

Practical End-to-end Fault Attacks on PERK

Tanguy Stekke¹, Durba Chatterjee², Lejla Batina²

1. Université Libre de Bruxelles, Belgium

2. Radboud University, The Netherlands

Introduction to Post-Quantum Cryptography



- New call for non-lattice signatures
- New scheme definitions

PQC submissions (different families, based on different mathematical problems):

- code-based
- Isogeny-based
- Hash-based
- Lattice-based
- Multivariate Quadratic Equations

Why do we need secure implementations?

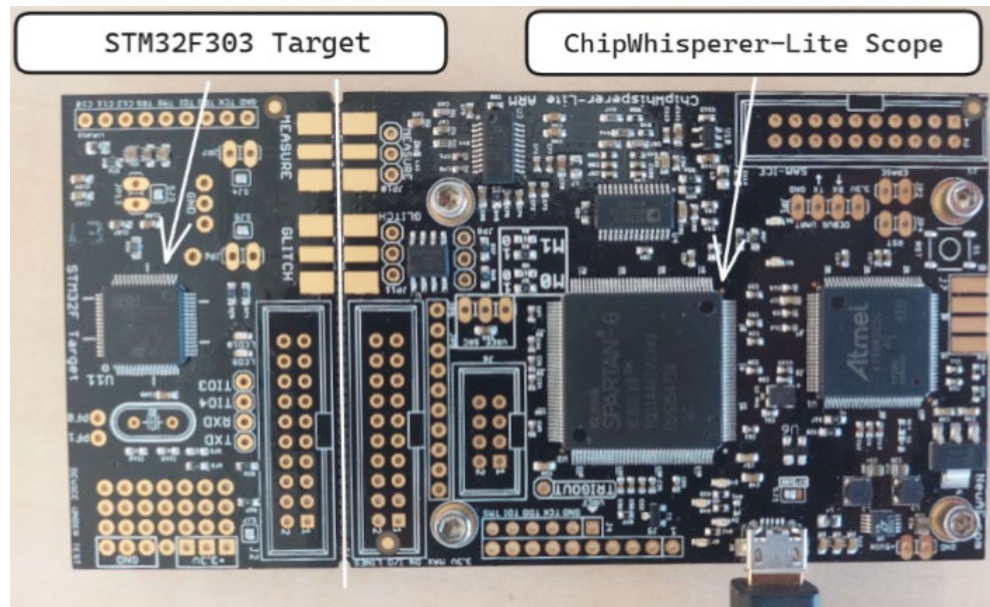
Practical Deployment Scenarios

- Implementations deployed in real-world devices
 - Embedded platforms and FPGAs
- Allows physical proximity
- Physical attacks:
 - Side channel (passive) – adversary observes output and physical emanations
 - Fault injection attacks (active) – adversary perturbs during execution.
 - Clock signal manipulation
 - Power/Voltage tampering
 - Electromagnetic signals

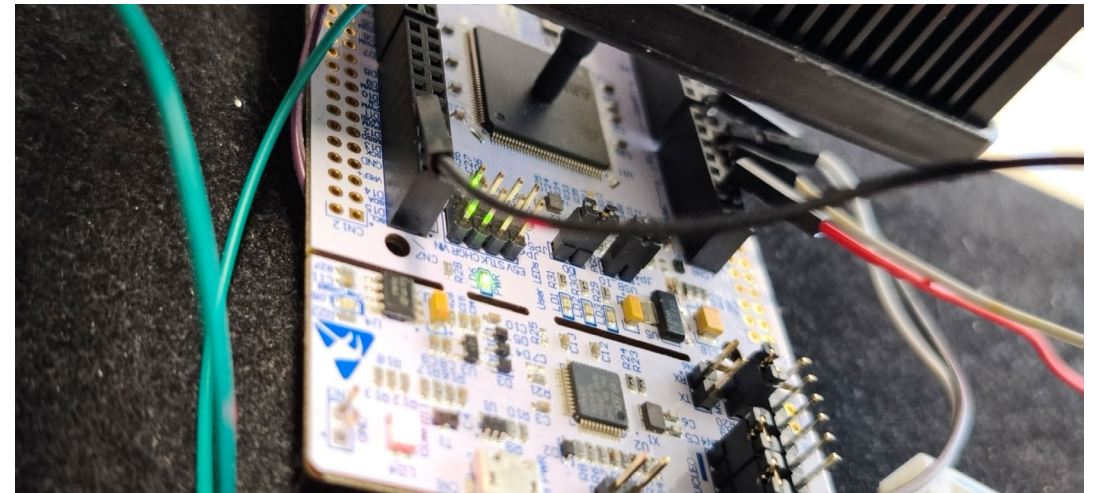
Practical Deployment Scenarios

Platforms: Software implementations on ARM Cortex-M4

Hardware Implementations on FPGAs

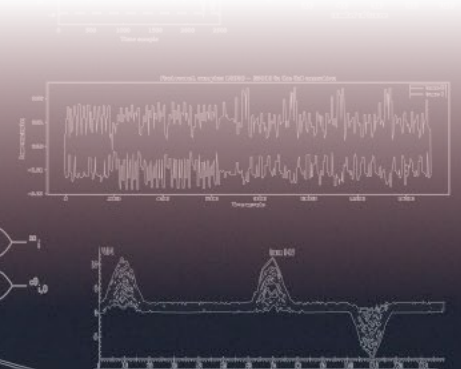
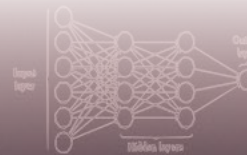


Clock/Voltage Glitching



Electromagnetic Fault Injection

What is PERK ?



PERK: PERmuted Kernal Problem

Permuted Kernel Problem

Input

- Secret permutation $\pi \in \mathcal{S}_n$
- Public matrix $\mathbf{H} \in \mathbb{F}_q^{m \times n}$ and vector $\mathbf{x} \in \mathbb{F}_q^n$ such that $\mathbf{H}(\pi[\mathbf{x}]) = \mathbf{0}$

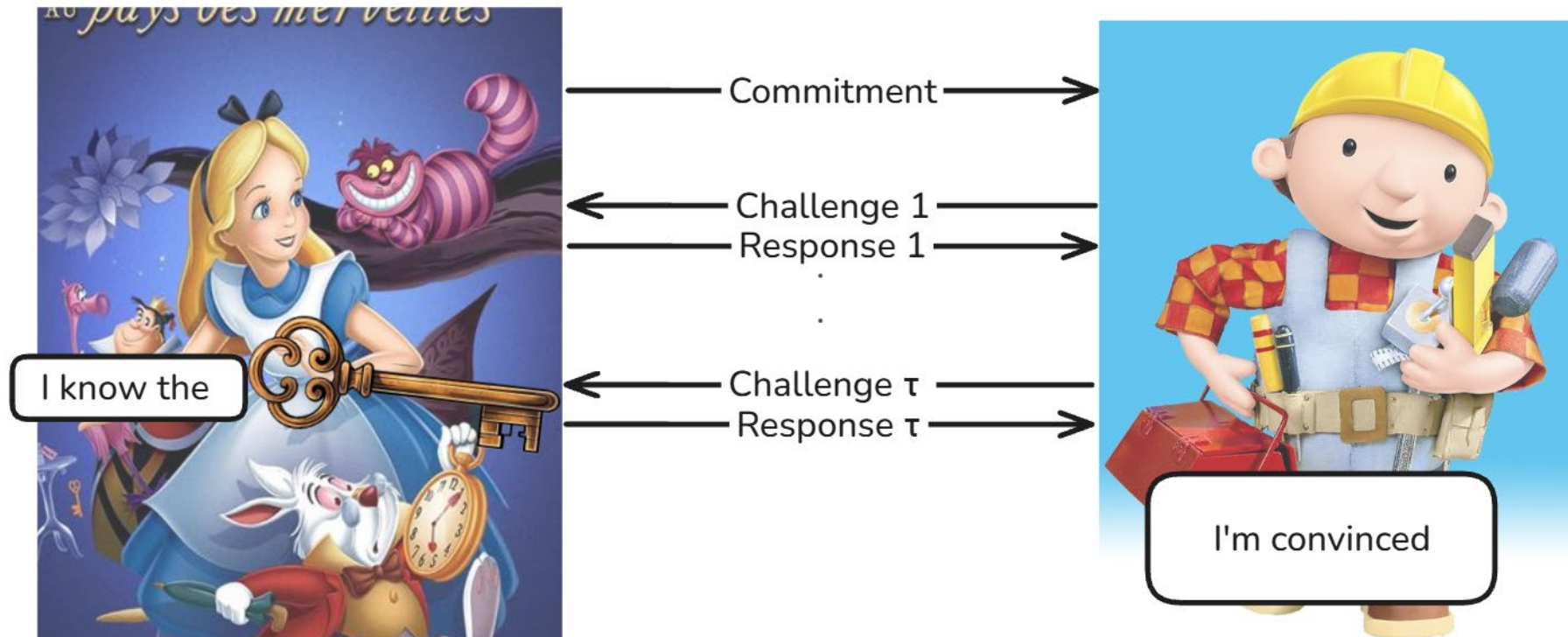
Goal

- Find $\tilde{\pi} \in \mathcal{S}_n$ such that $\mathbf{H}(\tilde{\pi}[\mathbf{x}]) = \mathbf{0}$

- For the remainder of the discussion, $\boldsymbol{\pi}$ can be treated as the key, given that recovering $\boldsymbol{\pi}$ is sufficient to enable message forgery

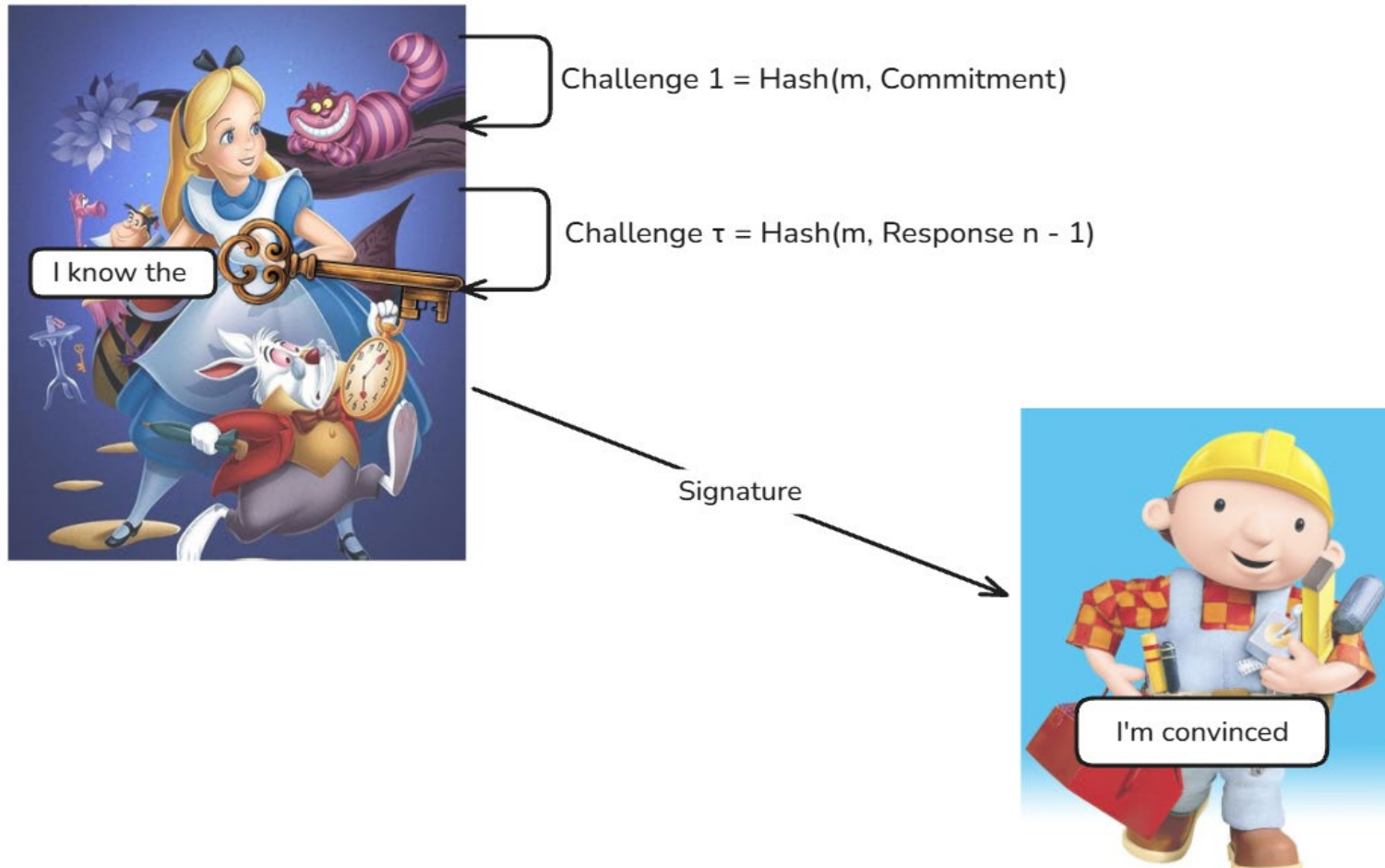
Proving you know π without revealing it

- Alice wants to convince Bob she knows the secret permutation π , without leaking any information about it.
- Use the 3-moves structure: commit, challenge, response
- This can be repeated many times until the verifier is statistically convinced.

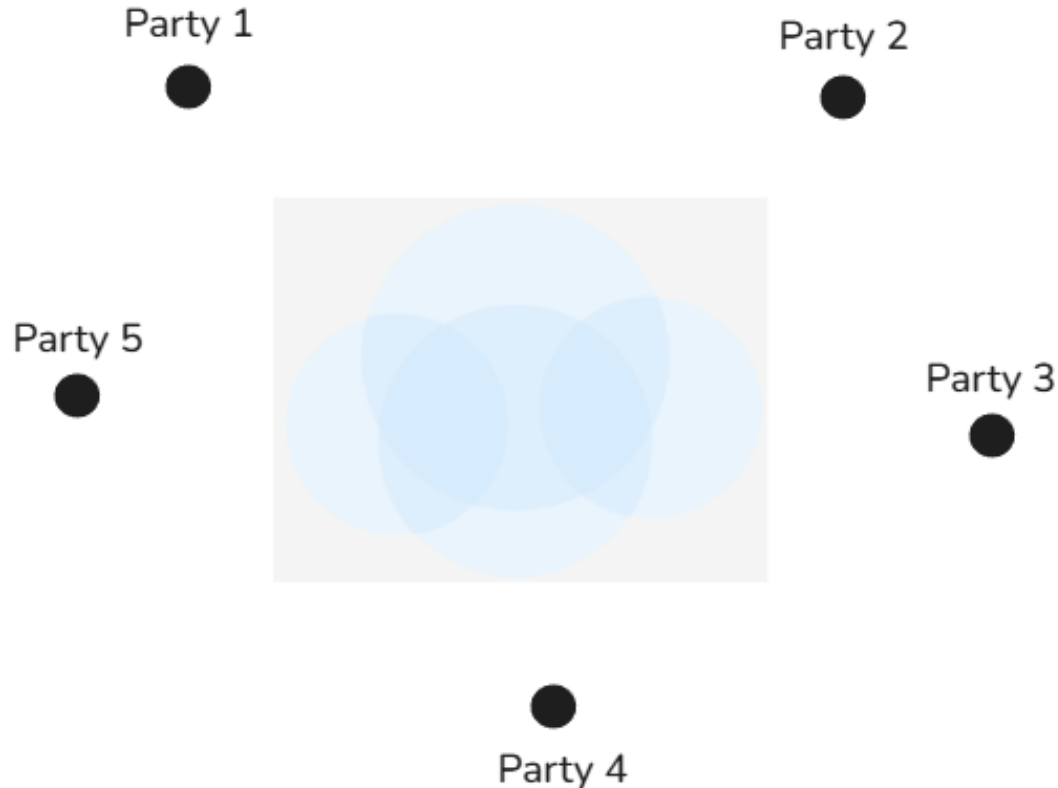


Zero-Knowledge Proofs

Instead of having Alice and Bob the scheme is made non-interactive using Fiat-Shamir, Alice computes everything and sends results to BOB



Multi-Party Computation in the Head

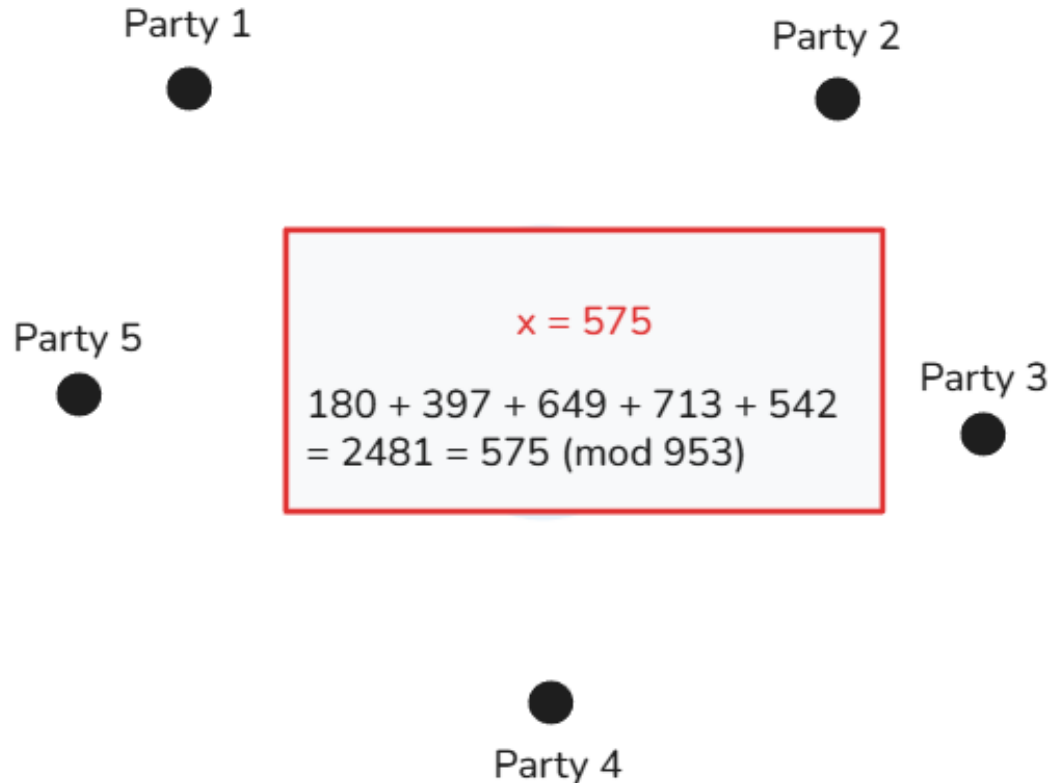
**1**

- Public: base $z=3$
target $y = 1467 \bmod 1907$
- Secret: exponent $x = 575$
with $y = z^x$
- Goal: check that $y = z^x$ without
party learning x

Public: $z=3, y=1467 \bmod 1907$

Secret: $x=575$

Multi-Party Computation in the Head

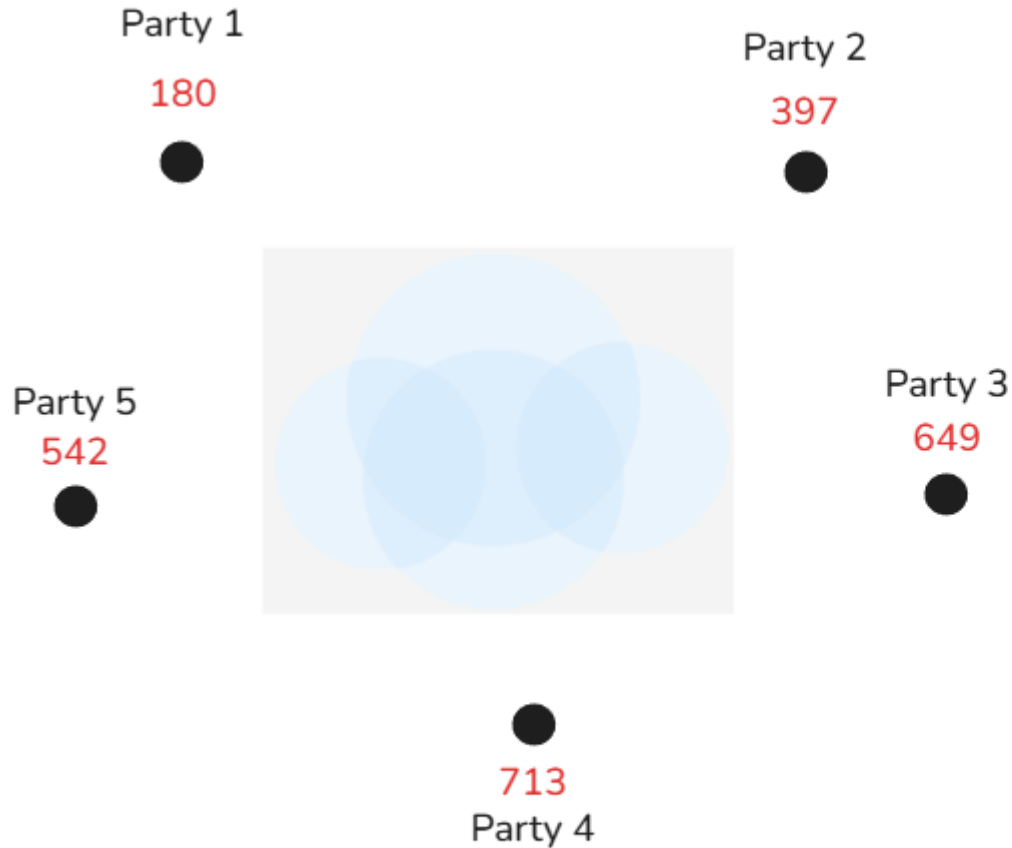
**2**

- x is cut into 5 random looking shares
- The shares add up to x modulo 953
- One share alone reveals nothing about x

Public: $z=3, y=1467 \pmod{1907}$

Secret: $x=575$

Multi-Party Computation in the Head



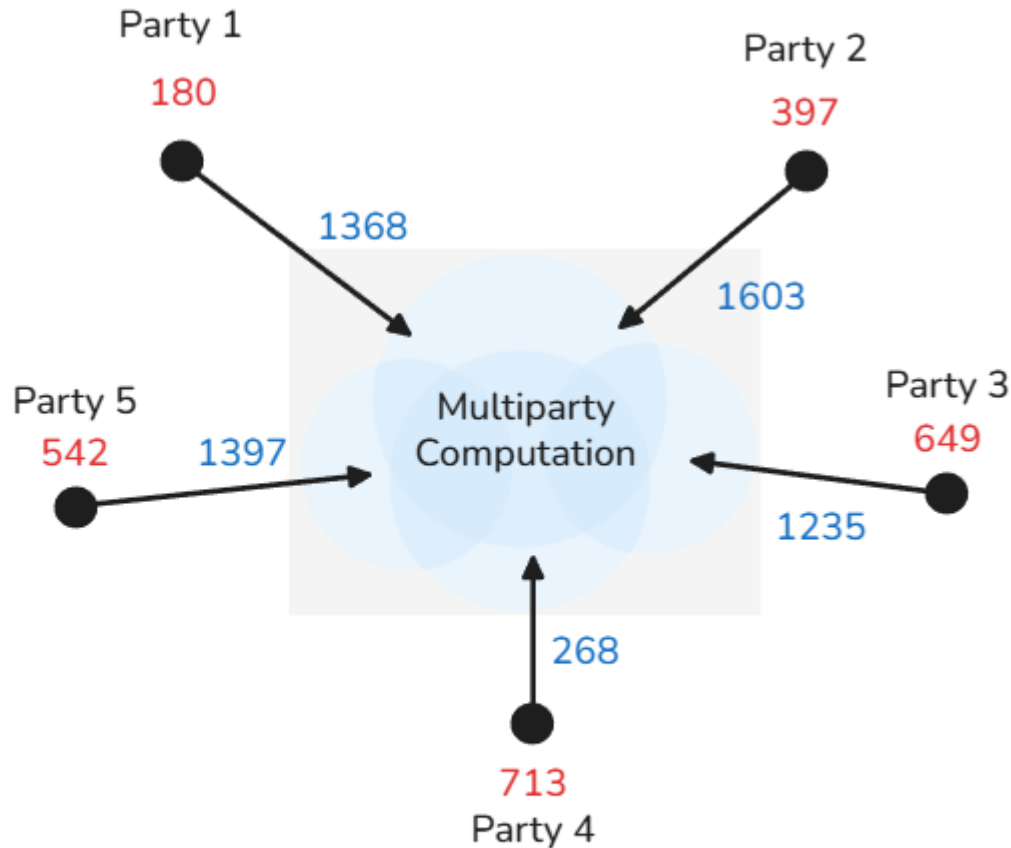
3

- Party receives only its own share
- No party knows x on its own
- Only all 5 shares together give x

Public: $z=3, y=1467 \pmod{1907}$

Secret: $x=575$

Multi-Party Computation in the Head



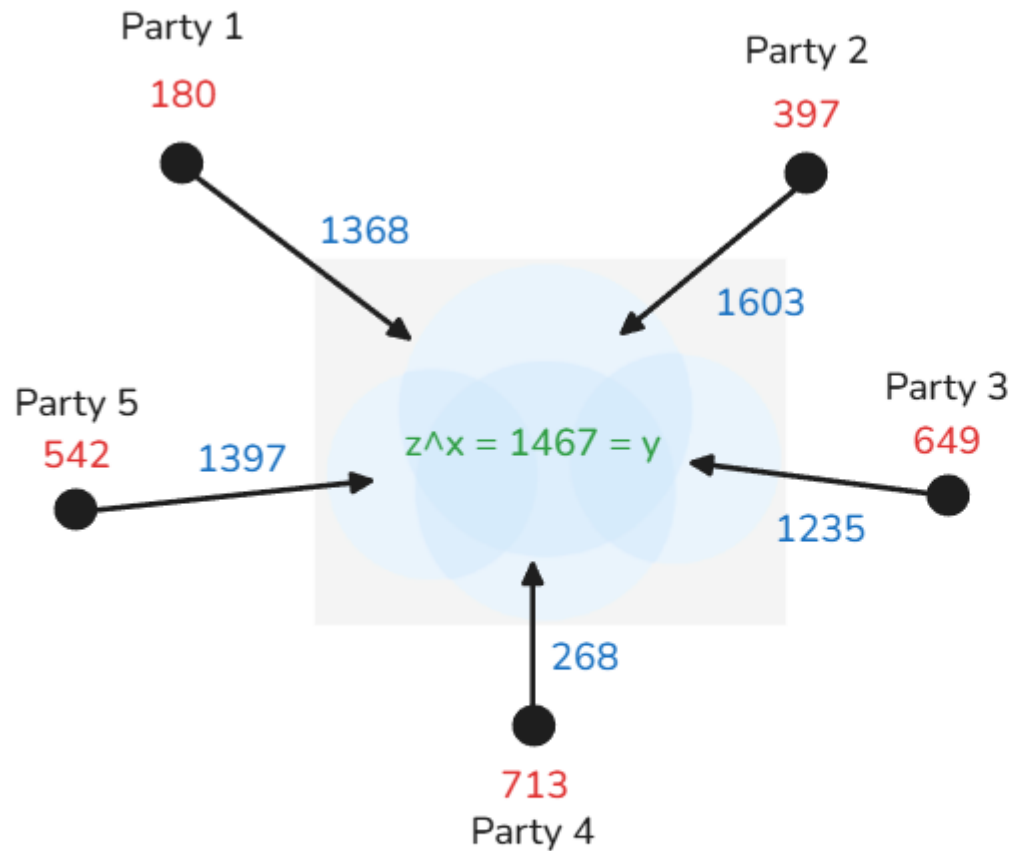
4

- Party i computes z^{share}
- No party knows x on its own

Public: $z=3, y=1467 \pmod{1907}$

Secret: $x=575$

Multi-Party Computation in the Head



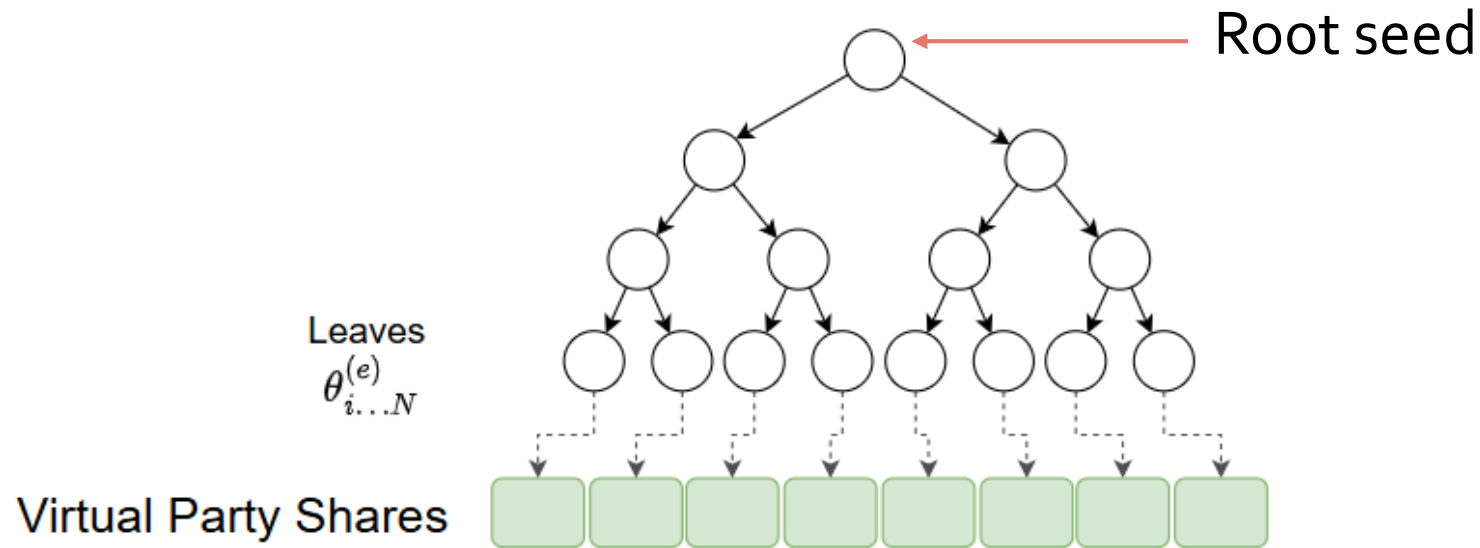
5

- Multiply broadcast values mod 1907
- Product is correct so accepted

Public: $z=3, y=1467 \pmod{1907}$

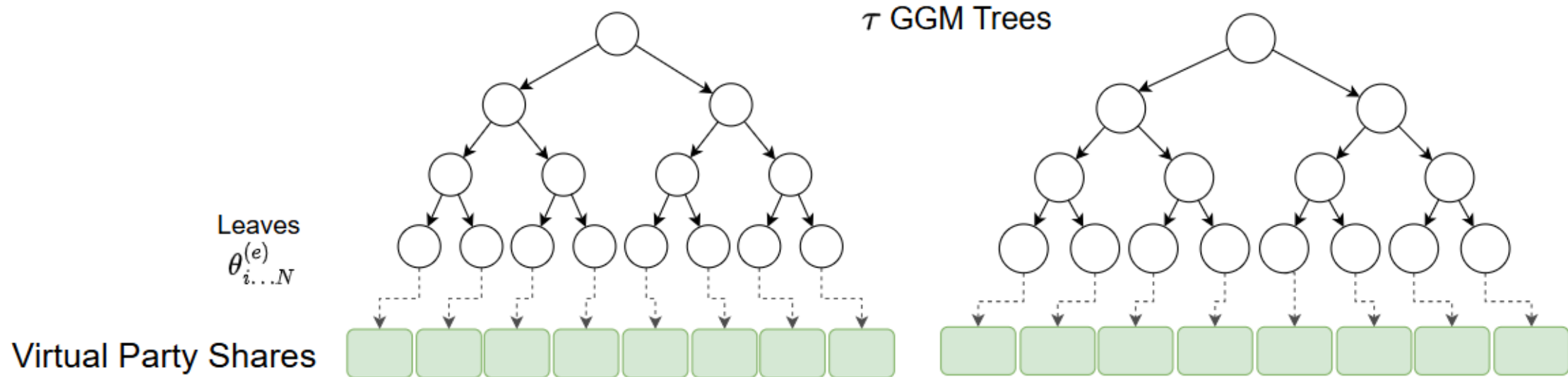
Secret: $x=575$

GGM Trees



- Used to make it suitable for embedded platforms

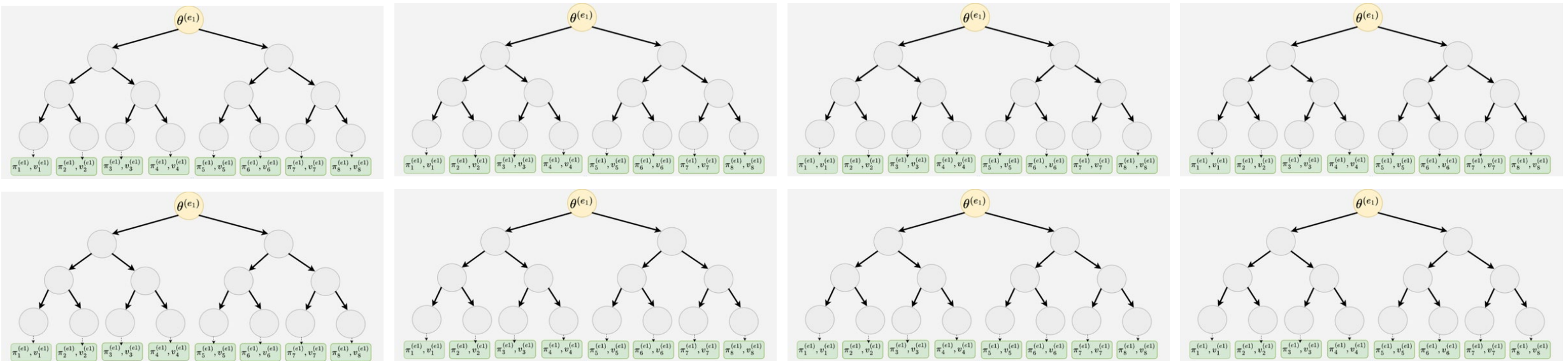
GGM Trees



- Used to make it suitable for embedded platforms

GGM Trees

1. Commitment: Salt and master seed to derive the τ trees

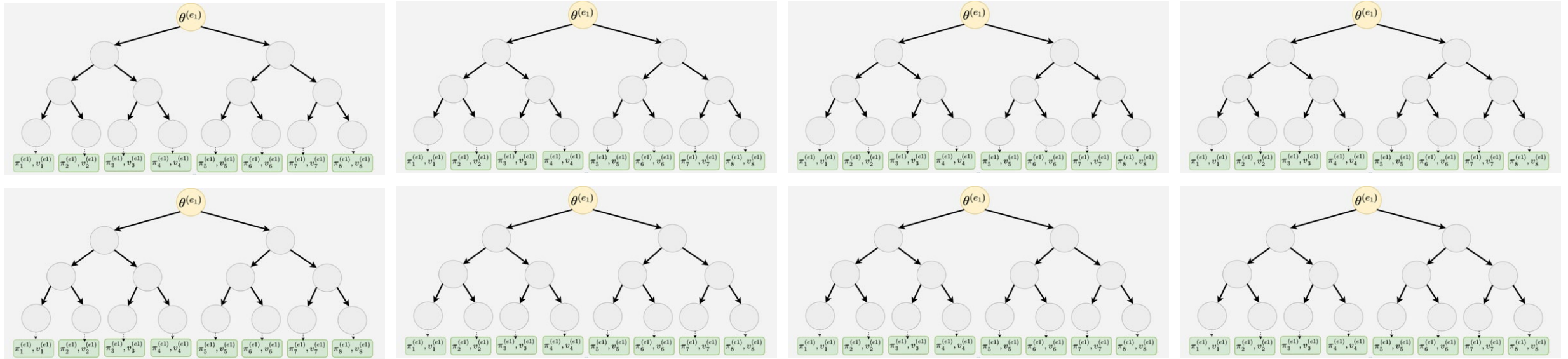


Commit hash(Shares, Cmt)



GGM Trees

2. First challenge, t coefficients for a linear combinations



3. First response

Commit hash(1Challenge, States)

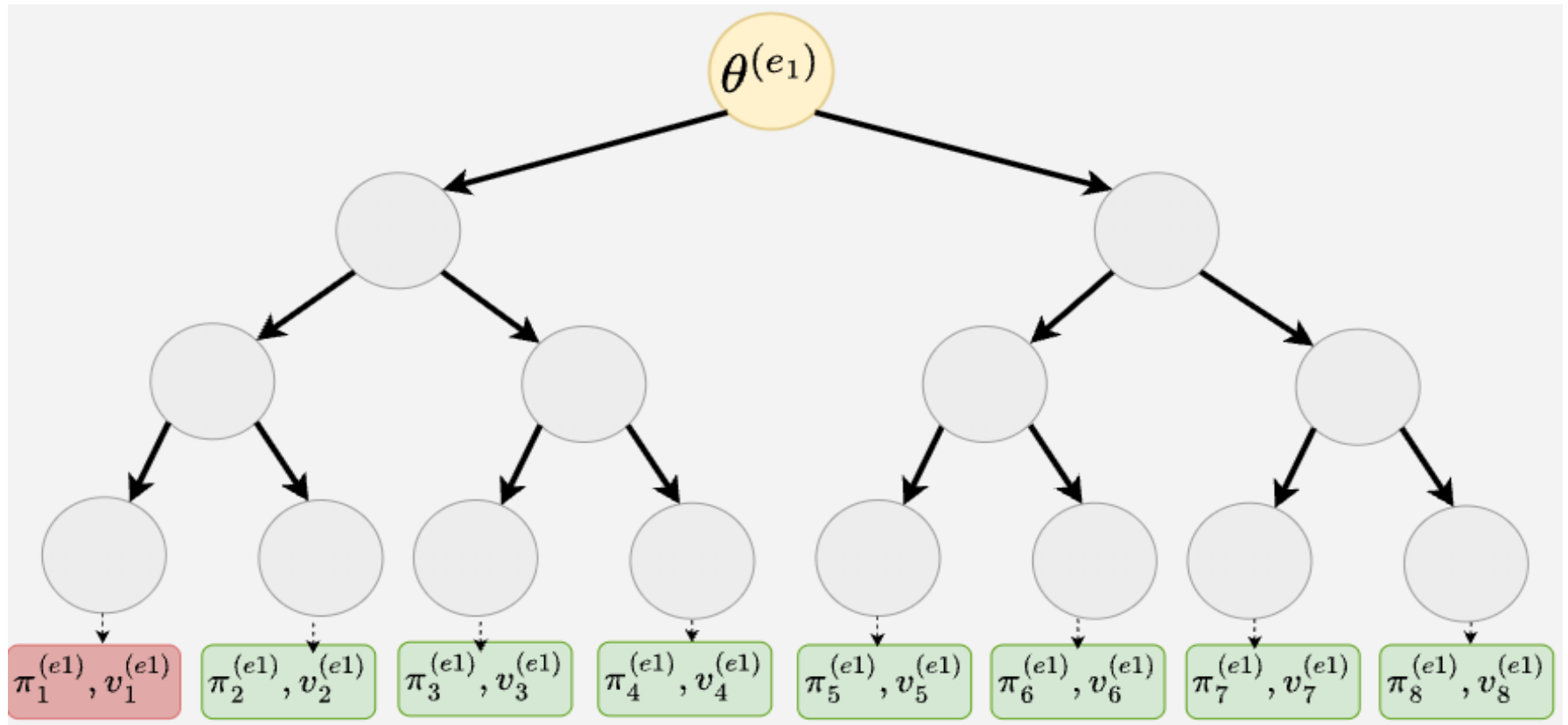


$$s_i^{(e)} = \pi_i^{(e)}[s_{i-1}^{(e)}] + v_i^{(e)}.$$



GGM Trees

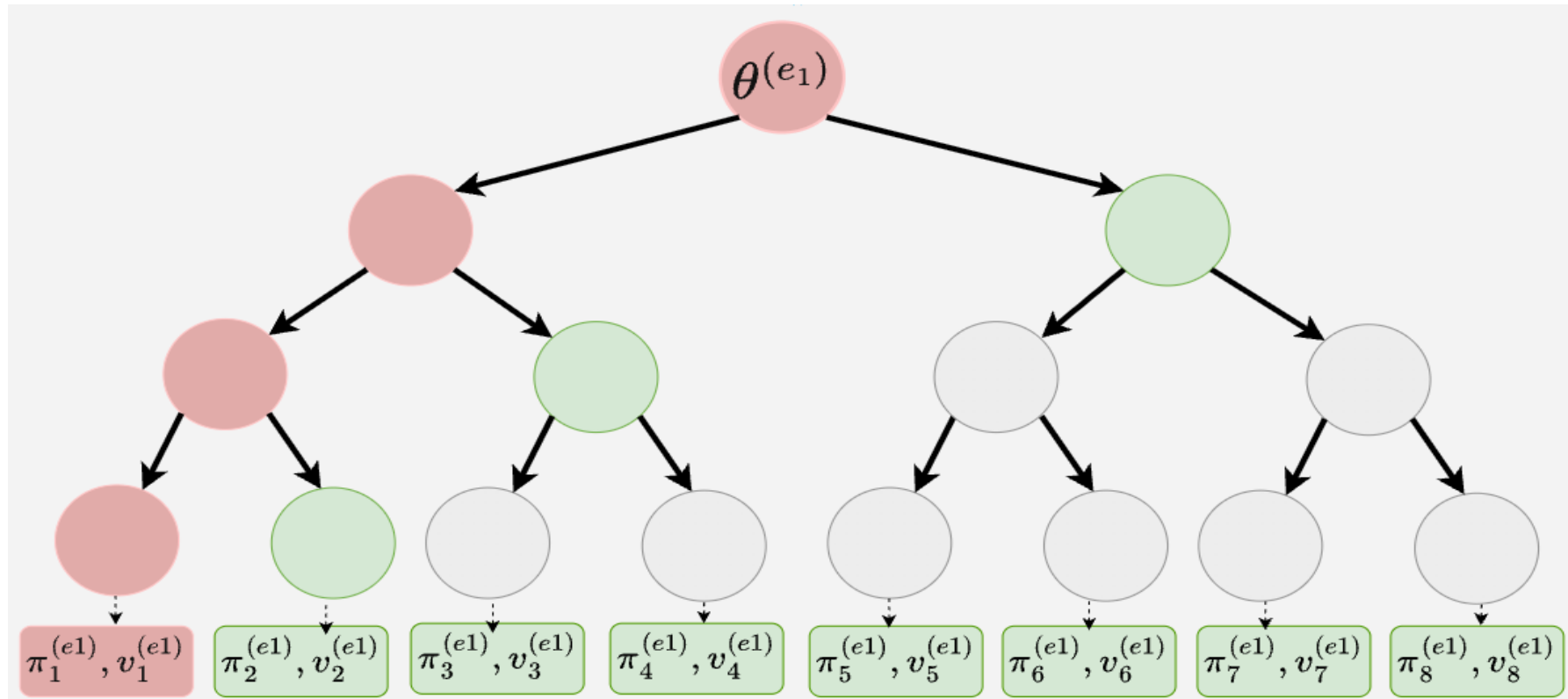
4. Second challenge



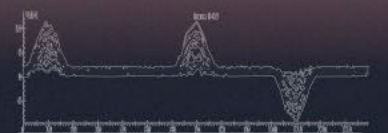
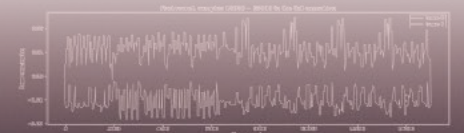
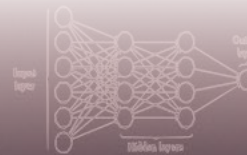
GGM Trees

5. Second response

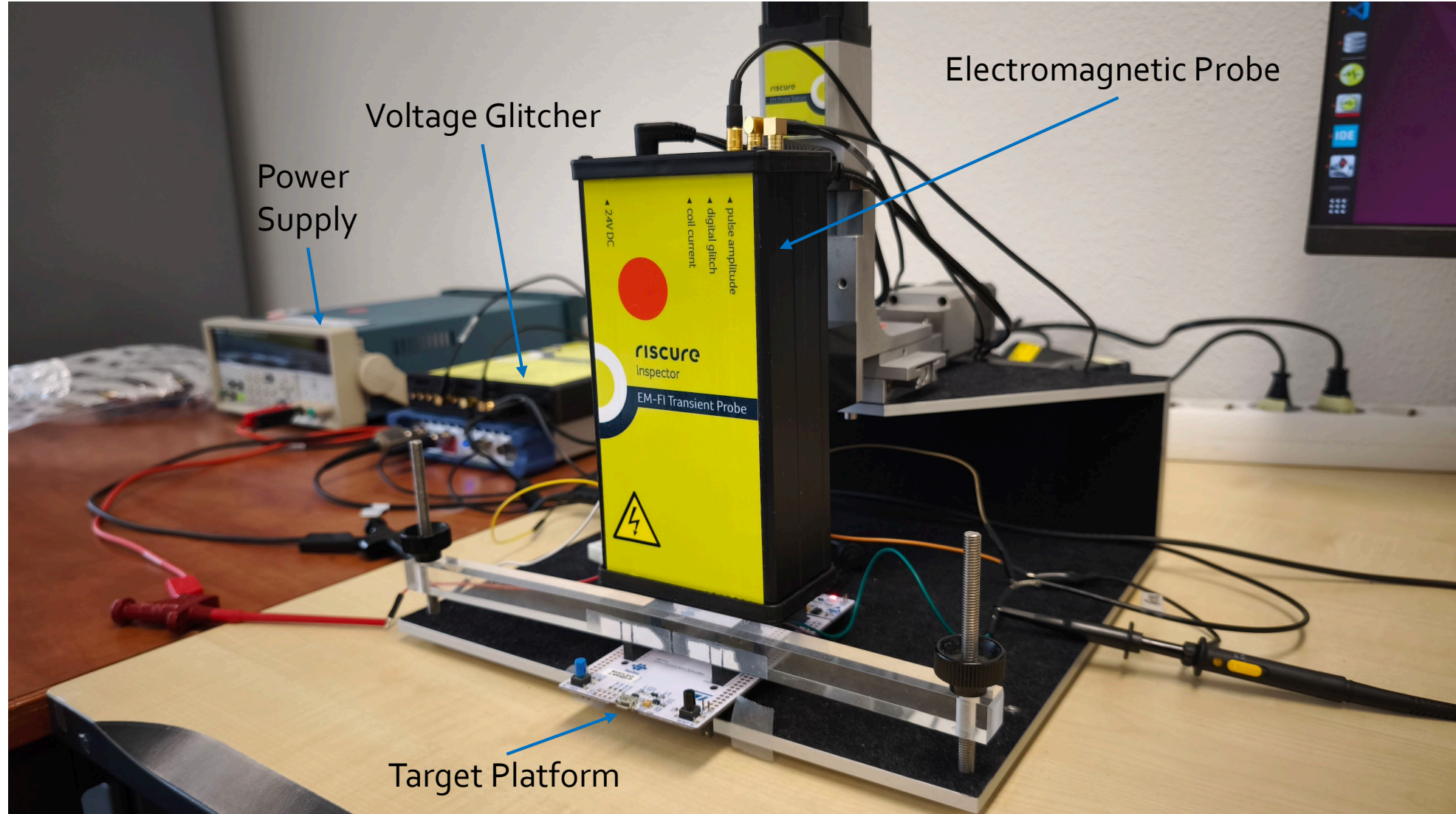
- Send everything (commitment, state, seeds) but the α which is the hidden view



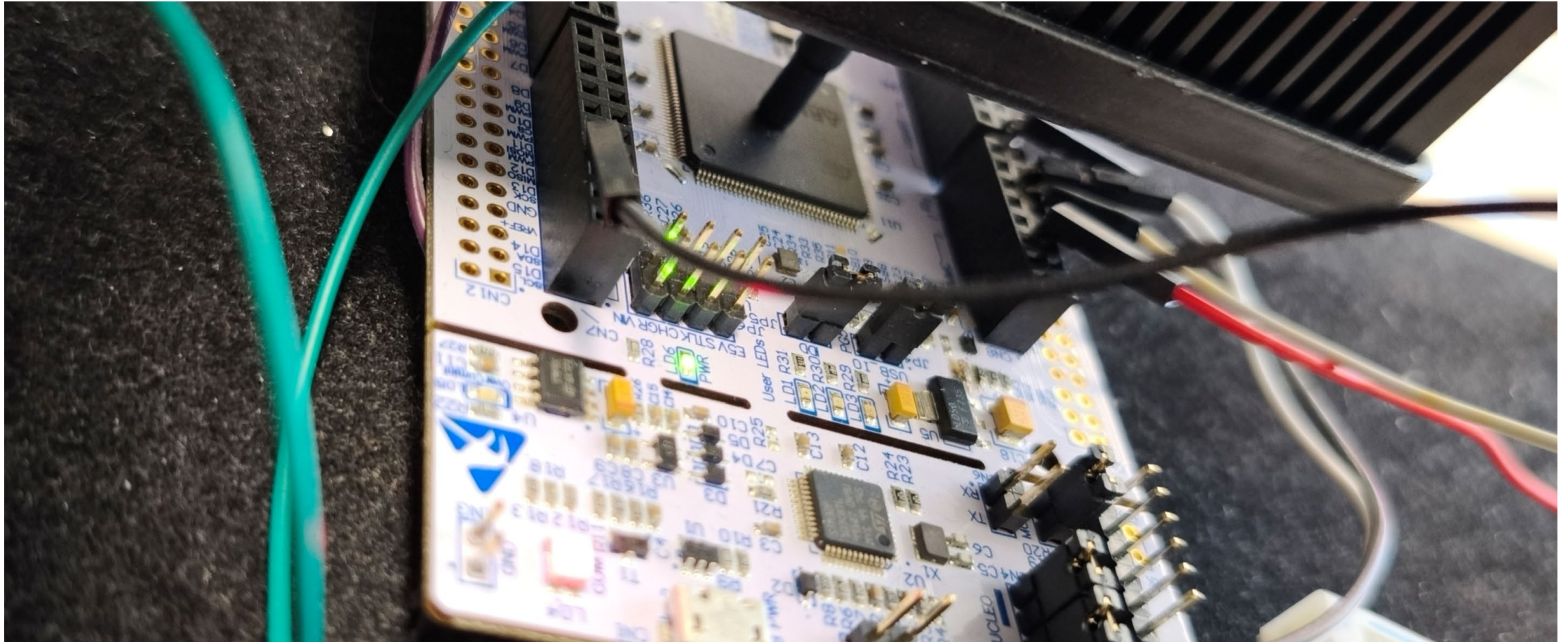
Setup and Attacks



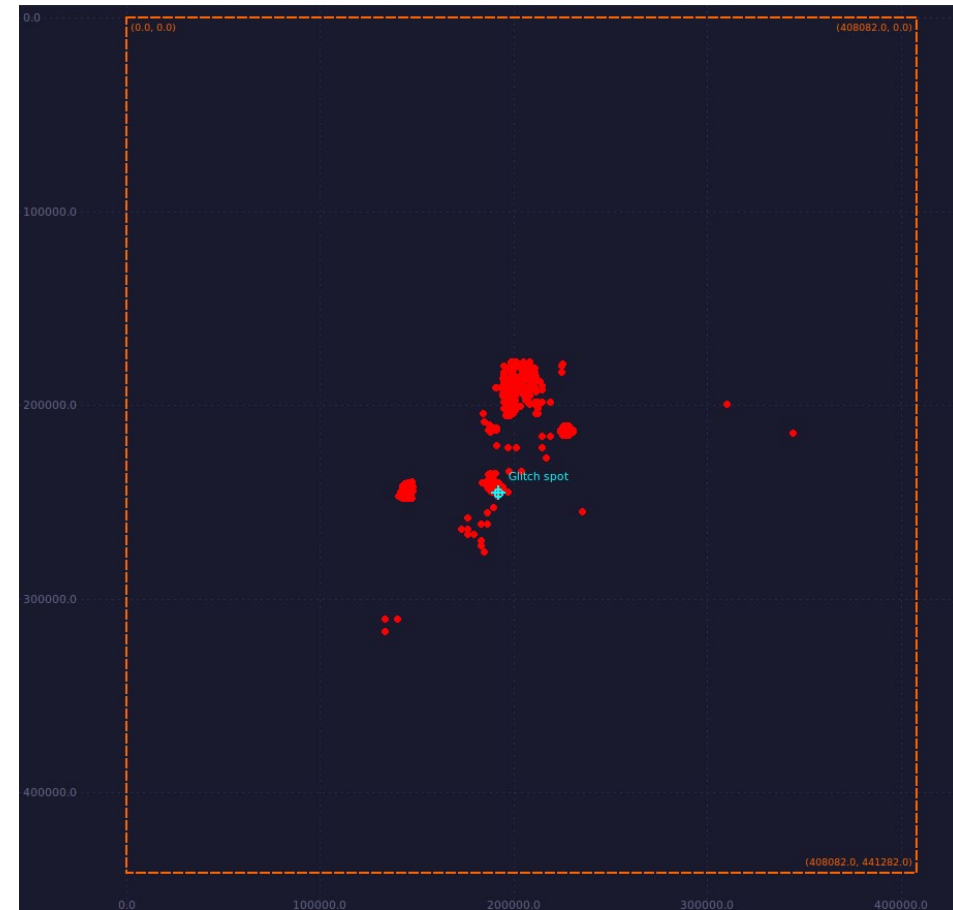
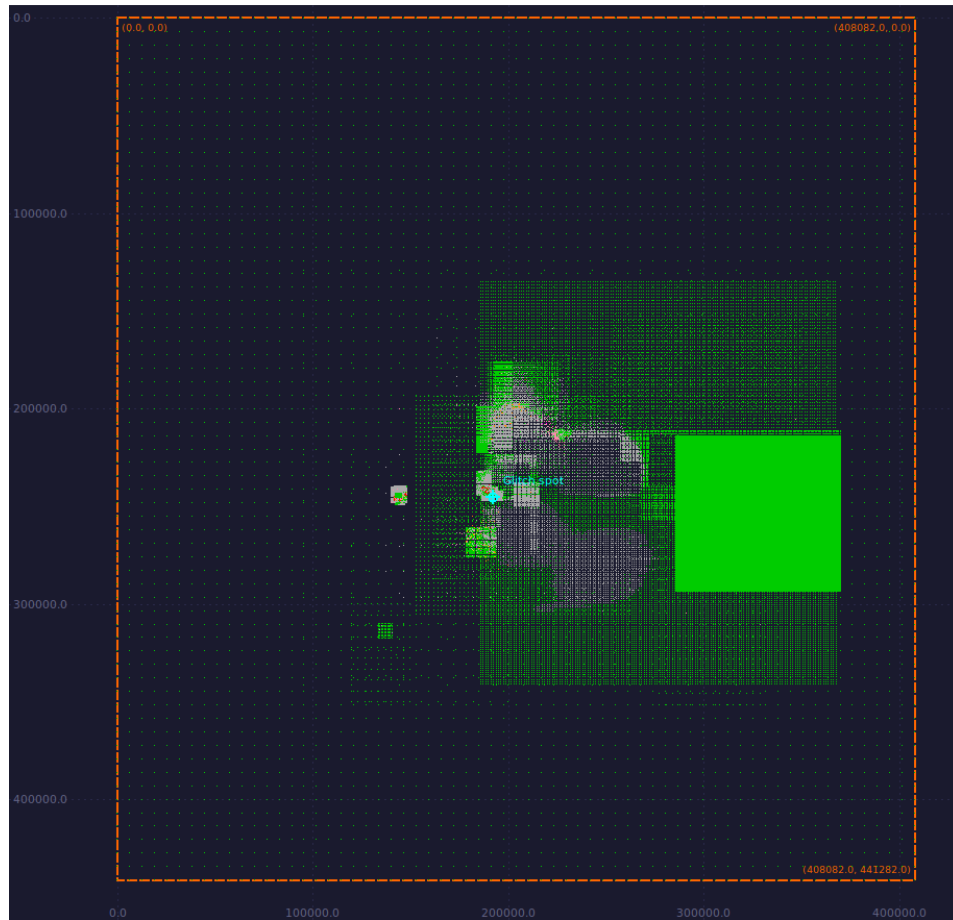
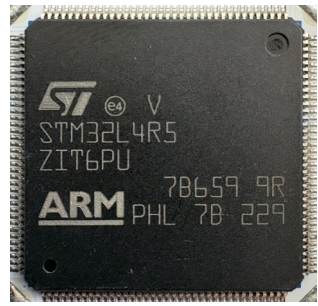
Electromagnetic Fault Injection Setup



Electromagnetic Fault Injection Setup (target)



Spatial Characterization



Attack 1: Fixing Root Seed

PERK Signing Steps

- ➔ 1. Commitment
- 2. First Challenge
- 3. First Response
- 4. Second Challenge
- 5. Second Response

Attack 1: Fixing Root Seed

```

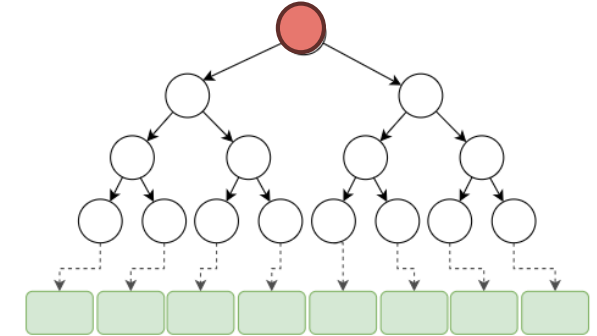
void sig_perk_gen_commitment(...){
    uint8_t rand_bytes[SEED_BYTES + SALT_BYTES] = {0};
    sig_perk_prg_state_t prg_state;

    sig_perk_perm_inverse(pi_inv, pi);
    randombytes(rand_bytes, sizeof(rand_bytes));
    memcpy(mseed, rand_bytes, SEED_BYTES);
    memcpy(signature->salt, rand_bytes + SEED_BYTES, SALT_BYTES);

    sig_perk_prg_init(&prg_state, PRG1, signature->salt, mseed);

    for (uint8_t e = 0; e < PARAM_TAU; ++e) {
        perk_theta_seeds_tree_t theta_tree;
        trigger_high();
        sig_perk_prg(&prg_state, theta_tree[0], sizeof(theta_t));
        trigger_low();
    }
}

```



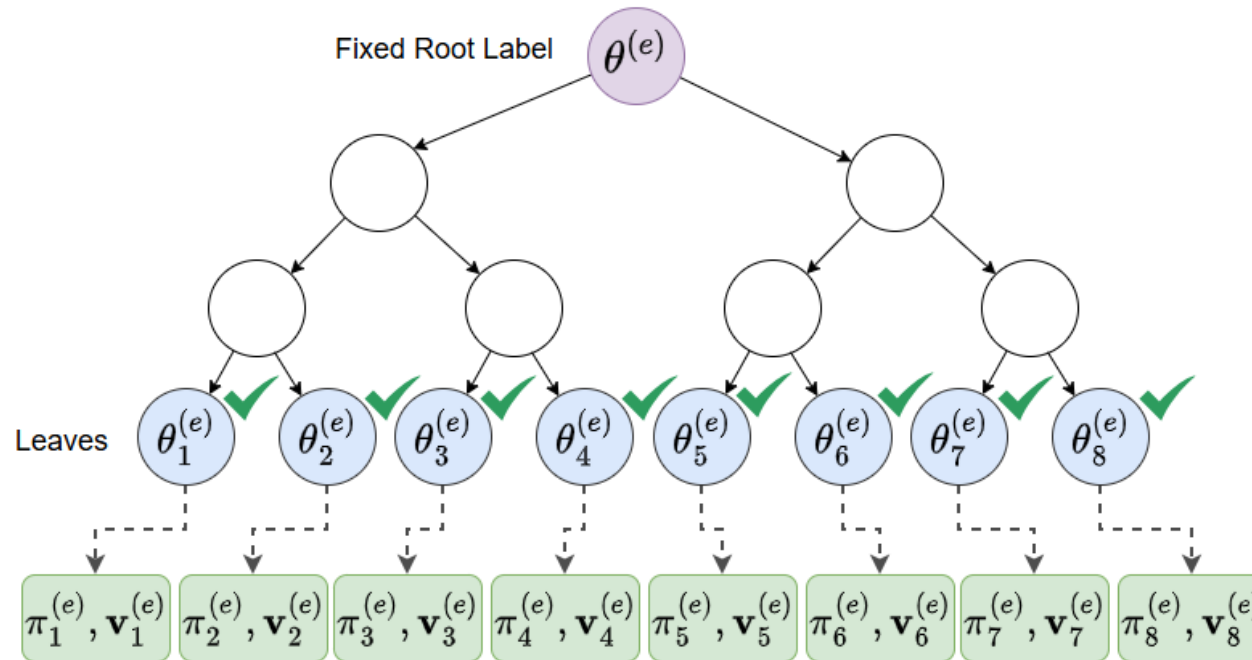
salt = 0x0 mseed = 0x0

```

1  movs    r1, 0x0           ; memset value = 0
2  movs    r2, 0x30          ; size = 48 bytes
3  add     r0, sp, 0x38       ; r0 points to rand_bytes buffer
4  bl      memset            ; memset(rand_bytes, 0, 48)
5
6  add     r0, sp, 0x38       ; r0 = pointer to rand_bytes buffer
7  bl      PQCLEAN_randombytes ; call randombytes(r0, r1)

```

Attack 1: Fixing Root Seed



- Targets the root seed.
- Enabled due to implementation specifics

Attack 2: Node Collision

$$(\theta^{(e)})_{e \in [1, \tau]} \leftarrow \text{PRG}(\text{salt}, \text{mseed})$$



Attack 2: Node Collision

NIST IMPLEMENTATION

-  1.Commitment
-  2.First Challenge
-  3.First Response
-  4.Second Challenge
-  5.Second Response

PQM4 Implementation

-  1.Commitment
 -  2.First Challenge
 -  3.First Response
-
-  4.Second Challenge
 -  5.Second Response

Recomputed

Attack 2: Node Collision

PQM4 Implementation



1. Commitment



2. First Challenge



3. First Response



4. Second Challenge

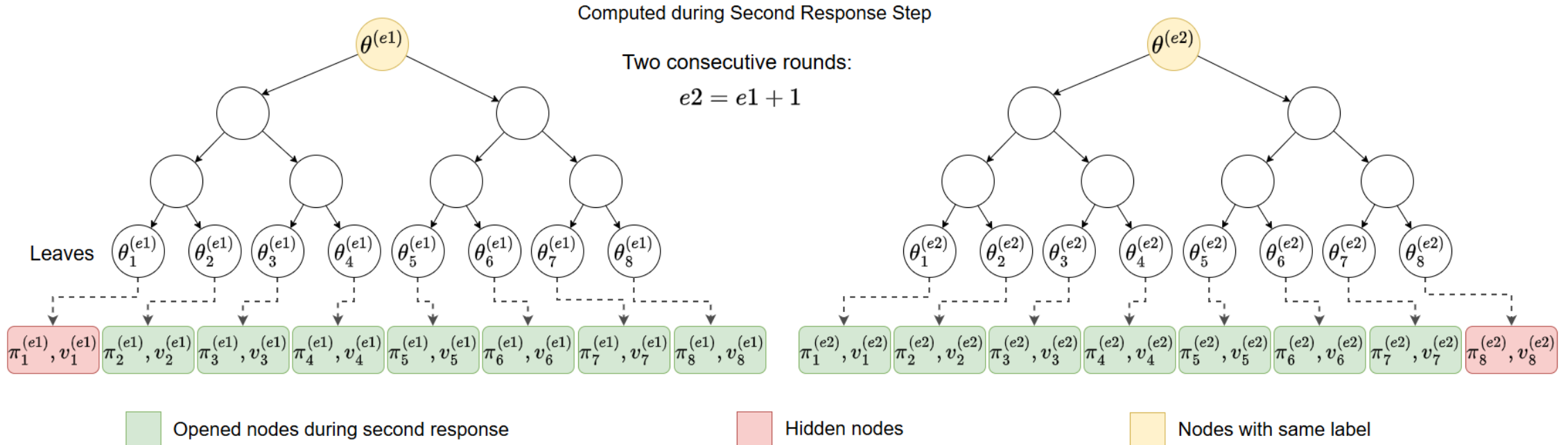


5. Second Response

Attack 2: Node Collision

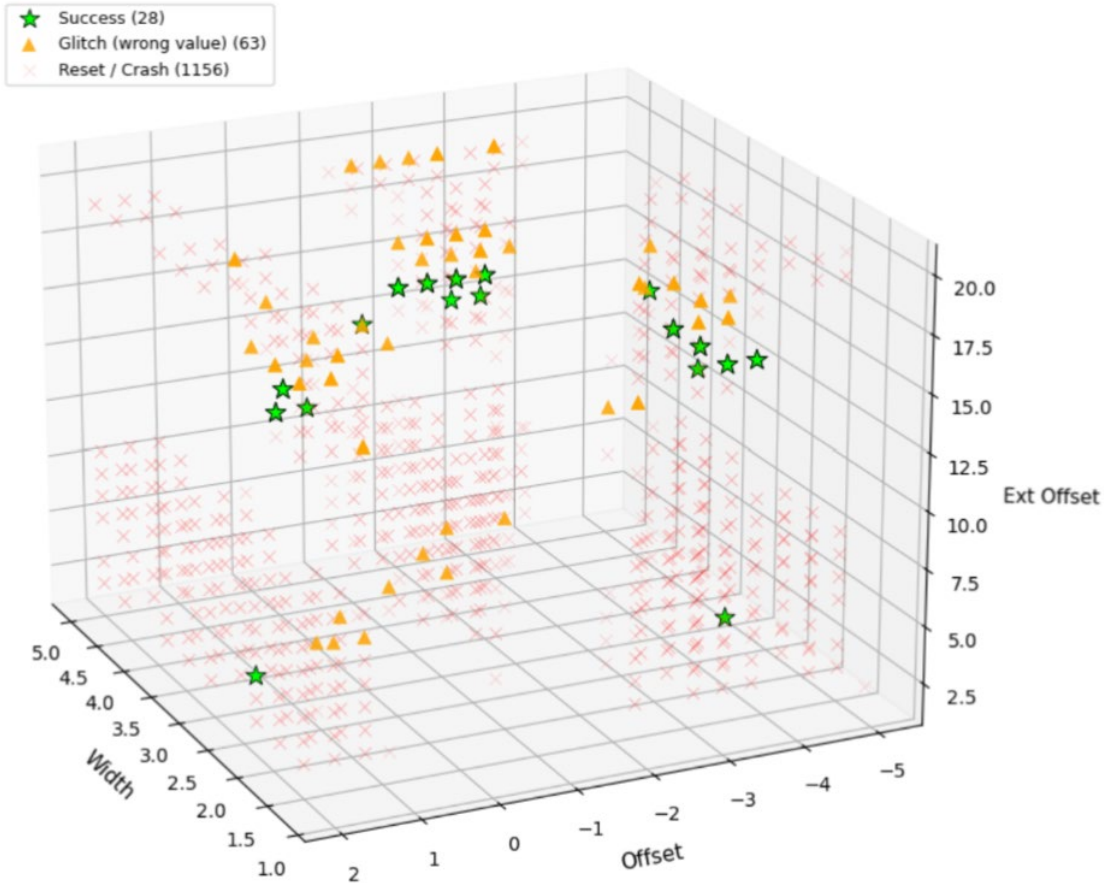
```
1 void sig_perk_gen_second_response(){
2     for (uint8_t e = 0; e < PARAM_TAU; ++e) {
3         vect1_t tmp; vect1_t v_i; vect1_t s_0; vect1_t s_i;
4         perk_theta_seeds_tree_t theta_tree;
5         const uint16_t alpha = challenges[e].alpha;
6         sig_perk_vect1_mult_scalar_vect(s_0, challenges[e].kappa[0], x[0]);
7         for (int j = 1; j < PARAM_T; j++) {
8             sig_perk_vect1_mult_scalar_vect(tmp, challenges[e].kappa[j], x[j]);
9             sig_perk_vect1_add(s_0, s_0, tmp);
10        }
11        if (e==1) {
12            trigger_high();
13            sig_perk_prg(&prg_state, theta_tree[0], sizeof(theta_t));
14            trigger_low();
15        }else{
16            sig_perk_prg(&prg_state, theta_tree[0], sizeof(theta_t));
17            ...
18    }
```

Attack 2: Node Collision



- Feasible due to the computations performed in a loop.
- Same register used for the same intermediates

Attack Validation



- Both attacks require only one successful fault.

Attack 2 success rate:

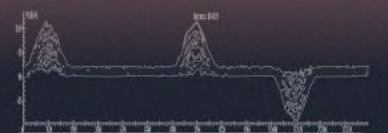
Outcome	Count
Successful key recovery	85
Target non-responsive	13
Challenge collision ($\alpha_{e_1} = \alpha_{e_2}$)	2

Success rates:

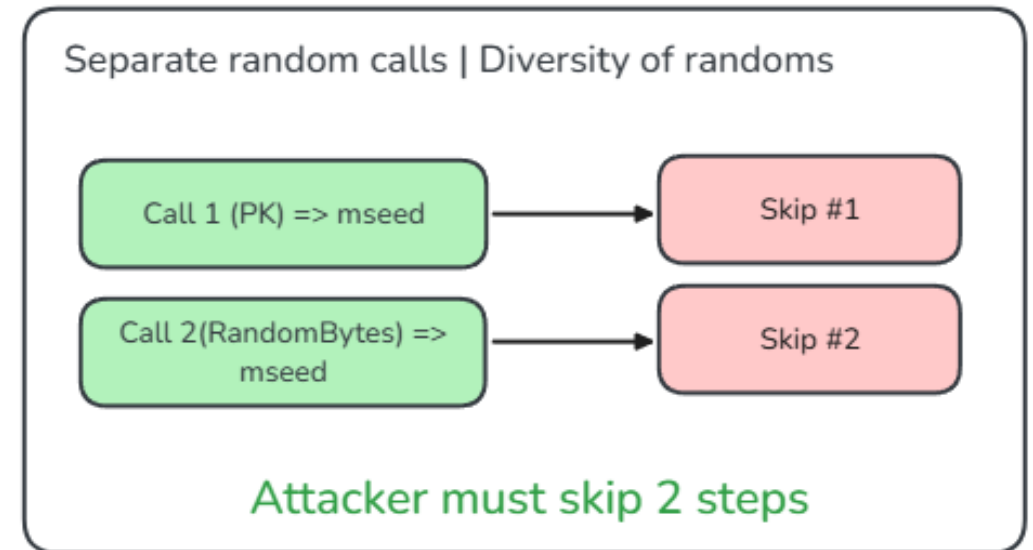
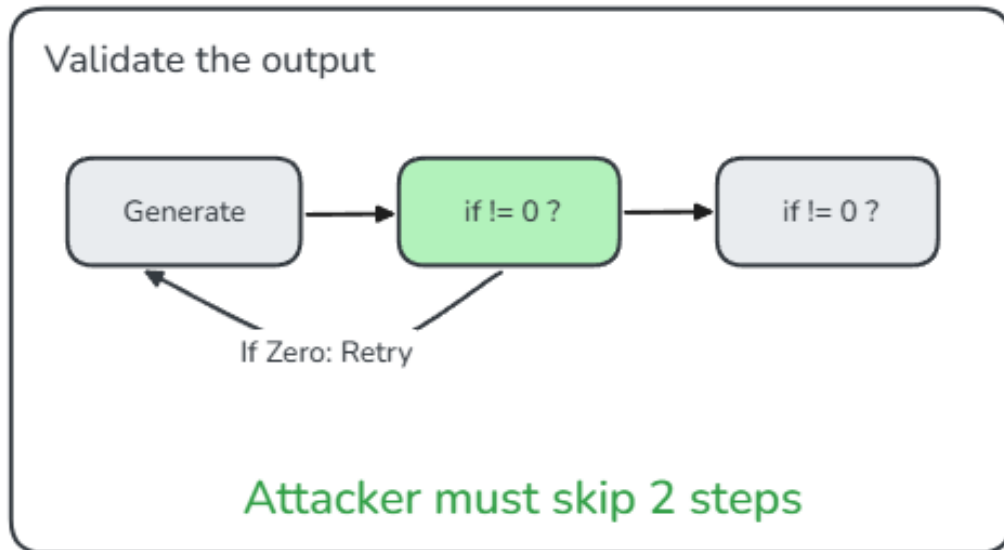
Attack 1: 100%

Attack 2: 85%

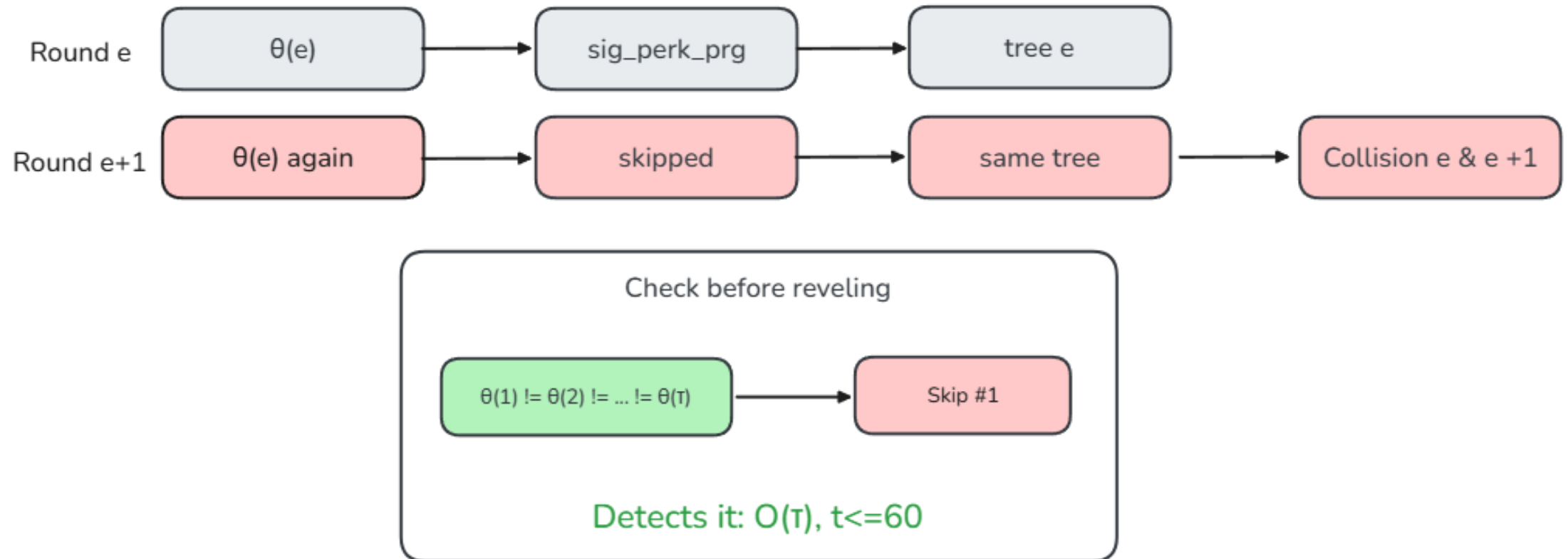
Countermeasures



Countermeasures for Attack 1



Countermeasures for Attack 2



Key Take-aways

- Securing Implementations is very different from developing secure algorithms
- Physical security evaluation is essential for PQC deployment.
- Memory optimizations can introduce implementation vulnerabilities enabling physical attacks.
- Implementation-level countermeasures required.

Thank you

Questions?

