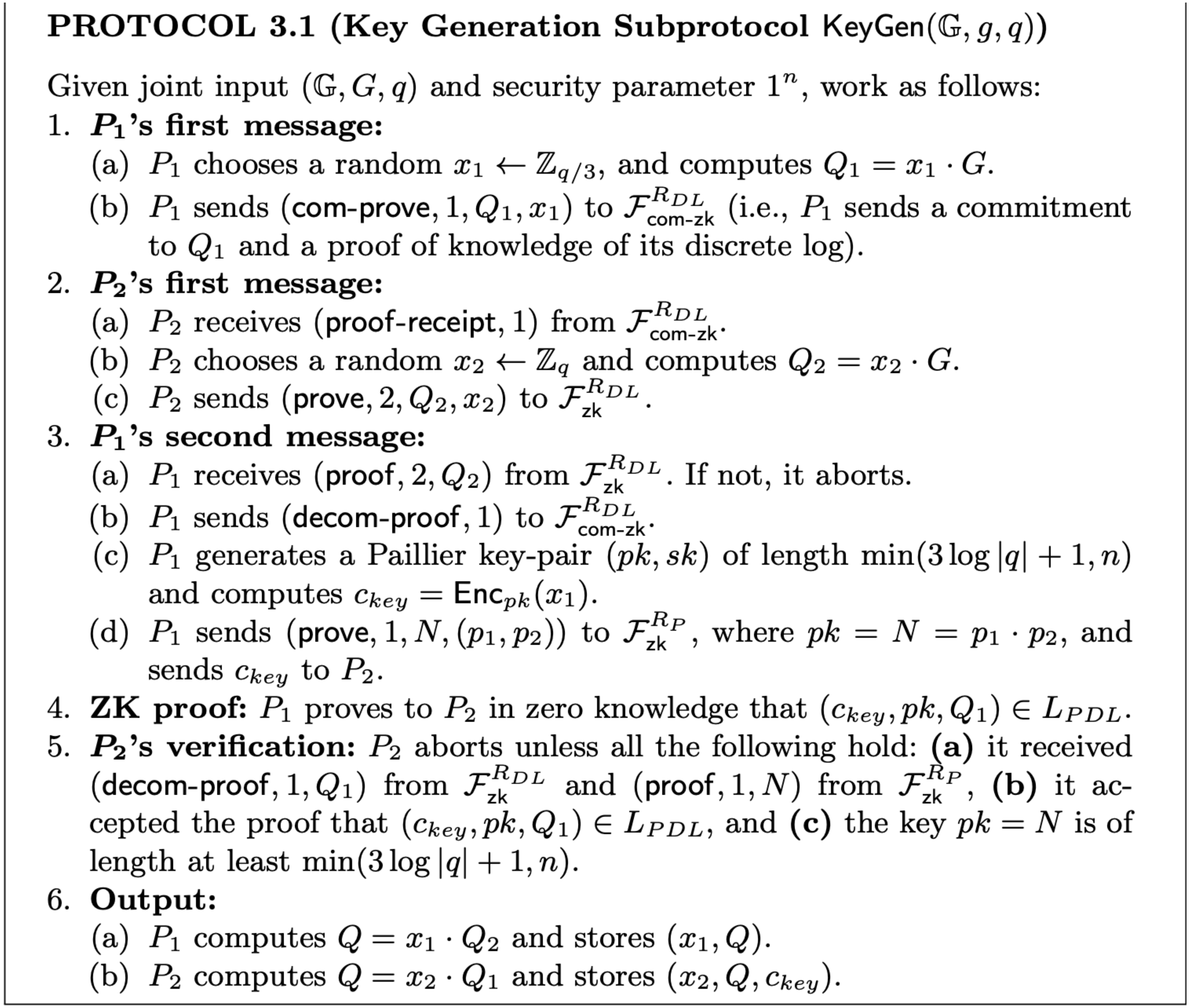
Implementation

- We take [L17] as a study case:
  - easier to explain
  - old enough to be implemented by several projects

ECDSA

- EC public parameters : q, G
- Choose Random k
- Compute $R = k \cdot G$
- Compute $r = r_x \bmod q$ where $R = (r_x, r_y)$
- Compute $s = k^{-1} \cdot (H(m) + r \cdot \boldsymbol{x}) \bmod q$ where $\boldsymbol{x}$ is the private key
- Output (r, s)

Lindell 2P-KeyGen

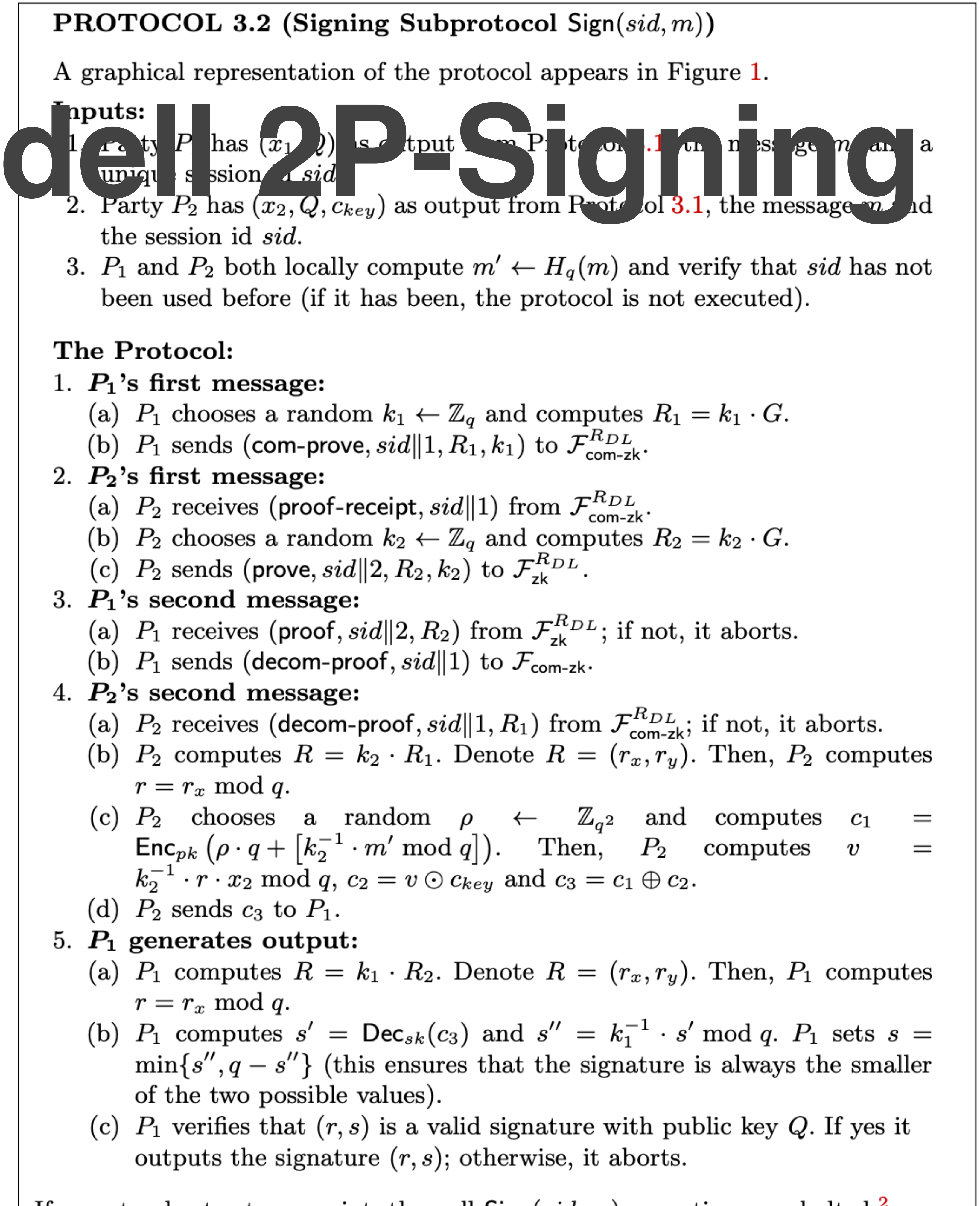


PROTOCOL 3.1 (Key Generation Subprotocol $\mathsf{KeyGen}(\mathbb{G}, g, q)$)

Given joint input $(\mathbb{G}, G, q)$ and security parameter 1^n, work as follows:

1. **P_1's first message:**
   (a) P_1 chooses a random $x_1 \leftarrow \mathbb{Z}_{q/3}$, and computes $Q_1 = x_1 \cdot G$.
   (b) P_1 sends $(\mathsf{com\text{-}prove}, 1, Q_1, x_1)$ to $\mathcal{F}_{\mathsf{com\text{-}zk}}^{R_{DL}}$ (i.e., P_1 sends a commitment to Q_1 and a proof of knowledge of its discrete log).
2. **P_2's first message:**
   (a) P_2 receives $(\mathsf{proof\text{-}receipt}, 1)$ from $\mathcal{F}_{\mathsf{com\text{-}zk}}^{R_{DL}}$.
   (b) P_2 chooses a random $x_2 \leftarrow \mathbb{Z}_q$ and computes $Q_2 = x_2 \cdot G$.
   (c) P_2 sends $(\mathsf{prove}, 2, Q_2, x_2)$ to $\mathcal{F}_{\mathsf{zk}}^{R_{DL}}$.
3. **P_1's second message:**
   (a) P_1 receives $(\mathsf{proof}, 2, Q_2)$ from $\mathcal{F}_{\mathsf{zk}}^{R_{DL}}$. If not, it aborts.
   (b) P_1 sends $(\mathsf{decom\text{-}proof}, 1)$ to $\mathcal{F}_{\mathsf{com\text{-}zk}}^{R_{DL}}$.
   (c) P_1 generates a Paillier key-pair (pk, sk) of length $\min(3 \log |q| + 1, n)$ and computes $c_{key} = \mathsf{Enc}_{pk}(x_1)$.
   (d) P_1 sends $(\mathsf{prove}, 1, N, (p_1, p_2))$ to $\mathcal{F}_{\mathsf{zk}}^{R_P}$, where $pk = N = p_1 \cdot p_2$, and sends c_{key} to P_2.
4. **ZK proof:** P_1 proves to P_2 in zero knowledge that $(c_{key}, pk, Q_1) \in L_{PDL}$.
5. **P_2's verification:** P_2 aborts unless all the following hold: **(a)** it received $(\mathsf{decom\text{-}proof}, 1, Q_1)$ from $\mathcal{F}_{\mathsf{zk}}^{R_{DL}}$ and $(\mathsf{proof}, 1, N)$ from $\mathcal{F}_{\mathsf{zk}}^{R_P}$, **(b)** it accepted the proof that $(c_{key}, pk, Q_1) \in L_{PDL}$, and **(c)** the key $pk = N$ is of length at least $\min(3 \log |q| + 1, n)$.
6. **Output:**
   (a) P_1 computes $Q = x_1 \cdot Q_2$ and stores (x_1, Q).
   (b) P_2 computes $Q = x_2 \cdot Q_1$ and stores (x_2, Q, c_{key}).

Lindell 2P-Signing



PROTOCOL 3.2 (Signing Subprotocol $\mathsf{Sign}(sid, m)$)

A graphical representation of the protocol appears in Figure 1.

Inputs:

1. Party P_1 has (x_1, Q) as output from Protocol 3.1, the message m and a unique session id sid.
2. Party P_2 has (x_2, Q, c_{key}) as output from Protocol 3.1, the message m and the session id sid.
3. P_1 and P_2 both locally compute $m' \leftarrow H_q(m)$ and verify that sid has not been used before (if it has been, the protocol is not executed).

The Protocol:

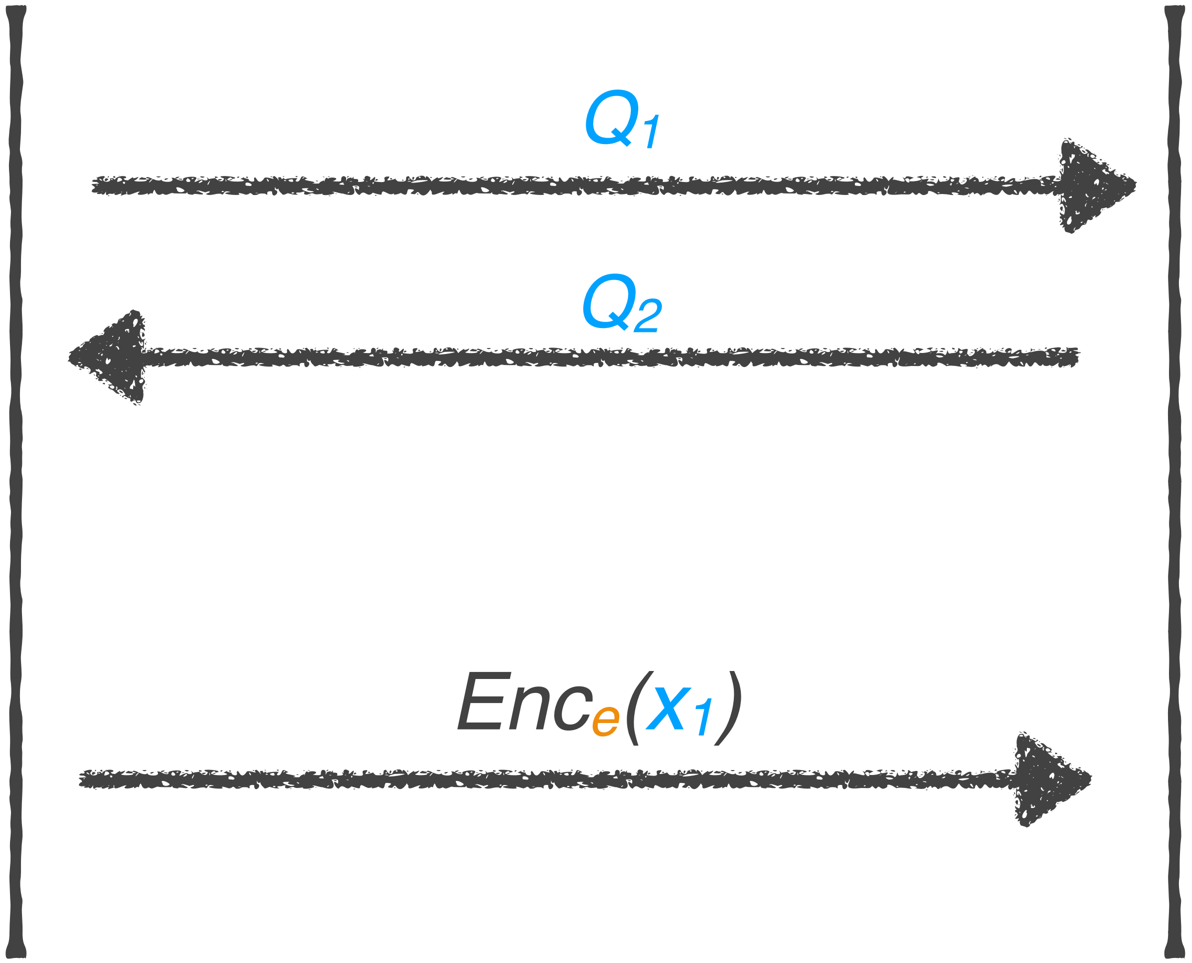
1. **P_1's first message:**
    (a) P_1 chooses a random $k_1 \leftarrow \mathbb{Z}_q$ and computes $R_1 = k_1 \cdot G$.
    (b) P_1 sends $(\mathsf{com\text{-}prove}, sid\|1, R_1, k_1)$ to $\mathcal{F}^{R_{DL}}_{\mathsf{com\text{-}zk}}$.
2. **P_2's first message:**
    (a) P_2 receives $(\mathsf{proof\text{-}receipt}, sid\|1)$ from $\mathcal{F}^{R_{DL}}_{\mathsf{com\text{-}zk}}$.
    (b) P_2 chooses a random $k_2 \leftarrow \mathbb{Z}_q$ and computes $R_2 = k_2 \cdot G$.
    (c) P_2 sends $(\mathsf{prove}, sid\|2, R_2, k_2)$ to $\mathcal{F}^{R_{DL}}_{\mathsf{zk}}$.
3. **P_1's second message:**
    (a) P_1 receives $(\mathsf{proof}, sid\|2, R_2)$ from $\mathcal{F}^{R_{DL}}_{\mathsf{zk}}$; if not, it aborts.
    (b) P_1 sends $(\mathsf{decom\text{-}proof}, sid\|1)$ to $\mathcal{F}_{\mathsf{com\text{-}zk}}$.
4. **P_2's second message:**
    (a) P_2 receives $(\mathsf{decom\text{-}proof}, sid\|1, R_1)$ from $\mathcal{F}^{R_{DL}}_{\mathsf{com\text{-}zk}}$; if not, it aborts.
    (b) P_2 computes $R = k_2 \cdot R_1$. Denote $R = (r_x, r_y)$. Then, P_2 computes $r = r_x \bmod q$.
    (c) P_2 chooses a random $\rho \leftarrow \mathbb{Z}_{q^2}$ and computes $c_1 = \mathsf{Enc}_{pk}\left(\rho \cdot q + \left[k_2^{-1} \cdot m' \bmod q\right]\right)$. Then, P_2 computes $v = k_2^{-1} \cdot r \cdot x_2 \bmod q$, $c_2 = v \odot c_{key}$ and $c_3 = c_1 \oplus c_2$.
    (d) P_2 sends c_3 to P_1.
5. **P_1 generates output:**
    (a) P_1 computes $R = k_1 \cdot R_2$. Denote $R = (r_x, r_y)$. Then, P_1 computes $r = r_x \bmod q$.
    (b) P_1 computes $s' = \mathsf{Dec}_{sk}(c_3)$ and $s'' = k_1^{-1} \cdot s' \bmod q$. P_1 sets $s = \min\{s'', q - s''\}$ (this ensures that the signature is always the smaller of the two possible values).
    (c) P_1 verifies that (r, s) is a valid signature with public key Q. If yes it outputs the signature (r, s); otherwise, it aborts.

2P-Keygen [L17]

Party1

$Q_1 = x_1 \cdot G$

$Q = Q_1 + Q_2$



Q_1

Q_2

$Enc_e(x_1)$

Party2

$Q_2 = x_2 \cdot G$

$Q = Q_1 + Q_2$

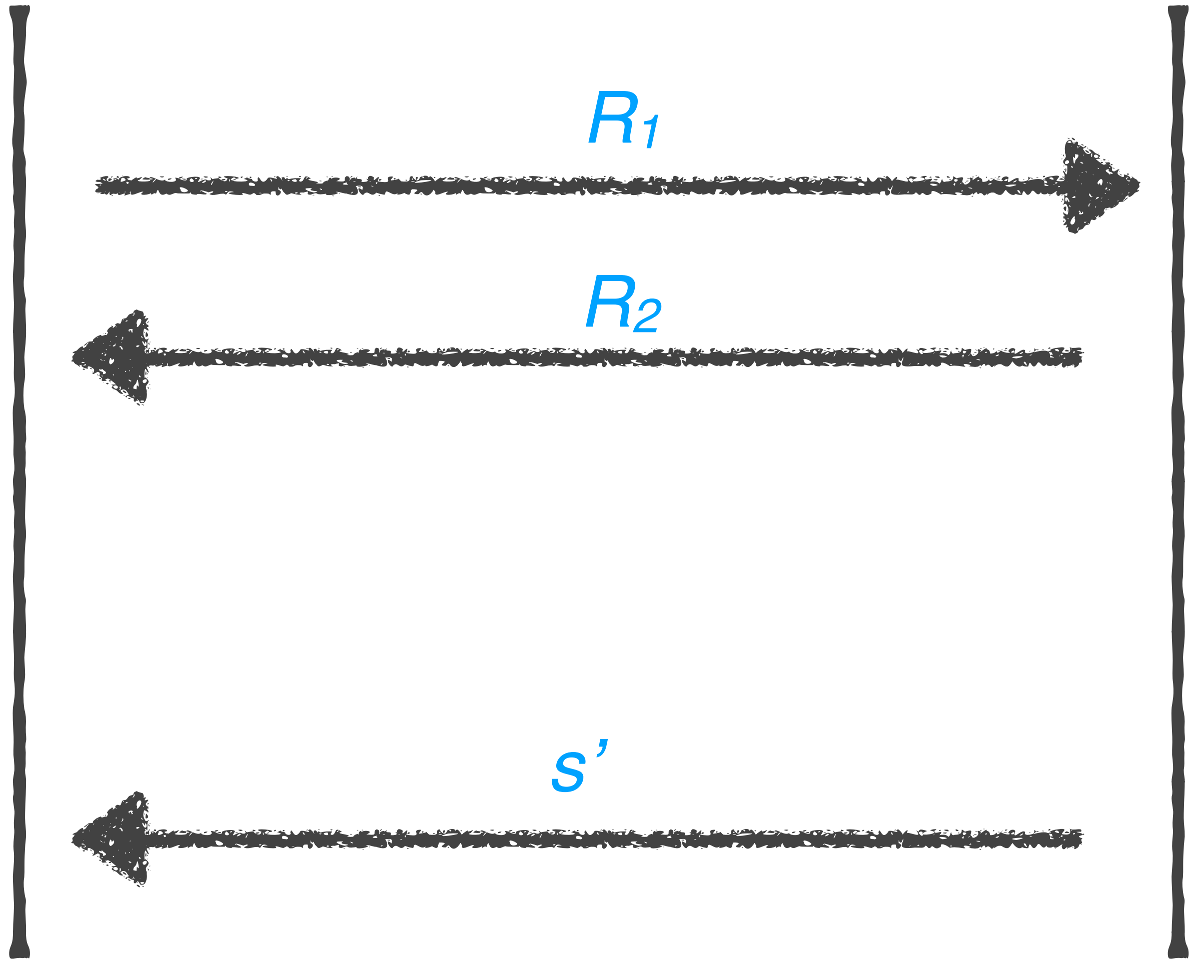
The protocol promises: (1) Privacy, (2) Correctness

2P-Signing [L17]

Signing message m: $m' = \text{Hash}(m)$

Party1

Party2



$R_1 = k_1 \cdot G$

$R_2 = k_2 \cdot G$

$\xrightarrow{R_1}$

$\xleftarrow{R_2}$

$R = k_1 \cdot R$

$R = k_2 \cdot R_1$

$s' =$

$Enc_e(m' / k_2) \boxplus$

$Enc_e(x_1) \odot Enc_e(x_2 \cdot r / k_2)$

$\xleftarrow{s'}$

$s = Dec_d(s')/k_1$

Output: $\boldsymbol{\sigma} = (s, r)$, s.t. $Verify(\boldsymbol{\sigma}, Q, m') = 1$

The protocol promises: (1) Completeness (2) Consistency (3) Unforgeability

Paillier CryptoSystem

Paillier is additively homomorphic public key encryption scheme

- Homomorphic addition of plaintexts: $Dec_d(Enc_e(a) \boxplus Enc_e(b)) = a + b$
- Homomorphic multiplication by scalar: $Dec_d(Enc_e(a) \odot k) = a \cdot k$

Paillier CryptoSystem

Paillier is additively homomorphic public key encryption scheme

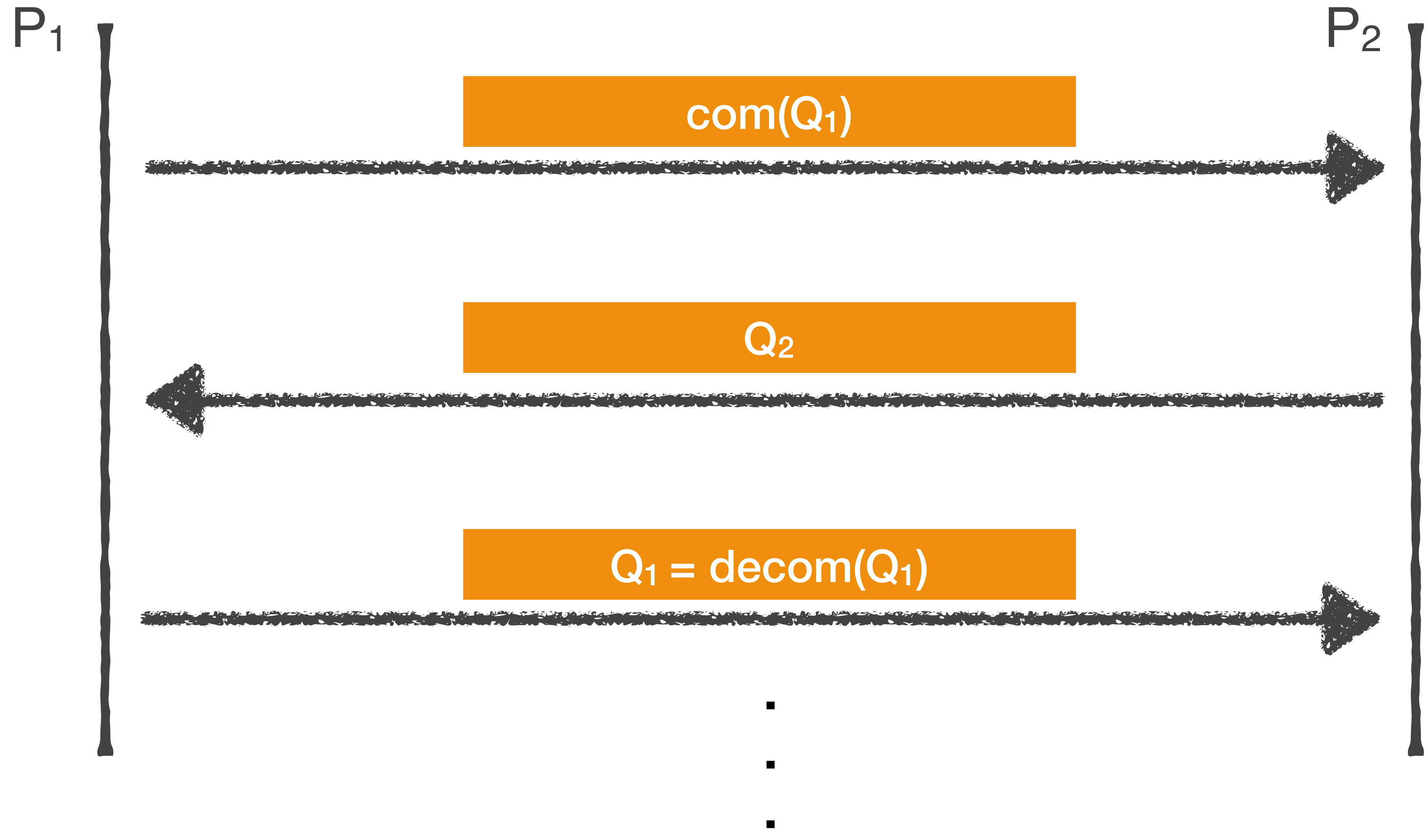
- Homomorphic addition of plaintexts: $Dec_d(Enc_e(a) \boxplus Enc_e(b)) = a + b$
- Homomorphic multiplication by scalar: $Dec_d(Enc_e(a) \odot k) = a \cdot k$

- Zero-knowledge proofs (https://github.com/KZen-networks/zk-paillier):
  - Proof of correct key generation (d,e)
  - Range proof (r): $c = Enc_e(a),\ a < r$
  - Proof that two ciphertexts encrypts the same message : $c_1 = Enc_{e_1}(a),\ c_2 = Enc_{e_2}(b),\ b == a$
  - …

Getting Dirty

- All examples were found in the wild
- Most of them in our multi-party-ECDSA code:
    - https://github.com/KZen-networks/multi-party-ecdsa
- All other important issues were reported and fixed

DO NOT OPTIMIZE #1



P_1

P_2

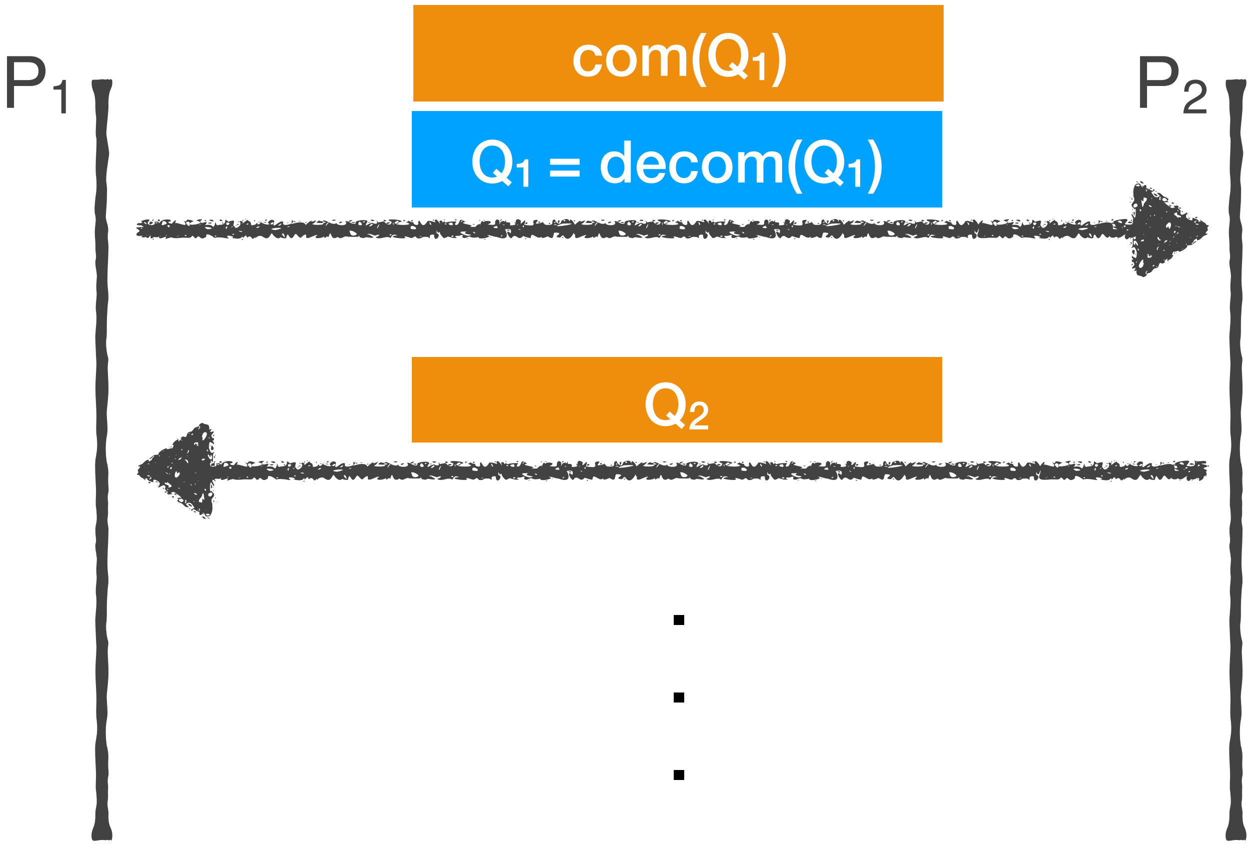
com(Q_1)

Q_2

Q_1 = decom(Q_1)

.
.
.

DO NOT OPTIMIZE #1



P_1

com(Q_1)

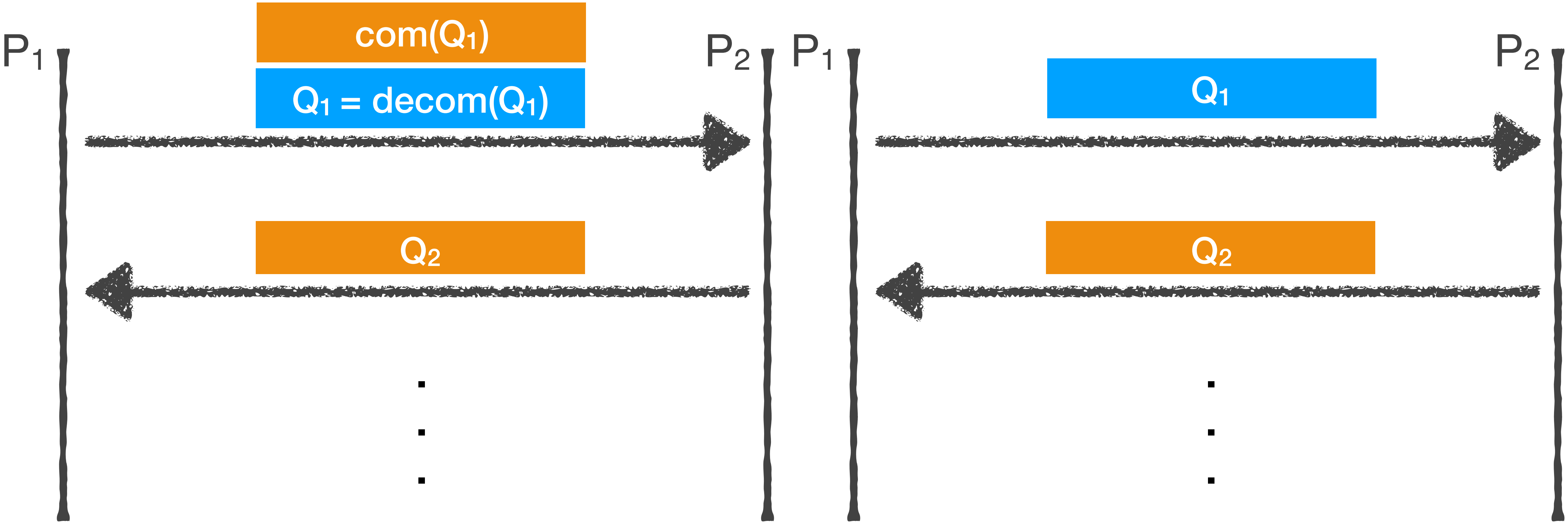
Q_1 = decom(Q_1)

P_2

Q_2

⋮

#1 DO NOT OPTIMIZE



P_1

$\text{com}(Q_1)$

$Q_1 = \text{decom}(Q_1)$

P_2

Q_2

⋮

P_1

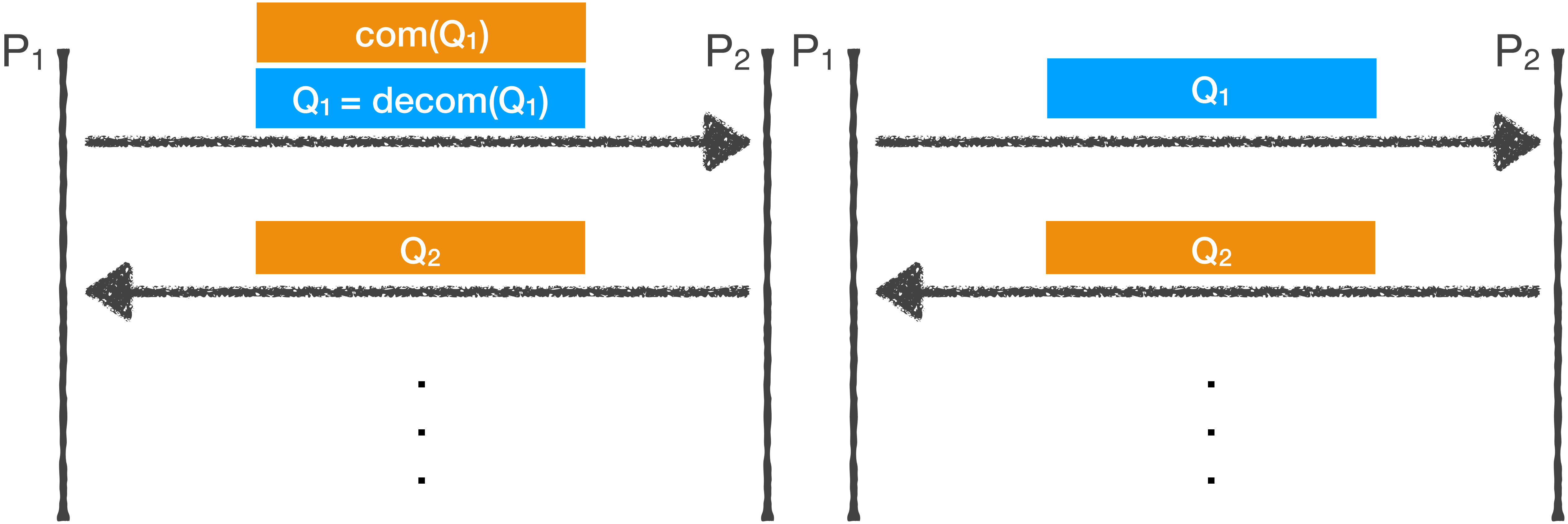
Q_1

P_2

Q_2

⋮

#1 DO NOT OPTIMIZE



P_1 P_2

$com(Q_1)$

$Q_1 = decom(Q_1)$

Q_2

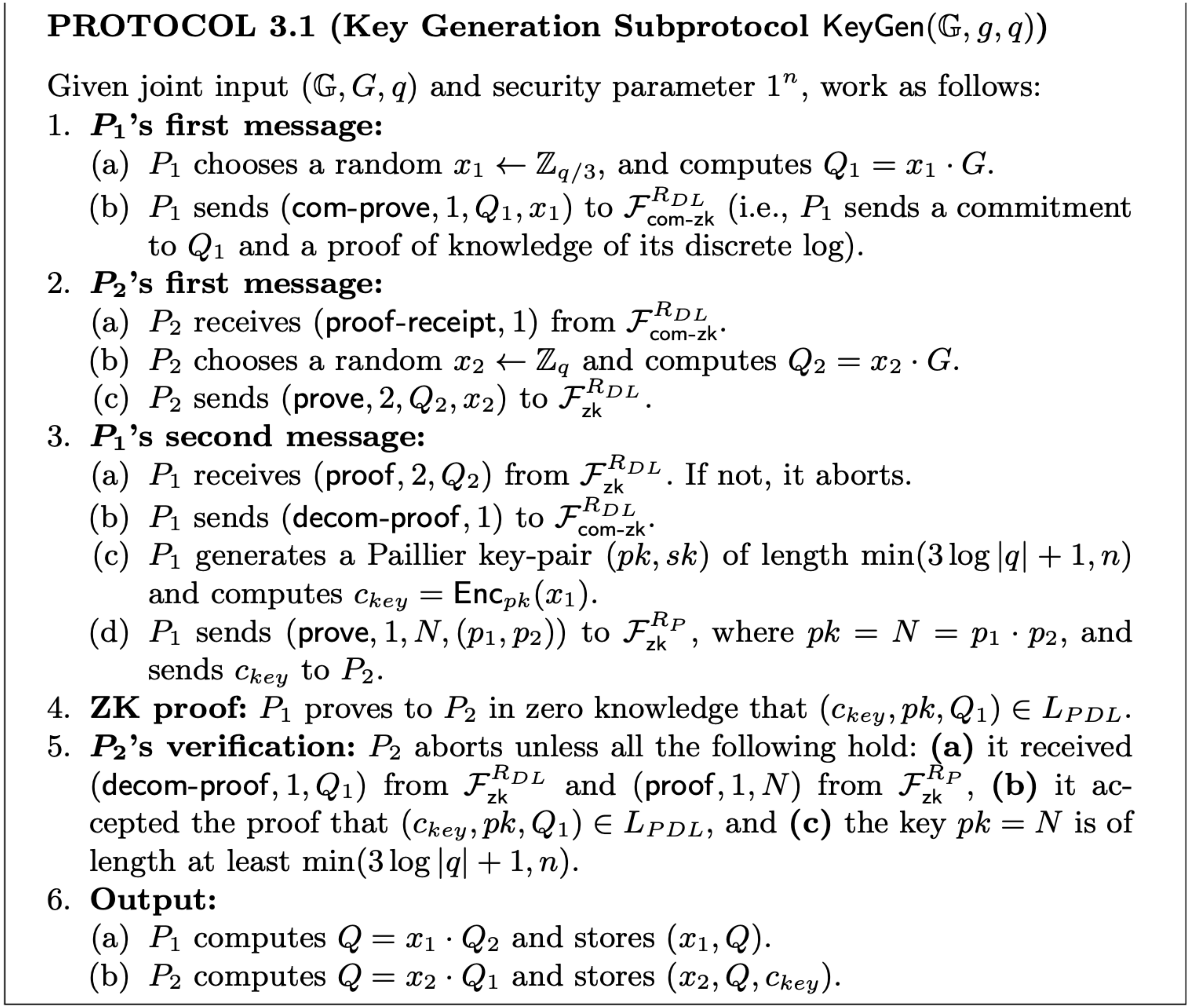
P_1 P_2

Q_1

Q_2

Rogue Key Attack (https://eprint.iacr.org/2018/068.pdf): $Q_2 = Q - Q_1$

#2 DO NOT OPTIMIZE

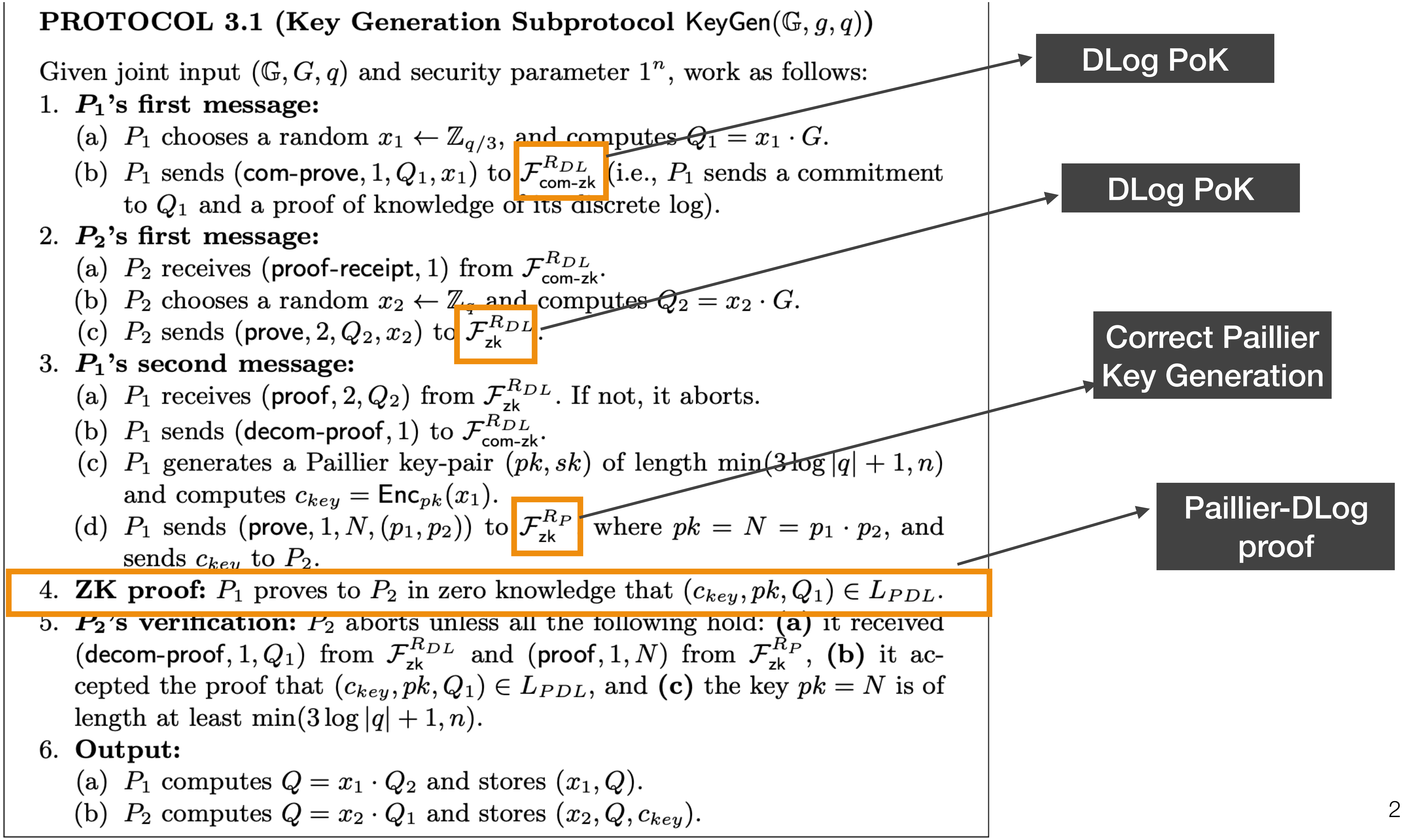


PROTOCOL 3.1 (Key Generation Subprotocol $\mathsf{KeyGen}(\mathbb{G}, g, q)$)

Given joint input $(\mathbb{G}, G, q)$ and security parameter 1^n, work as follows:

1. ***P_1*'s first message:**
   (a) P_1 chooses a random $x_1 \leftarrow \mathbb{Z}_{q/3}$, and computes $Q_1 = x_1 \cdot G$.
   (b) P_1 sends $(\mathsf{com\text{-}prove}, 1, Q_1, x_1)$ to $\mathcal{F}^{R_{DL}}_{\mathsf{com\text{-}zk}}$ (i.e., P_1 sends a commitment to Q_1 and a proof of knowledge of its discrete log).
2. ***P_2*'s first message:**
   (a) P_2 receives $(\mathsf{proof\text{-}receipt}, 1)$ from $\mathcal{F}^{R_{DL}}_{\mathsf{com\text{-}zk}}$.
   (b) P_2 chooses a random $x_2 \leftarrow \mathbb{Z}_q$ and computes $Q_2 = x_2 \cdot G$.
   (c) P_2 sends $(\mathsf{prove}, 2, Q_2, x_2)$ to $\mathcal{F}^{R_{DL}}_{\mathsf{zk}}$.
3. ***P_1*'s second message:**
   (a) P_1 receives $(\mathsf{proof}, 2, Q_2)$ from $\mathcal{F}^{R_{DL}}_{\mathsf{zk}}$. If not, it aborts.
   (b) P_1 sends $(\mathsf{decom\text{-}proof}, 1)$ to $\mathcal{F}^{R_{DL}}_{\mathsf{com\text{-}zk}}$.
   (c) P_1 generates a Paillier key-pair (pk, sk) of length $\min(3\log|q| + 1, n)$ and computes $c_{key} = \mathsf{Enc}_{pk}(x_1)$.
   (d) P_1 sends $(\mathsf{prove}, 1, N, (p_1, p_2))$ to $\mathcal{F}^{R_P}_{\mathsf{zk}}$, where $pk = N = p_1 \cdot p_2$, and sends c_{key} to P_2.
4. **ZK proof:** P_1 proves to P_2 in zero knowledge that $(c_{key}, pk, Q_1) \in L_{PDL}$.
5. ***P_2*'s verification:** P_2 aborts unless all the following hold: **(a)** it received $(\mathsf{decom\text{-}proof}, 1, Q_1)$ from $\mathcal{F}^{R_{DL}}_{\mathsf{zk}}$ and $(\mathsf{proof}, 1, N)$ from $\mathcal{F}^{R_P}_{\mathsf{zk}}$, **(b)** it accepted the proof that $(c_{key}, pk, Q_1) \in L_{PDL}$, and **(c)** the key $pk = N$ is of length at least $\min(3\log|q| + 1, n)$.
6. **Output:**
   (a) P_1 computes $Q = x_1 \cdot Q_2$ and stores (x_1, Q).
   (b) P_2 computes $Q = x_2 \cdot Q_1$ and stores (x_2, Q, c_{key}).

#2 DO NOT OPTIMIZE



PROTOCOL 3.1 (Key Generation Subprotocol $\mathsf{KeyGen}(\mathbb{G}, g, q)$)

Given joint input $(\mathbb{G}, G, q)$ and security parameter 1^n, work as follows:

1. **P_1's first message:**
   - (a) P_1 chooses a random $x_1 \leftarrow \mathbb{Z}_{q/3}$, and computes $Q_1 = x_1 \cdot G$.
   - (b) P_1 sends $(\mathsf{com\text{-}prove}, 1, Q_1, x_1)$ to $\boxed{\mathcal{F}_{\mathsf{com\text{-}zk}}^{R_{DL}}}$ (i.e., P_1 sends a commitment to Q_1 and a proof of knowledge of its discrete log).
2. **P_2's first message:**
   - (a) P_2 receives $(\mathsf{proof\text{-}receipt}, 1)$ from $\mathcal{F}_{\mathsf{com\text{-}zk}}^{R_{DL}}$.
   - (b) P_2 chooses a random $x_2 \leftarrow \mathbb{Z}_q$ and computes $Q_2 = x_2 \cdot G$.
   - (c) P_2 sends $(\mathsf{prove}, 2, Q_2, x_2)$ to $\boxed{\mathcal{F}_{\mathsf{zk}}^{R_{DL}}}$.
3. **P_1's second message:**
   - (a) P_1 receives $(\mathsf{proof}, 2, Q_2)$ from $\mathcal{F}_{\mathsf{zk}}^{R_{DL}}$. If not, it aborts.
   - (b) P_1 sends $(\mathsf{decom\text{-}proof}, 1)$ to $\mathcal{F}_{\mathsf{com\text{-}zk}}^{R_{DL}}$.
   - (c) P_1 generates a Paillier key-pair (pk, sk) of length $\min(3 \log |q| + 1, n)$ and computes $c_{key} = \mathsf{Enc}_{pk}(x_1)$.
   - (d) P_1 sends $(\mathsf{prove}, 1, N, (p_1, p_2))$ to $\boxed{\mathcal{F}_{\mathsf{zk}}^{R_P}}$ where $pk = N = p_1 \cdot p_2$, and sends c_{key} to P_2.
4. **ZK proof:** P_1 proves to P_2 in zero knowledge that $(c_{key}, pk, Q_1) \in L_{PDL}$.
5. **P_2's verification:** P_2 aborts unless all the following hold: **(a)** it received $(\mathsf{decom\text{-}proof}, 1, Q_1)$ from $\mathcal{F}_{\mathsf{zk}}^{R_{DL}}$ and $(\mathsf{proof}, 1, N)$ from $\mathcal{F}_{\mathsf{zk}}^{R_P}$, **(b)** it accepted the proof that $(c_{key}, pk, Q_1) \in L_{PDL}$, and **(c)** the key $pk = N$ is of length at least $\min(3 \log |q| + 1, n)$.
6. **Output:**
   - (a) P_1 computes $Q = x_1 \cdot Q_2$ and stores (x_1, Q).
   - (b) P_2 computes $Q = x_2 \cdot Q_1$ and stores (x_2, Q, c_{key}).

DLog PoK

DLog PoK

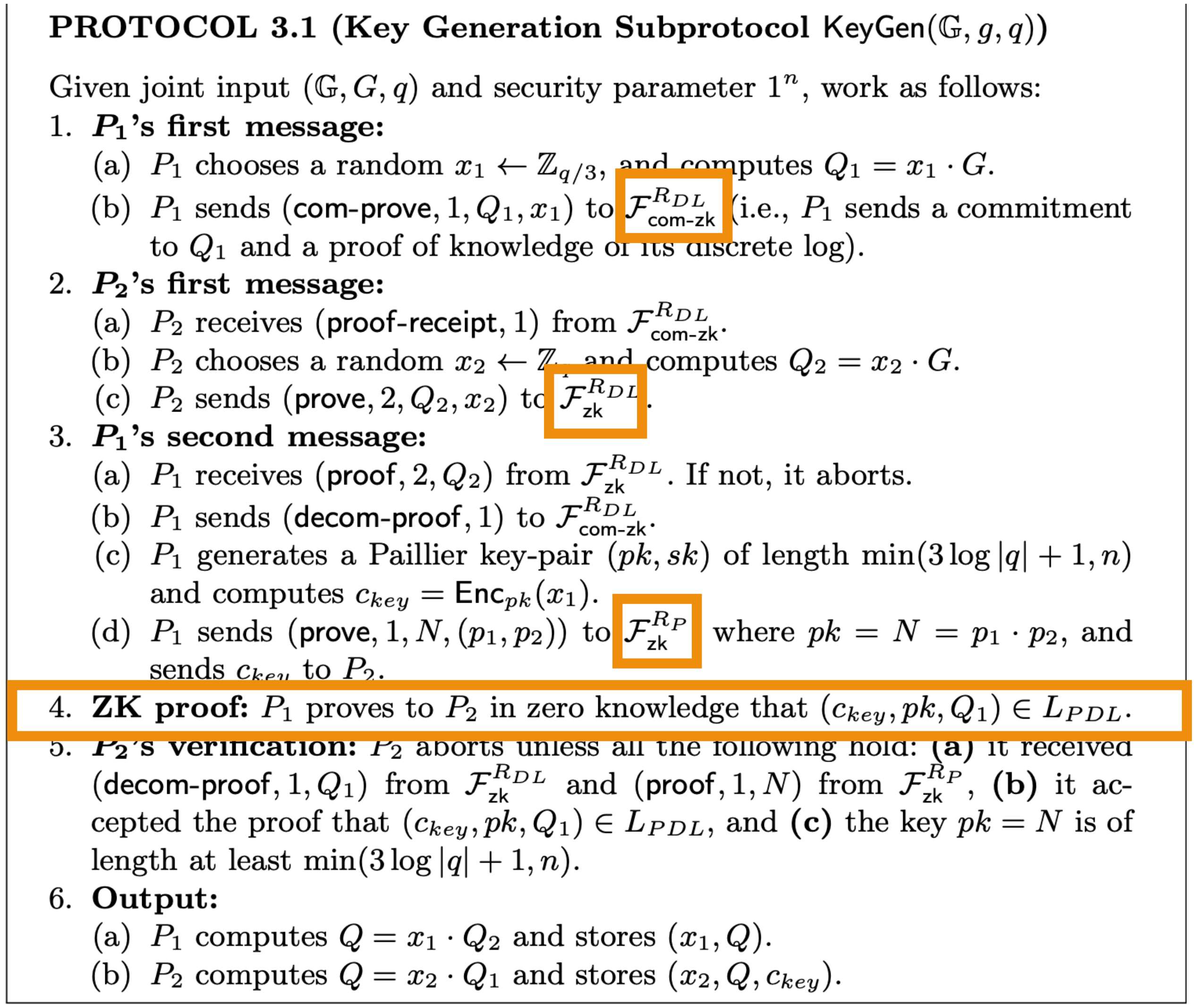
Correct Paillier Key Generation

Paillier-DLog proof

DO NOT OPTIMIZE #2

Hey!

ZK proofs can be removed and code still works!



PROTOCOL 3.1 (Key Generation Subprotocol $\mathsf{KeyGen}(\mathbb{G}, g, q)$)

Given joint input $(\mathbb{G}, G, q)$ and security parameter 1^n, work as follows:

1. ***P_1*'s first message:**
   - (a) P_1 chooses a random $x_1 \leftarrow \mathbb{Z}_{q/3}$, and computes $Q_1 = x_1 \cdot G$.
   - (b) P_1 sends $(\mathsf{com\text{-}prove}, 1, Q_1, x_1)$ to $\mathcal{F}^{R_{DL}}_{\mathsf{com\text{-}zk}}$ (i.e., P_1 sends a commitment to Q_1 and a proof of knowledge of its discrete log).
2. ***P_2*'s first message:**
   - (a) P_2 receives $(\mathsf{proof\text{-}receipt}, 1)$ from $\mathcal{F}^{R_{DL}}_{\mathsf{com\text{-}zk}}$.
   - (b) P_2 chooses a random $x_2 \leftarrow \mathbb{Z}_q$, and computes $Q_2 = x_2 \cdot G$.
   - (c) P_2 sends $(\mathsf{prove}, 2, Q_2, x_2)$ to $\mathcal{F}^{R_{DL}}_{\mathsf{zk}}$.
3. ***P_1*'s second message:**
   - (a) P_1 receives $(\mathsf{proof}, 2, Q_2)$ from $\mathcal{F}^{R_{DL}}_{\mathsf{zk}}$. If not, it aborts.
   - (b) P_1 sends $(\mathsf{decom\text{-}proof}, 1)$ to $\mathcal{F}^{R_{DL}}_{\mathsf{com\text{-}zk}}$.
   - (c) P_1 generates a Paillier key-pair (pk, sk) of length $\min(3\log|q| + 1, n)$ and computes $c_{key} = \mathsf{Enc}_{pk}(x_1)$.
   - (d) P_1 sends $(\mathsf{prove}, 1, N, (p_1, p_2))$ to $\mathcal{F}^{R_P}_{\mathsf{zk}}$ where $pk = N = p_1 \cdot p_2$, and sends c_{key} to P_2.
4. **ZK proof:** P_1 proves to P_2 in zero knowledge that $(c_{key}, pk, Q_1) \in L_{PDL}$.
5. ***P_2*'s verification:** P_2 aborts unless all the following hold: **(a)** it received $(\mathsf{decom\text{-}proof}, 1, Q_1)$ from $\mathcal{F}^{R_{DL}}_{\mathsf{zk}}$ and $(\mathsf{proof}, 1, N)$ from $\mathcal{F}^{R_P}_{\mathsf{zk}}$, **(b)** it accepted the proof that $(c_{key}, pk, Q_1) \in L_{PDL}$, and **(c)** the key $pk = N$ is of length at least $\min(3\log|q| + 1, n)$.
6. **Output:**
   - (a) P_1 computes $Q = x_1 \cdot Q_2$ and stores (x_1, Q).
   - (b) P_2 computes $Q = x_2 \cdot Q_1$ and stores (x_2, Q, c_{key}).

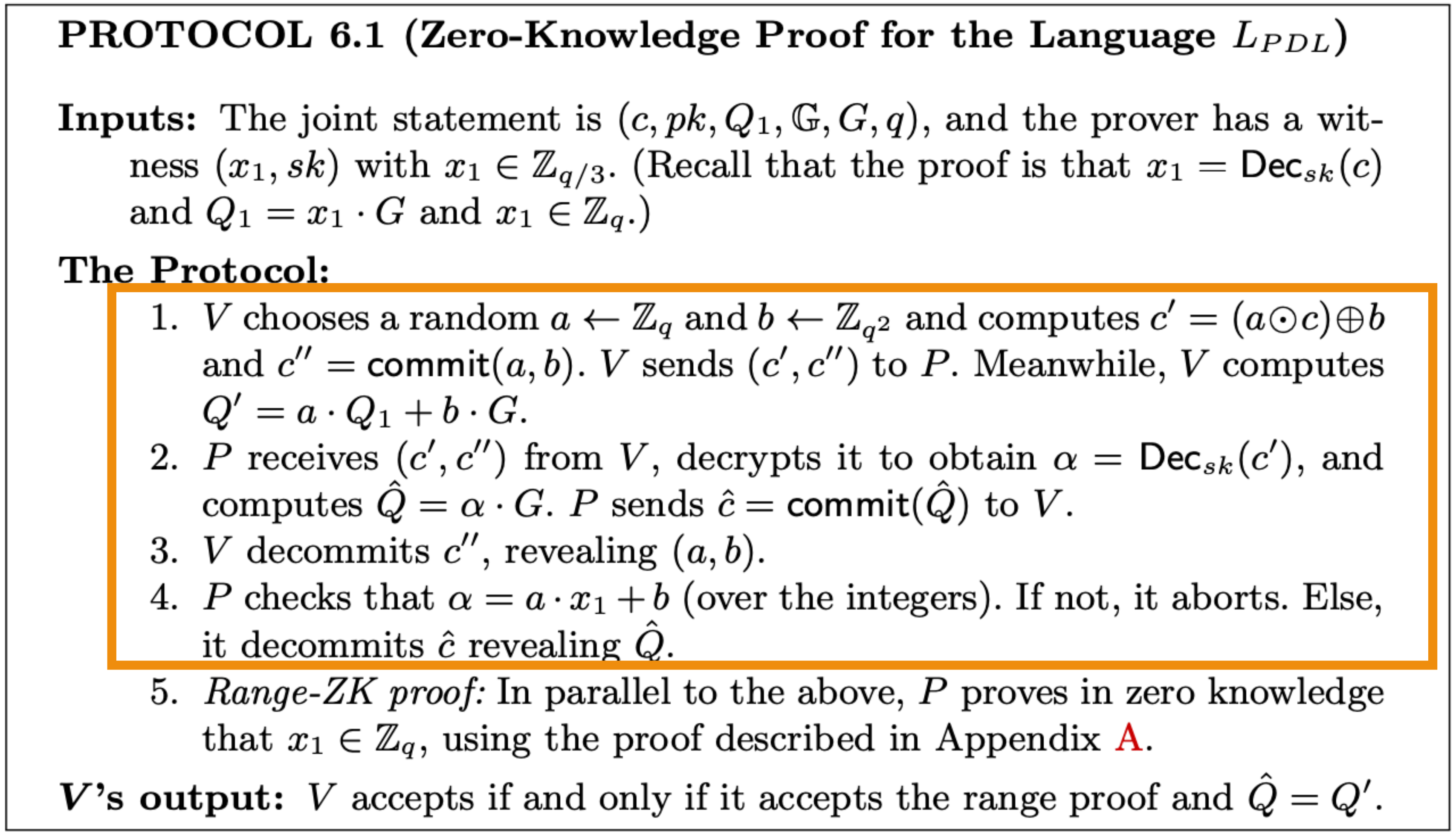
[ZK in MPC]

- Zero knowledge proofs hold 3 properties: Completeness, Soundness and Zero-Knowledge
- MPC (at least here) uses ZK as part of security proof to protect against Malicious adversaries
- This mean that removing/changing the ZK proof will break the security proof

Broken security proof does not always lead to immediate attack

#2 DO NOT OPTIMIZE

- In KZen implementation of Lindell 2P-KeyGen we neglected $L_{PDL.}$ (to our defence we did integrate the Paillier range proof)



PROTOCOL 6.1 (Zero-Knowledge Proof for the Language L_{PDL})

Inputs: The joint statement is $(c, pk, Q_1, \mathbb{G}, G, q)$, and the prover has a witness (x_1, sk) with $x_1 \in \mathbb{Z}_{q/3}$. (Recall that the proof is that $x_1 = \mathsf{Dec}_{sk}(c)$ and $Q_1 = x_1 \cdot G$ and $x_1 \in \mathbb{Z}_q$.)

The Protocol:

1. V chooses a random $a \leftarrow \mathbb{Z}_q$ and $b \leftarrow \mathbb{Z}_{q^2}$ and computes $c' = (a \odot c) \oplus b$ and $c'' = \mathsf{commit}(a, b)$. V sends (c', c'') to P. Meanwhile, V computes $Q' = a \cdot Q_1 + b \cdot G$.
2. P receives (c', c'') from V, decrypts it to obtain $\alpha = \mathsf{Dec}_{sk}(c')$, and computes $\hat{Q} = \alpha \cdot G$. P sends $\hat{c} = \mathsf{commit}(\hat{Q})$ to V.
3. V decommits c'', revealing (a, b).
4. P checks that $\alpha = a \cdot x_1 + b$ (over the integers). If not, it aborts. Else, it decommits $\hat{c}$ revealing $\hat{Q}$.
5. *Range-ZK proof:* In parallel to the above, P proves in zero knowledge that $x_1 \in \mathbb{Z}_q$, using the proof described in Appendix A.

***V*'s output:** V accepts if and only if it accepts the range proof and $\hat{Q} = Q'$.

#3 ZK Confusion

- Proof of correct Paillier Keypair:
    - $R_P = \{(N, (p_1, p_2)), N = p_1 \cdot p_2$ and p_1, p_2 are primes$\}$

#3 ZK Confusion

- Proof of correct Paillier Keypair:
    - $R_P = \{(N, (p_1,p_2)), N = p_1 \bullet p_2$ and p_1,p_2 are primes$\}$
- The author used zk proof from an old paper:
    - $R'_P = \{(N,\phi(N)), \gcd((N,\phi(N)) = 1\}$

#3 ZK Confusion

- Proof of correct Paillier Keypair:
    - $R_P = \{(N, (p_1,p_2)), N = p_1 \cdot p_2$ and p_1,p_2 are primes$\}$

- The author used zk proof from an old paper:
    - $R'_P = \{(N,\phi(N)), \gcd((N,\phi(N)) = 1\}$

- This two relation are not equivalent. i.e if N is a product of 3 distinct primes of the same length

#3 ZK Confusion

- Proof of correct Paillier Keypair:
  - $R_P = \{(N, (p_1,p_2)), N = p_1 \cdot p_2$ and p_1,p_2 are primes$\}$

- The author used zk proof from an old paper:
  - $R'_P = \{(N,\phi(N)), \gcd((N,\phi(N)) = 1\}$

- This two relation are not equivalent. i.e if N is a product of 3 distinct primes of the same length

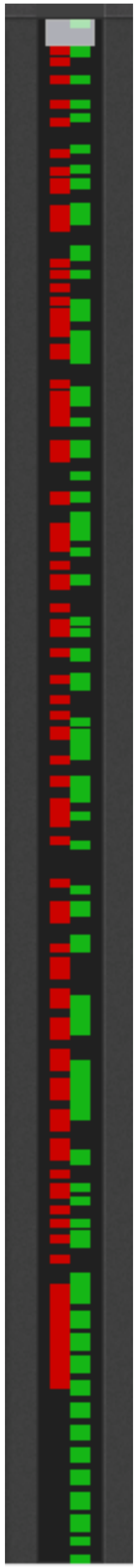
- This was fixed at a later version. Luckily, in this specific protocol R'_P *is enough*

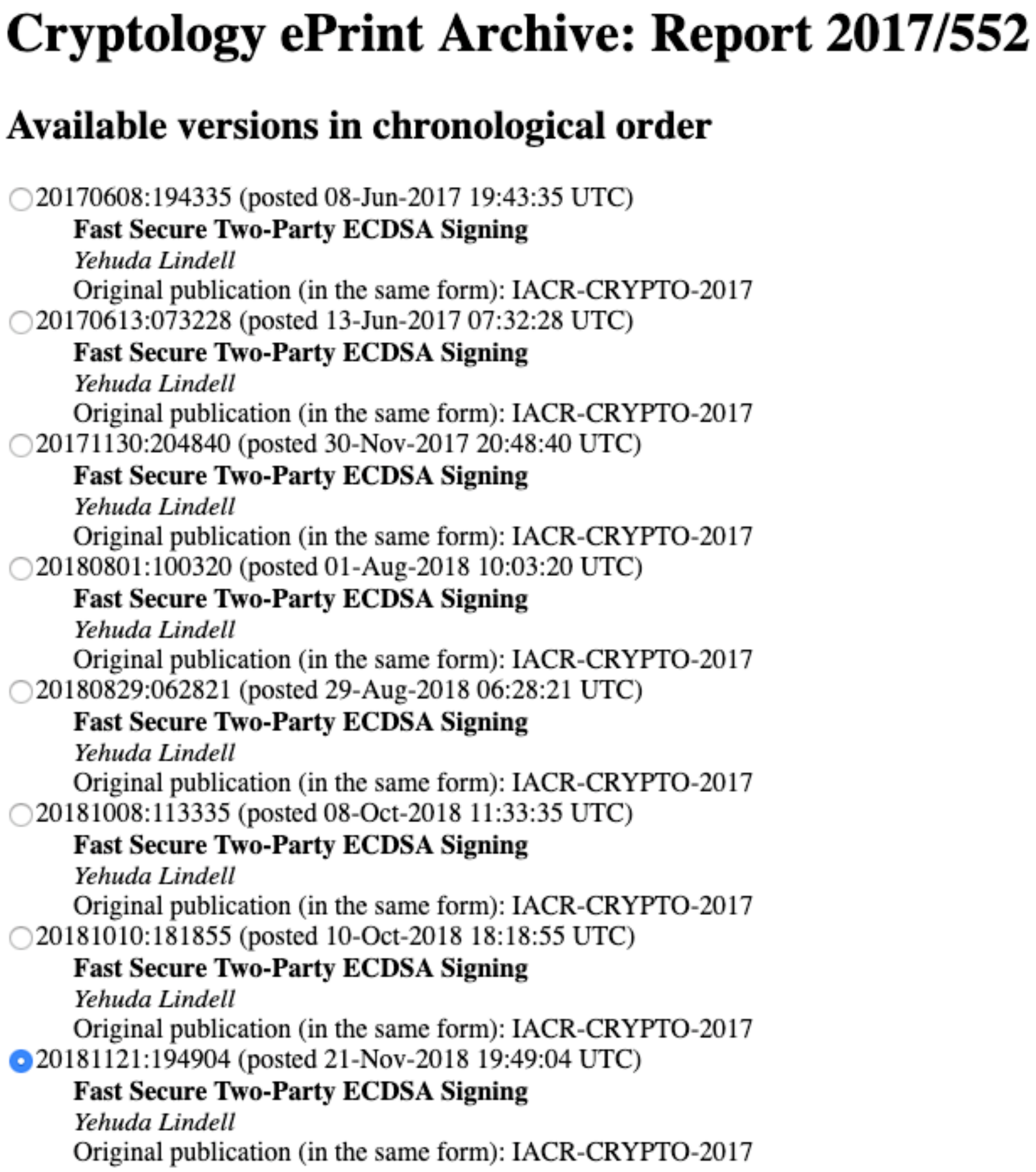
#4 ZK Confusion

- Not to neglect other papers:
  - [GG18] KeyGen is using an unnecessary proof of knowledge of DLog
  - [LNR18] zk Range proof protocol (6.2.5) is provided without a proof

#5 Work In Progress

- 8 versions in a span of 18 months
- First version accepted to Crypto17'
- +7 pages difference between first and last versions
- Unlike GitHub, there is no built in way to track changes
- In the last version there was an update replacing zkpok of DLog with zero knowledge of DH relation to enable concurrent signing





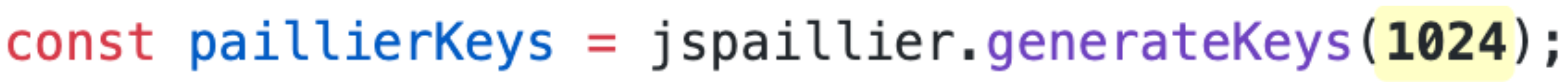
Cryptology ePrint Archive: Report 2017/552

Available versions in chronological order

○20170608:194335 (posted 08-Jun-2017 19:43:35 UTC)
Fast Secure Two-Party ECDSA Signing
Yehuda Lindell
Original publication (in the same form): IACR-CRYPTO-2017
○20170613:073228 (posted 13-Jun-2017 07:32:28 UTC)
Fast Secure Two-Party ECDSA Signing
Yehuda Lindell
Original publication (in the same form): IACR-CRYPTO-2017
○20171130:204840 (posted 30-Nov-2017 20:48:40 UTC)
Fast Secure Two-Party ECDSA Signing
Yehuda Lindell
Original publication (in the same form): IACR-CRYPTO-2017
○20180801:100320 (posted 01-Aug-2018 10:03:20 UTC)
Fast Secure Two-Party ECDSA Signing
Yehuda Lindell
Original publication (in the same form): IACR-CRYPTO-2017
○20180829:062821 (posted 29-Aug-2018 06:28:21 UTC)
Fast Secure Two-Party ECDSA Signing
Yehuda Lindell
Original publication (in the same form): IACR-CRYPTO-2017
○20181008:113335 (posted 08-Oct-2018 11:33:35 UTC)
Fast Secure Two-Party ECDSA Signing
Yehuda Lindell
Original publication (in the same form): IACR-CRYPTO-2017
○20181010:181855 (posted 10-Oct-2018 18:18:55 UTC)
Fast Secure Two-Party ECDSA Signing
Yehuda Lindell
Original publication (in the same form): IACR-CRYPTO-2017
●20181121:194904 (posted 21-Nov-2018 19:49:04 UTC)
Fast Secure Two-Party ECDSA Signing
Yehuda Lindell
Original publication (in the same form): IACR-CRYPTO-2017

#6 Bad Ways to Achieve Efficiency

- Zk proofs are to avoid malicious adversary but what if we use multiple devices of the same user: well in that case we can assume that all parties are acting honest



```
const paillierKeys = jspaillier.generateKeys(1024);
```

Good Ways to Achieve Efficiency

Good Ways to Achieve Efficiency

- If a protocol is abstracting Zero Knowledge proofs, they can be replaced with better version

Good Ways to Achieve Efficiency

- If a protocol is abstracting Zero Knowledge proofs, they can be replaced with better version
- Some protocols are presented as a chain of separate sub-protocols. Sometimes they can run in parallel instead of sequential.

Good Ways to Achieve Efficiency

- If a protocol is abstracting Zero Knowledge proofs, they can be replaced with better version
- Some protocols are presented as a chain of separate sub-protocols. Sometimes they can run in parallel instead of sequential.
- Trading stronger security assumption for efficiency

#7 Breaking a Threshold Wallet

Breaking 2P-Rotation

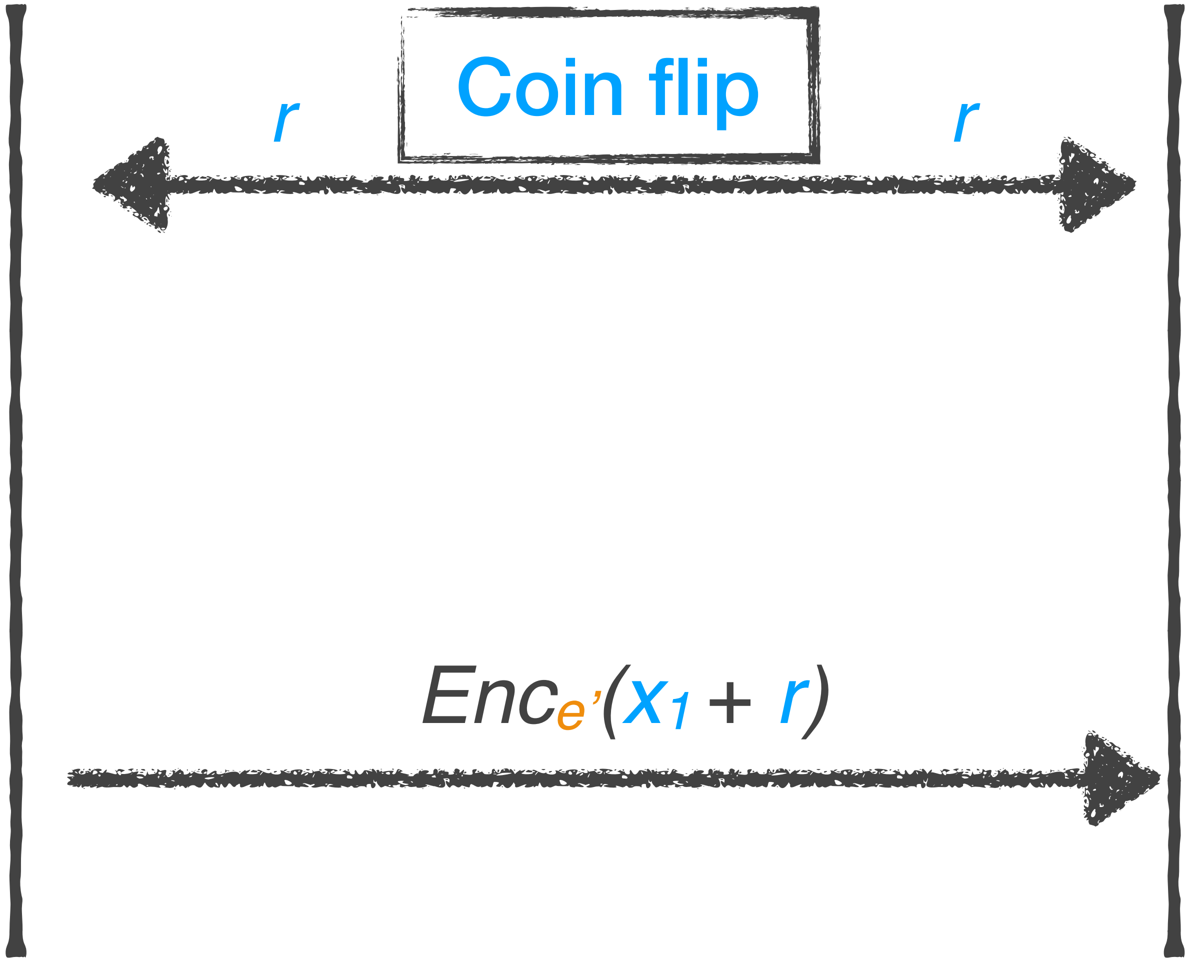
Call 2P-Sign and broken 2P-Rotation many times

Breaking 2P-KeyGen

2P-Rotation

Party1

Party2



Coin flip

r r

$Q'_1 = (x_1 + r) \cdot G$

$Q'_2 = (x_2 - r) \cdot G$

$Q' = Q$

$Q' = Q$

$Enc_{e'}(x_1 + r)$

2P-Rotation zk-Paillier

- P_1 Generates a new Paillier keypair (d', e') and encrypt $(x_1 + r)$ into

$$c_{key\_A} = Enc_{e'}(x_1 + r)$$

- P_2 homomorphically adds r to $Enc_e(x_1)$:

$$c_{key\_B} = Enc_e(x_1) \boxplus r$$

- P_1 Proves in zero knowledge that c_{key_A} and c_{key_B} encrypts the same message $x_1 + r$

2P-Rotation zk-Paillier

- P_1 Generates a new Paillier keypair (d',e') and encrypt ($x_1 + r$) into

$$\boxed{c_{key\_A} = Enc_{e'}(x_1 + r)}$$

- P_2 homomorphically adds r to $Enc_e(x_1)$:

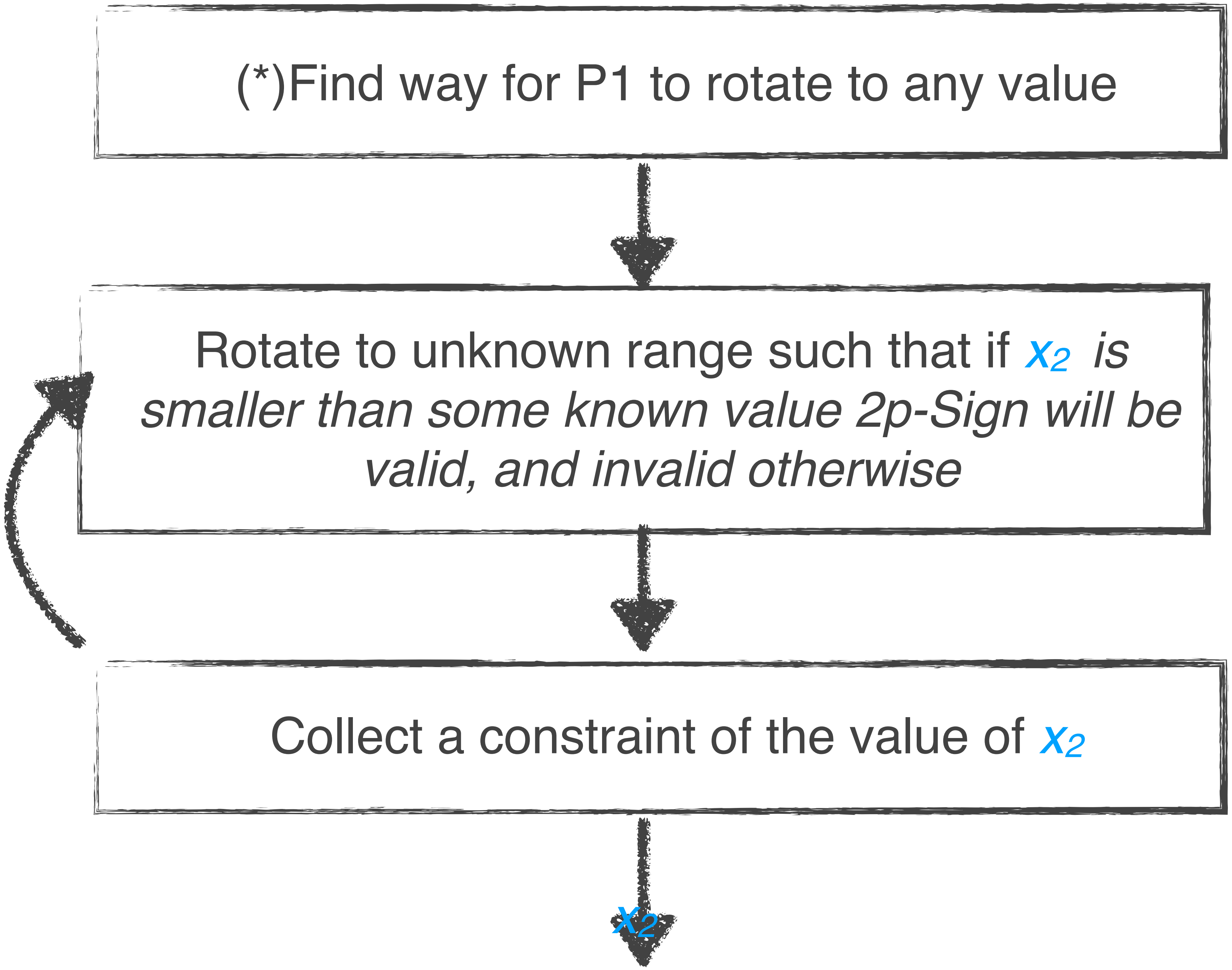
$$\boxed{c_{key\_B} = Enc_e(x_1) \boxplus r}$$

- P_1 Proves in zero knowledge that c_{key_A} and c_{key_B} encrypts the same message $x_1 + r$

- Problem: chosen zk proof works only if prover do not know factorization ➡️ P1 can cheat ?

*Appendix A: https://www.iacr.org/archive/eurocrypt2000/1807/18070437-new.pdf

Breaking 2P-Keygen



(*)Find way for P1 to rotate to any value

Rotate to unknown range such that if x_2 *is smaller than some known value 2p-Sign will be valid, and invalid otherwise*

Collect a constraint of the value of x_2

x_2

(*) Each “rotate” should cancel the coin flip r and add a big number such that verification succeed if x_2 did not invoke modulo operation on Paillier

#7 Mitigation

- We suggested that if the purpose is to skip the entire 2P-Keygen there is a general purpose for the zk proof (https://eprint.iacr.org/2016/583.pdf)
- In Practice, Re-run of 2p-keygen it is:

We thank Omer Shlomovits, Li Lin and Claudio Orlandi for reporting a vulnerability in our refresh procedure on February 10 2019. This has been fixed in the open source in this update by rerunning the Paillier ciphertext generation procedure used in key generation in every refresh.

But What If I MUST have a threshold wallet

But What If I MUST have a threshold wallet

- Understand what guarantees you get from the cryptography, what are the limits and the risks
- Map your security assumptions - try to minimise them as much as possible in comparison to the entire system
- Assume attacker has infinite resources and can do whatever she wants

But What If I MUST have a threshold wallet

- Understand what guarantees you get from the cryptography, what are the limits and the risks
- Map your security assumptions - try to minimise them as much as possible in comparison to the entire system
- Assume attacker has infinite resources and can do whatever she wants

And more:

- Hire a cryptographer
- Education, adversarial thinking, specifically for devs
- Cryptographic audits
- Battle test your code
- Programming language (e.g. Rust)
- Formal Verification

Summary

This talk was focused on the pitfalls of using threshold ECDSA in building new generation of SW wallets.

😇 Yay, Threshold Wallet!

😈 What can go wrong?

💩 I don’t care. I want it!

Questions?

https://t.me/kzen_research

https://github.com/KZen-networks