Attacking and Protecting SLH-DSA against Fault Injections

Thomas Prest (joint work with Adrian Thillard)

PQShield (Paris, FR)

Deployment of post-quantum cryptography (11/10/2024)

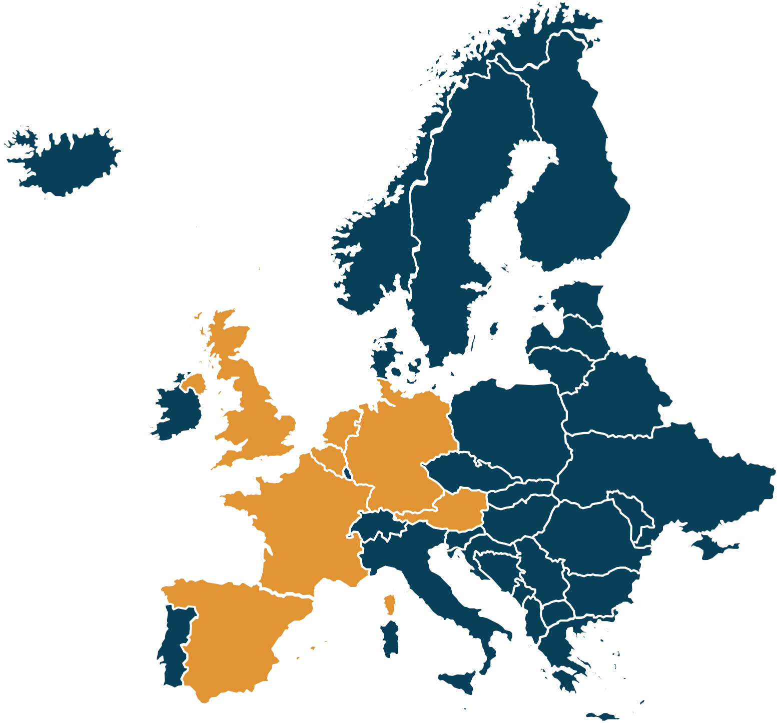
PQShield

Who are we?

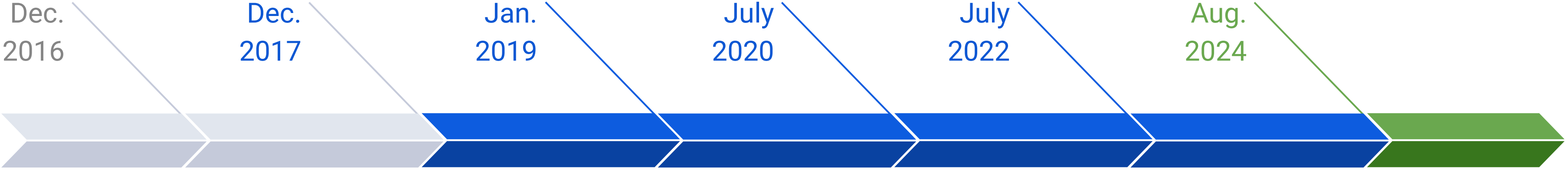
- A (mainly) European start-up specialised in post-quantum cryptography
  - Also present in Japan, USA, etc.
  - 70+ employees, with 20+ PhDs in PQC/implementation/security
- We provide:
  - Libraries (SW/HW)
  - SCA countermeasures
  - Expertise in various PQC topics

Who am I?

- Thomas Prest, Head of Research
  - Research Team
  - Paris office (come say hi!)



NIST standardisation



Dec. 2016

NIST Call for proposals

Dec. 2017

Round 1
69 candidates

Jan. 2019

Round 2
26 candidates

July 2020

Round 3
15 candidates

July 2022

Standards announced
- ML-KEM
- ML-DSA
- SLH-DSA
- FN-DSA

Aug. 2024

Standards published
- FIPS 203
- FIPS 204
- FIPS 205

Hash-based signatures?

Principle: build a signature scheme using generic properties of cryptographic hash functions

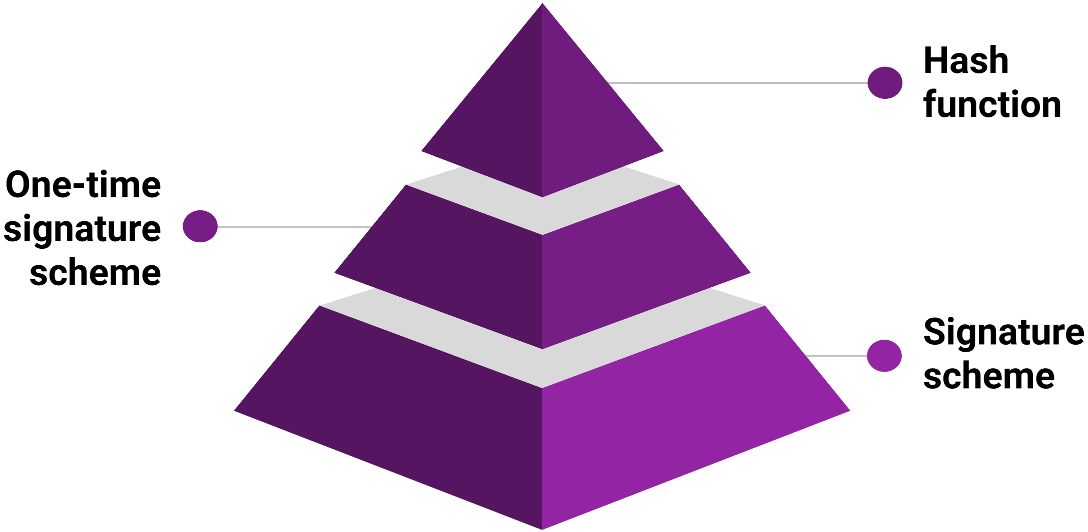
Pros:

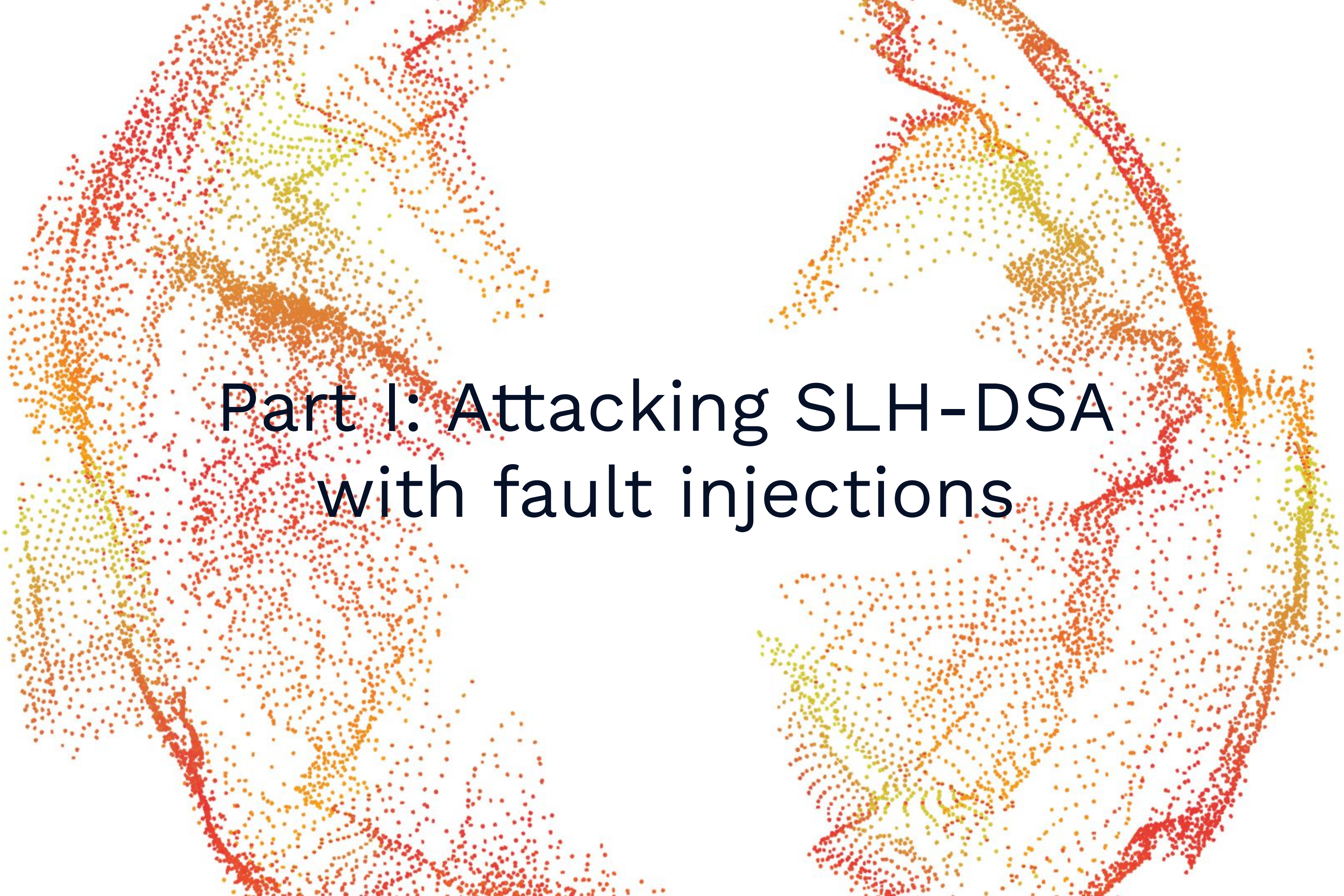
- \+ Compelling and elegant idea (the hash function is a black box)
- \+ Strong security guarantees
- \+ Post-quantum

Cons:

- \- Can get complicated
- \- Large signature size
- \- Slow signing

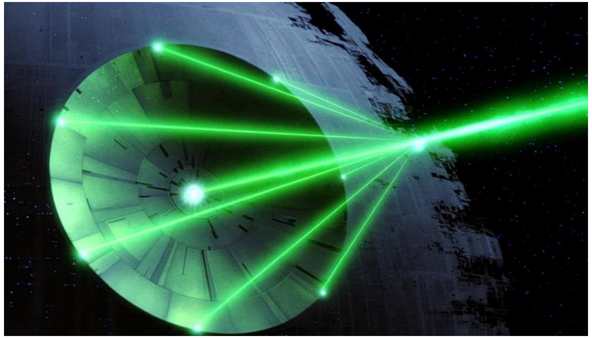
What about fault tolerance?



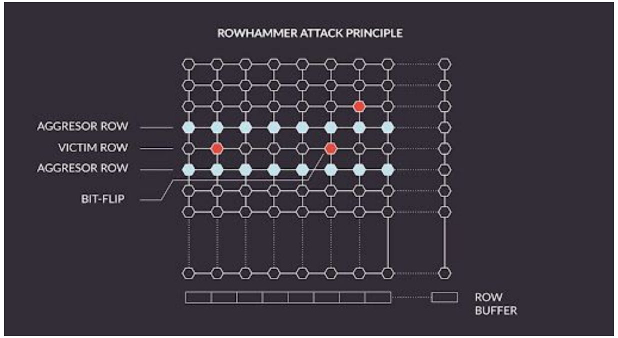


Part I: Attacking SLH-DSA with fault injections

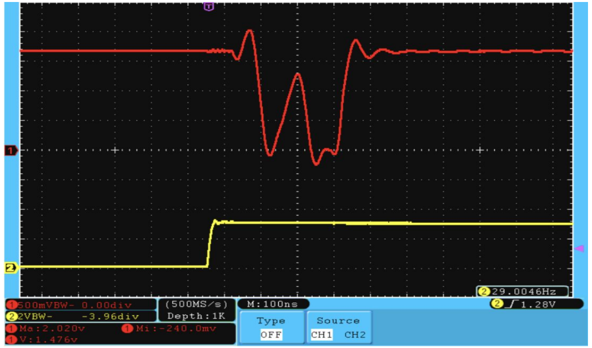
Fault injection attacks (FIA)



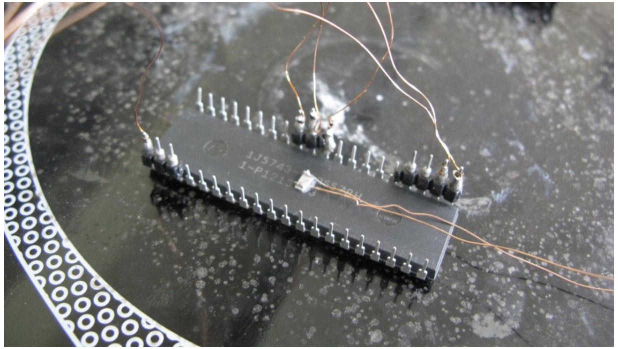
Lasers & other EM waves



Row Hammer

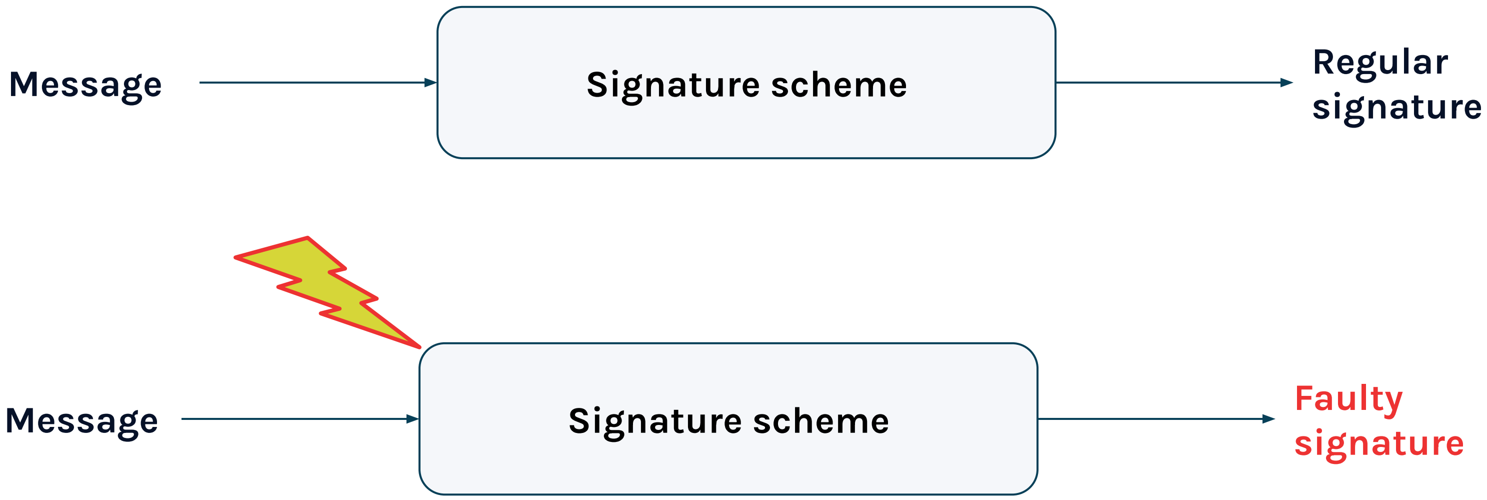


Voltage variation



Temperature variation

FIA and digital signatures



Message → Signature scheme → Regular signature

Message → Signature scheme → Faulty signature

Main idea:

1. Fault the signing procedure
2. Exploit the output (for example to recover the signing key)

The simplest hash-based signature

Main idea is to use *hash chains*

sk **pk**

$$\begin{tikzcd}
\boxed{s1} \arrow[r] & H(s1) \arrow[r] & H^2(s1) \arrow[r] & \dots \arrow[r] & H^{N-1}(s1) \arrow[r] & \boxed{H^N(s1) = p1} \\
\boxed{s2} \arrow[r] & H(s2) \arrow[r] & H^2(s2) \arrow[r] & \dots \arrow[r] & H^{N-1}(s2) \arrow[r] & \boxed{H^N(s2) = p2}
\end{tikzcd}$$

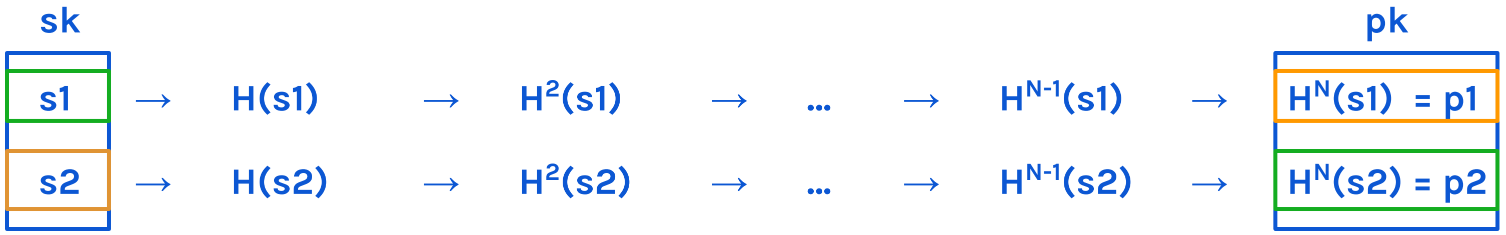
Signing key:	sk = (s1, s2) two 256-bit values
Verification key:	pk = (p1, p2)
Signature of m:	sig = (sig1, sig2) = $(H^m(s1), H^{N-m}(s2))$
Verification:	Check that $(H^{N-m}(sig1), H^m(sig2)) = (p1, p2)$

Observation 1: pk is a convoluted hash commitment of sk, sig partially opens this commitment

Observation 2: From any valid signature, we can recover the public key

Observation 3: This is a *one-time* signature (OTS). Asking two or more signatures breaks the scheme

Attacks on the simplest hash-based signature



$$\begin{tikzcd}
\text{sk} & & & & & \text{pk} \\
s1 \arrow[r] & H(s1) \arrow[r] & H^2(s1) \arrow[r] & \ldots \arrow[r] & H^{N-1}(s1) \arrow[r] & H^N(s1) = p1 \\
s2 \arrow[r] & H(s2) \arrow[r] & H^2(s2) \arrow[r] & \ldots \arrow[r] & H^{N-1}(s2) \arrow[r] & H^N(s2) = p2
\end{tikzcd}$$

Black box attack (two signatures):

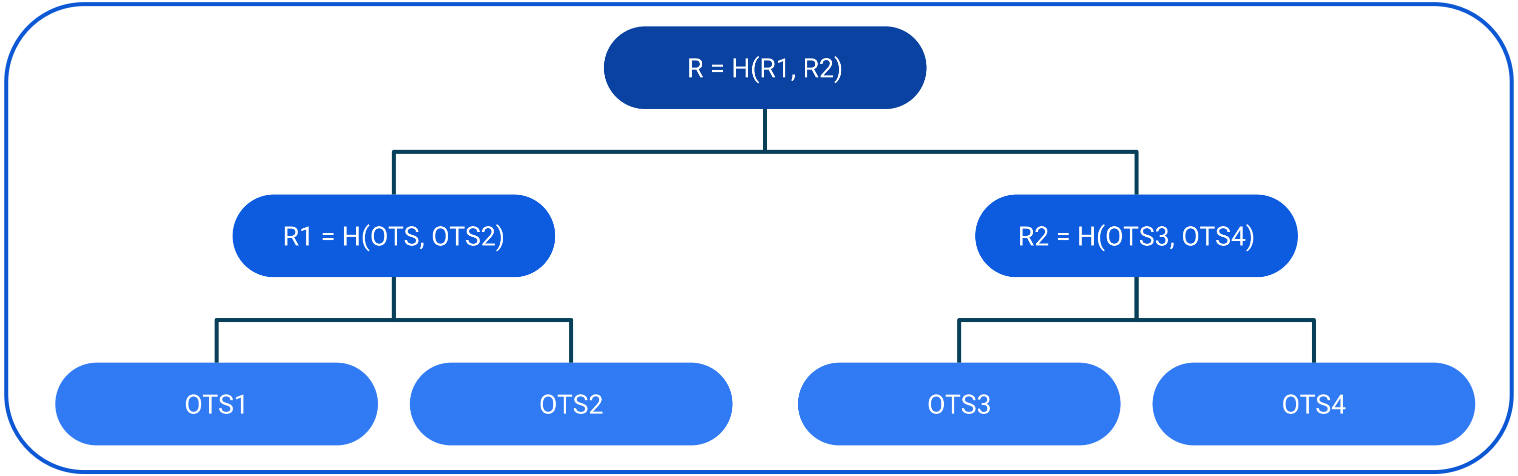
1. Ask two signatures (for **msg1** < **msg2**)
2. We can forge a signature for *any* **msg1** < **msg** < **msg2**

This is not acceptable ⇒ see next slides for a remediation

Fault injection attack (random fault):

1. Ask for a signature of **msg1 = 0** and fault the counter **msg1** (→ **msg2**) when computing $H^{msg1}(s2)$
2. We can forge a signature for any message **0 = msg1** < **msg** < **msg2**

Merkle trees: from one-time to few-time


R = H(R1, R2)

R1 = H(OTS, OTS2)

R2 = H(OTS3, OTS4)

OTS1

OTS2

OTS3

OTS4

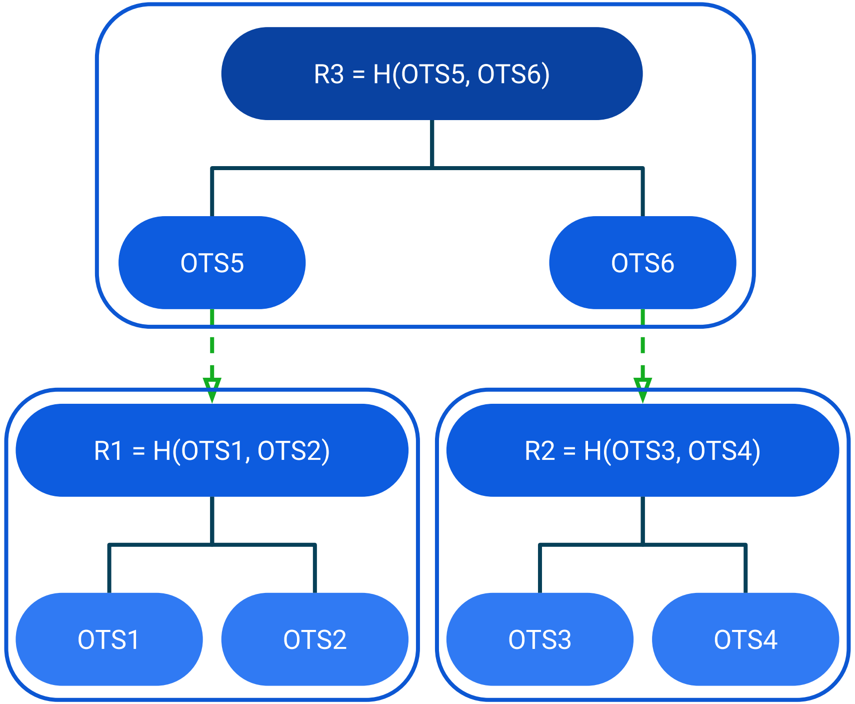
Merkle trees: allows to sign N times using N OTS

- **Signature:** 1 signature = { 1 OTS signature } + { log N hashes (= the co-path of the OTS used) }
  - We can think of a signature as a certificate chain
- **Limitation:**
  - Keygen requires to compute the entire tree ⇒ O(N) hashes
  - Requires a stateful counter → bad for deployment, bad against FIA!

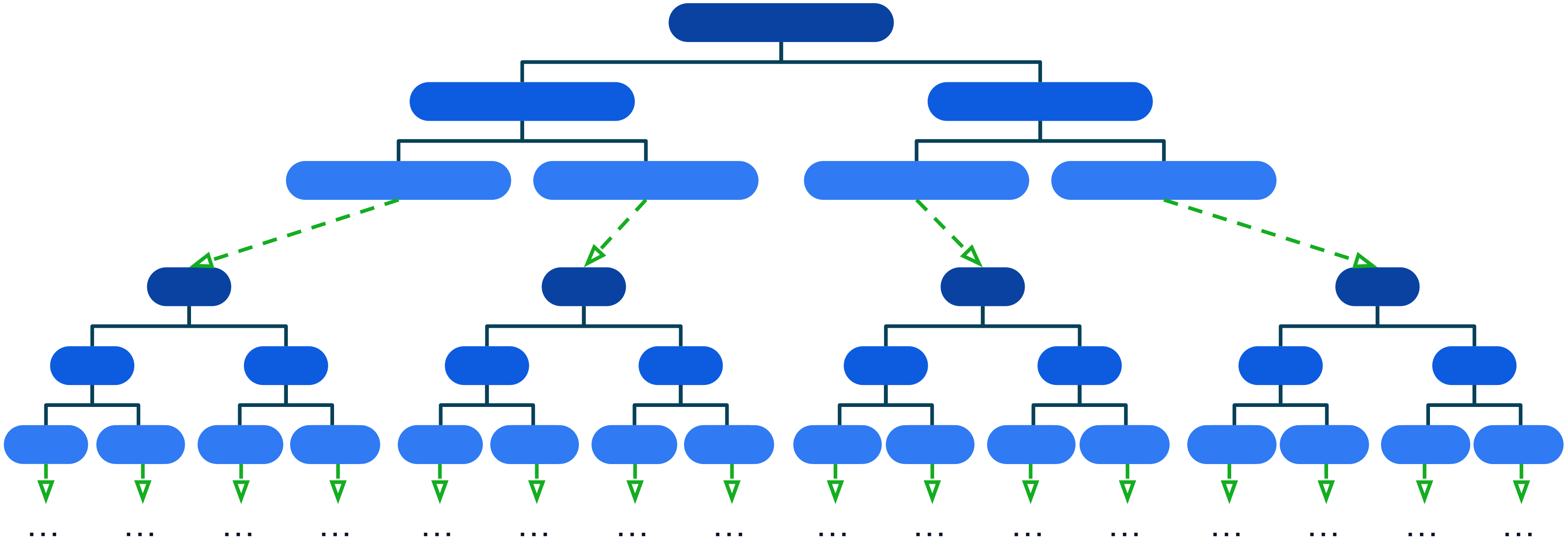
Goldreich trees: *stateless* few-time signatures

Goldreich trees:

- **Principle:**
  - N Merkle trees, each of depth 1
  - Each OTS signs the root of the Merkle tree below it
- **Signature:** 1 signature = { log N hashes } + { log N OTS signatures }
  - The "certificate chain" analogy still holds
- **Advantages:**
  - Generating pk = R2 takes time O(1), so scales for arbitrarily large N
  - Can be made *stateless* when n → ∞
- **Fault attacks?**
  - Fault the OTS
  - Fault the Merkle tree recomputation



SPHINCS+: Merkle + Goldreich + optimizations



SPHINCS+: a huge Goldreich “hyper-tree”, with each Merkle tree having many levels

1. The specific OTS used in SPHINCS+ is **WOTS+**
2. The bottom-most OTS are actually few-time signatures (specifically **FORS**)
3. 3 security levels (128/192/256), 2 variants (short/fast). *Stateless.*

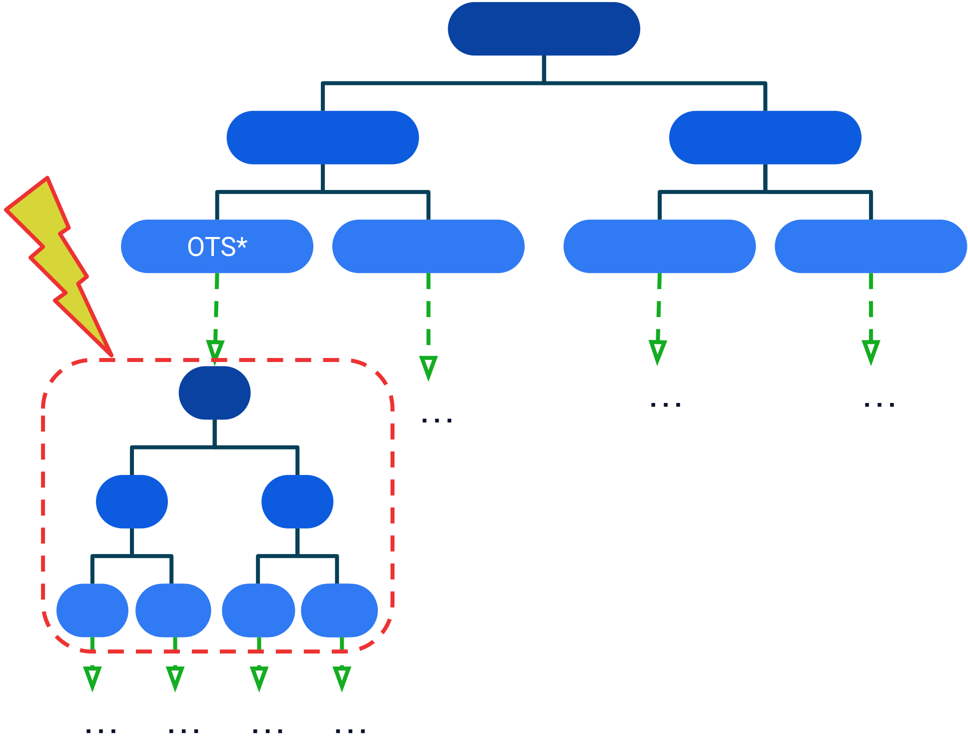
Fault injection on SPHINCS+ (Castelnovi et al, 2018)

Main idea: make a top-level OTS sign 2 ≠ values

1. Ask two signatures of msg
   - ○ SPHINCS+ is deterministic → the "signing path" is always the same
2. **First signature:** no fault
3. **Second signature:** fault the computation of the second-level Merkle tree ⚡
4. **OTS* signs two ≠ values → break the unforgeability of OTS* for a subset P of messages**

How to exploit this: Tree grafting 🌲

1. Generate a partial signature (up to the second-level Merkle tree M) for msg* until the root of M is in P
   a. Recall: a signature ≈ certificate chain
2. Sign M using the faulted OTS
3. We now have a forged signature



Fault injection on SPHINCS+ (Castelnovi et al, 2018)

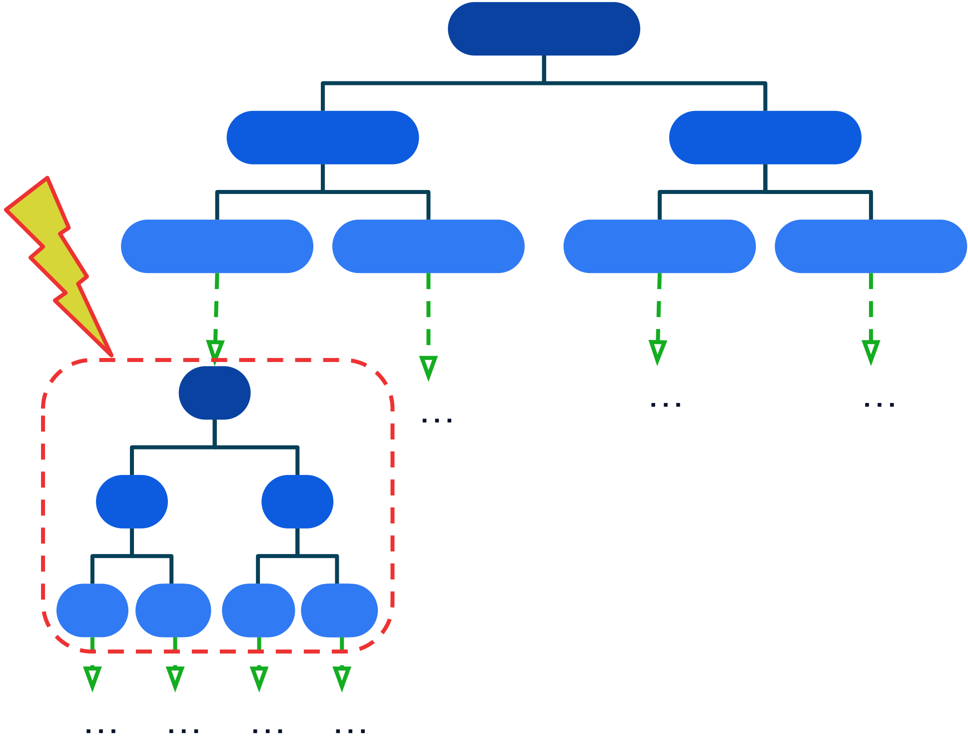
Main idea: make a top-level OTS sign 2 ≠ values

1. Ask two signatures of msg
   - SPHINCS+ is deterministic → the "signing path" is always the same
2. **First signature:** no fault
3. **Second signature:** fault the computation of the second-level Merkle tree
4. **OTS* signs two ≠ values → break the unforgeability of OTS* for a subset P of messages**

Bonus:

- One fault
- Low required precision
- Faulted signatures are valid

Extended & implemented in subsequent works



Part II: Protecting SLH-DSA against fault injections

Countermeasures

Goal: prevent triggering twice the same WOTS+ instance on different messages

Issue: SLH-DSA is *stateless*, so we need to add some shenanigans in memory to ensure that

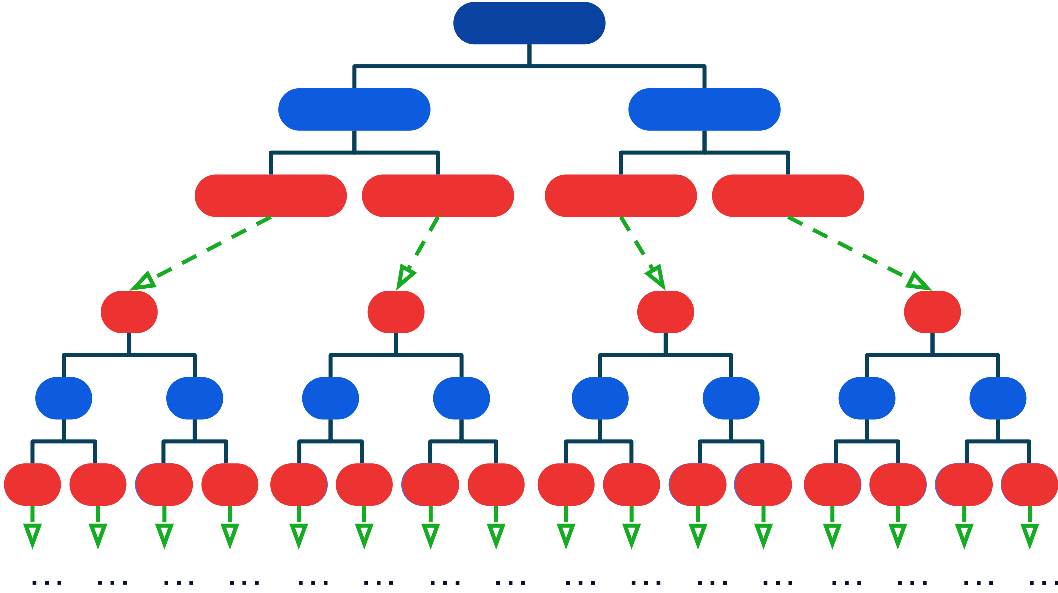
We discuss three countermeasures:

- Caching
- Redundancy
- Redundancy + dummies

Caching layers (Genêt CHES 2023)

Inspired by Gravity-SPHINCS:

- **Static:** cache all WOTS+ in the top layers
  - c = # of layers that can be cached depends on available memory
  - *Exponential* in c
- **Dynamic:** cache all WOTS+ operations occurring during previous computations



Caching layers (Genêt CHES 2023)

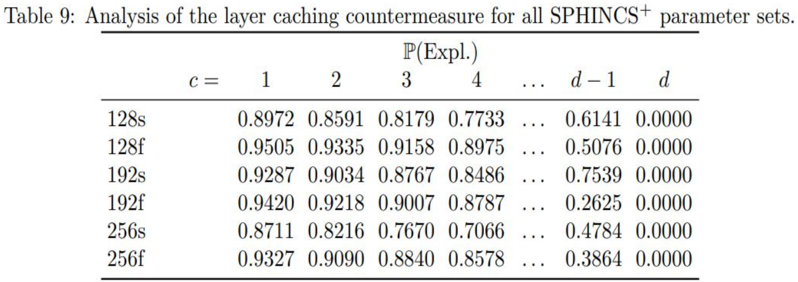


Table 9: Analysis of the layer caching countermeasure for all SPHINCS$^+$ parameter sets.

	ℙ(Expl.)						
$c =$	1	2	3	4	…	$d-1$	d
128s	0.8972	0.8591	0.8179	0.7733	…	0.6141	0.0000
128f	0.9505	0.9335	0.9158	0.8975	…	0.5076	0.0000
192s	0.9287	0.9034	0.8767	0.8486	…	0.7539	0.0000
192f	0.9420	0.9218	0.9007	0.8787	…	0.2625	0.0000
256s	0.8711	0.8216	0.7670	0.7066	…	0.4784	0.0000
256f	0.9327	0.9090	0.8840	0.8578	…	0.3864	0.0000

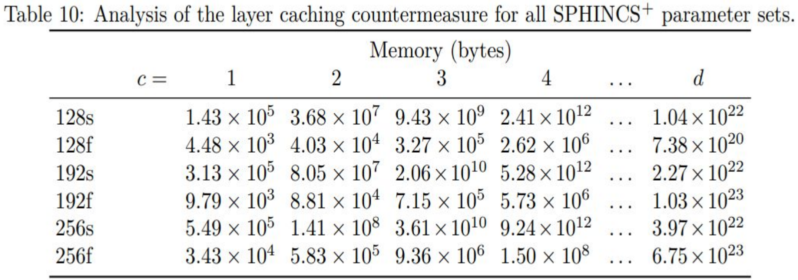
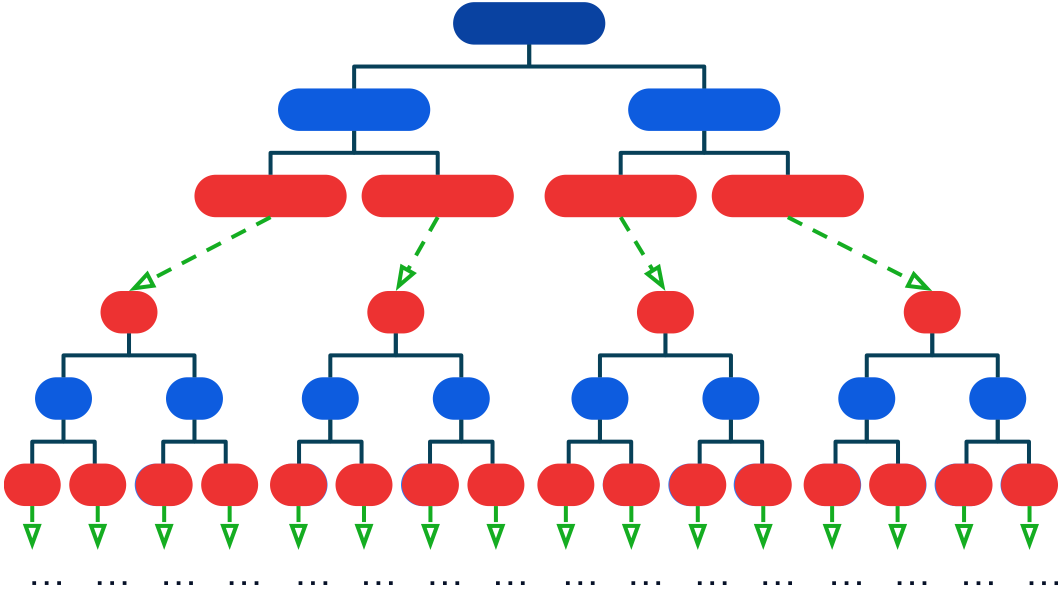


Table 10: Analysis of the layer caching countermeasure for all SPHINCS$^+$ parameter sets.

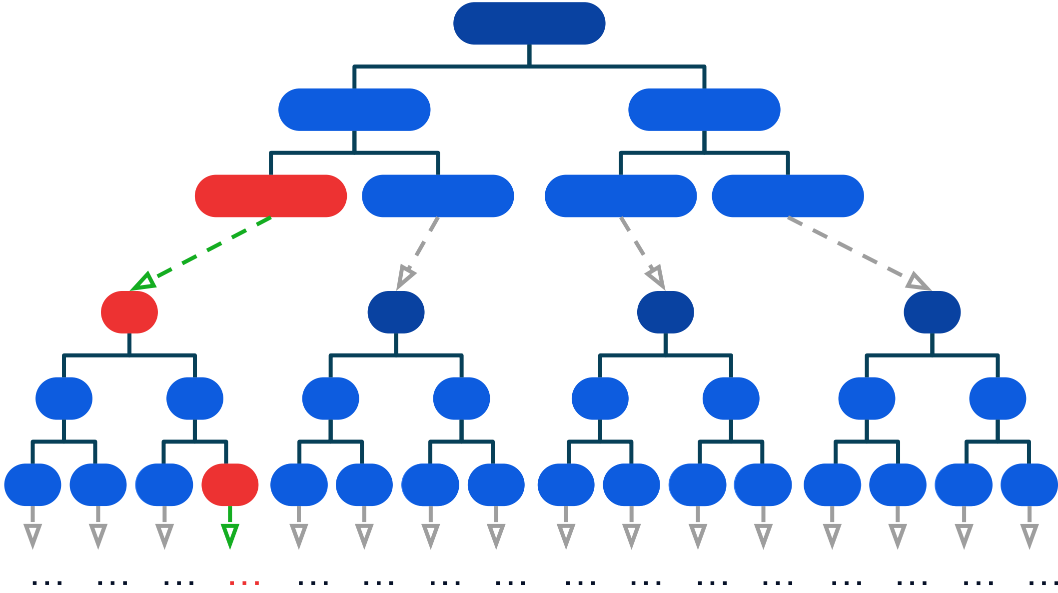
	Memory (bytes)					
$c =$	1	2	3	4	…	d
128s	1.43×10^5	3.68×10^7	9.43×10^9	2.41×10^{12}	…	1.04×10^{22}
128f	4.48×10^3	4.03×10^4	3.27×10^5	2.62×10^6	…	7.38×10^{20}
192s	3.13×10^5	8.05×10^7	2.06×10^{10}	5.28×10^{12}	…	2.27×10^{22}
192f	9.79×10^3	8.81×10^4	7.15×10^5	5.73×10^6	…	1.03×10^{23}
256s	5.49×10^5	1.41×10^8	3.61×10^{10}	9.24×10^{12}	…	3.97×10^{22}
256f	3.43×10^4	5.83×10^5	9.36×10^6	1.50×10^8	…	6.75×10^{23}



Caching layers (Genêt CHES 2023)

Inspired by Gravity-SPHINCS:

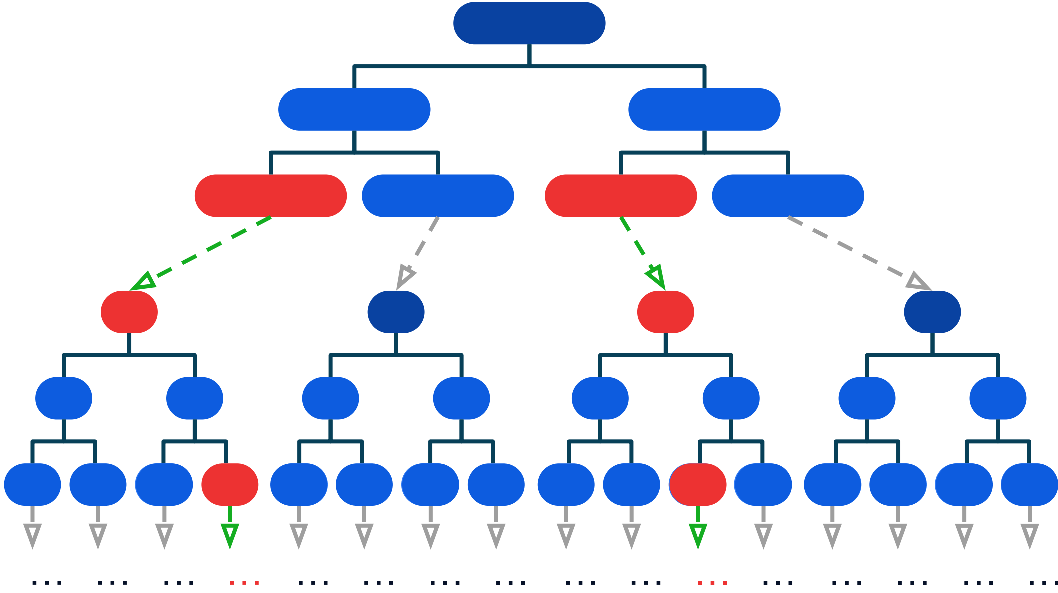
- **Static:** cache all WOTS+ in the top layers
    - c = # of layers that can be cached depends on available memory
    - *Exponential* in c
- **Dynamic:** cache all WOTS+ operations occurring during previous computations



Caching layers (Genêt CHES 2023)

Inspired by Gravity-SPHINCS:

- **Static:** cache all WOTS+ in the top layers
  - c = # of layers that can be cached depends on available memory
  - *Exponential* in c
- **Dynamic:** cache all WOTS+ operations occurring during previous computations



Caching layers (Genêt CHES 2023)

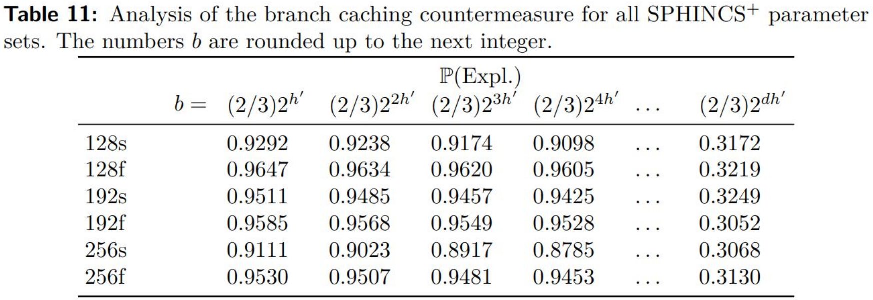


Table 11: Analysis of the branch caching countermeasure for all SPHINCS$^+$ parameter sets. The numbers b are rounded up to the next integer.

	$\mathbb{P}$(Expl.)					
$b =$	$(2/3)2^{h'}$	$(2/3)2^{2h'}$	$(2/3)2^{3h'}$	$(2/3)2^{4h'}$	...	$(2/3)2^{dh'}$
128s	0.9292	0.9238	0.9174	0.9098	...	0.3172
128f	0.9647	0.9634	0.9620	0.9605	...	0.3219
192s	0.9511	0.9485	0.9457	0.9425	...	0.3249
192f	0.9585	0.9568	0.9549	0.9528	...	0.3052
256s	0.9111	0.9023	0.8917	0.8785	...	0.3068
256f	0.9530	0.9507	0.9481	0.9453	...	0.3130

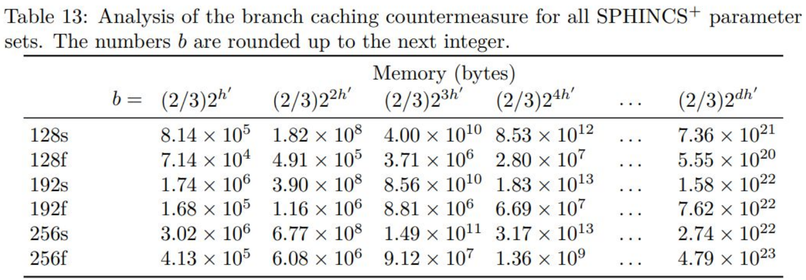
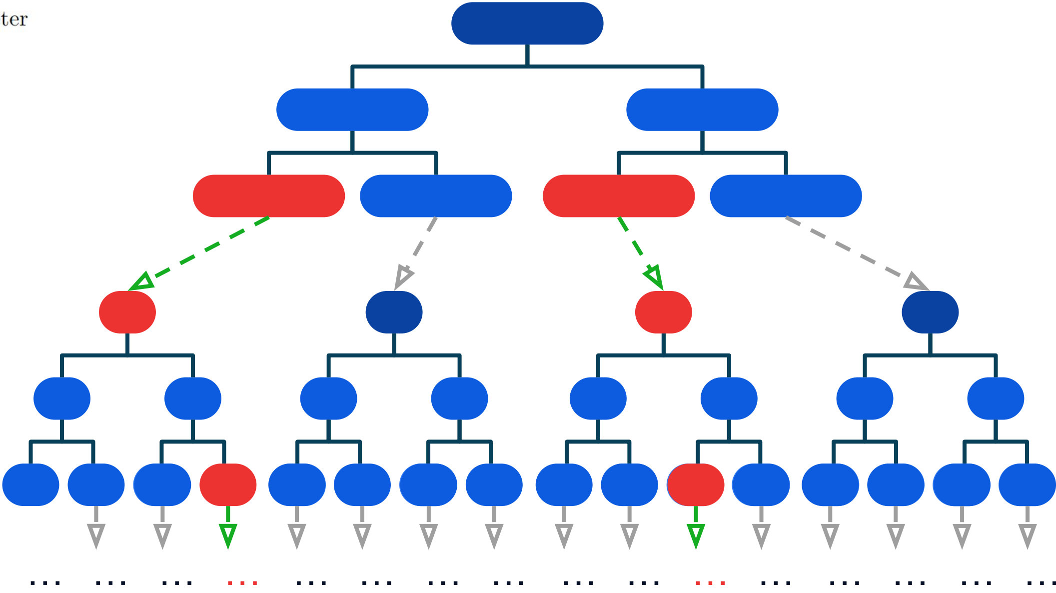


Table 13: Analysis of the branch caching countermeasure for all SPHINCS$^+$ parameter sets. The numbers b are rounded up to the next integer.

	Memory (bytes)					
$b =$	$(2/3)2^{h'}$	$(2/3)2^{2h'}$	$(2/3)2^{3h'}$	$(2/3)2^{4h'}$	...	$(2/3)2^{dh'}$
128s	8.14×10^5	1.82×10^8	4.00×10^{10}	8.53×10^{12}	...	7.36×10^{21}
128f	7.14×10^4	4.91×10^5	3.71×10^6	2.80×10^7	...	5.55×10^{20}
192s	1.74×10^6	3.90×10^8	8.56×10^{10}	1.83×10^{13}	...	1.58×10^{22}
192f	1.68×10^5	1.16×10^6	8.81×10^6	6.69×10^7	...	7.62×10^{22}
256s	3.02×10^6	6.77×10^8	1.49×10^{11}	3.17×10^{13}	...	2.74×10^{22}
256f	4.13×10^5	6.08×10^6	9.12×10^7	1.36×10^9	...	4.79×10^{23}

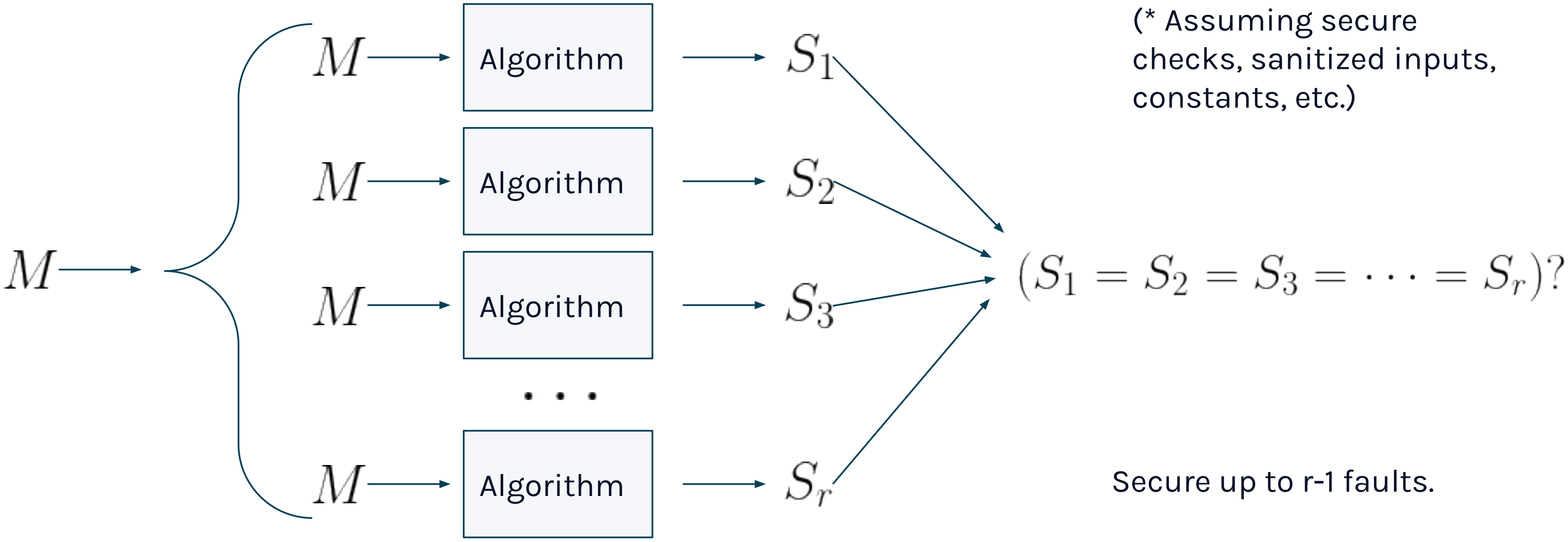


Caching strategies are too costly

“Since the threat of a fault can never be completely eliminated, the current best solution to protect the signature scheme against accidental and intentional faults is through redundancy; an observation that is shared by others”

“In conclusion, the results of this paper urge all real-world deployments of SPHINCS+ to come with redundancy checks, even if the use case is not prone to faults”

Best countermeasure yet: redundancy



$$(S_1 = S_2 = S_3 = \cdots = S_r)?$$

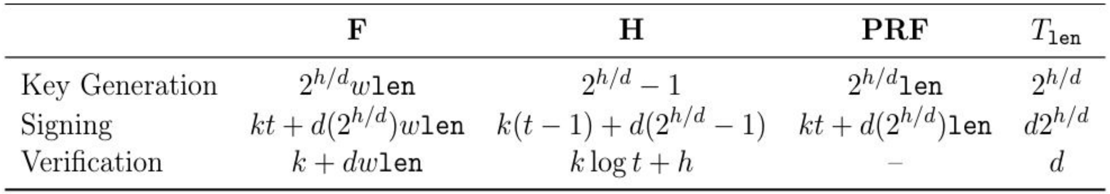
(* Assuming secure checks, sanitized inputs, constants, etc.)

Secure up to r-1 faults.

Attacker model

Attacker has a scope: they can recognize patterns on operations, but not their operands

=> can distinguish the operations based on the nb of input words



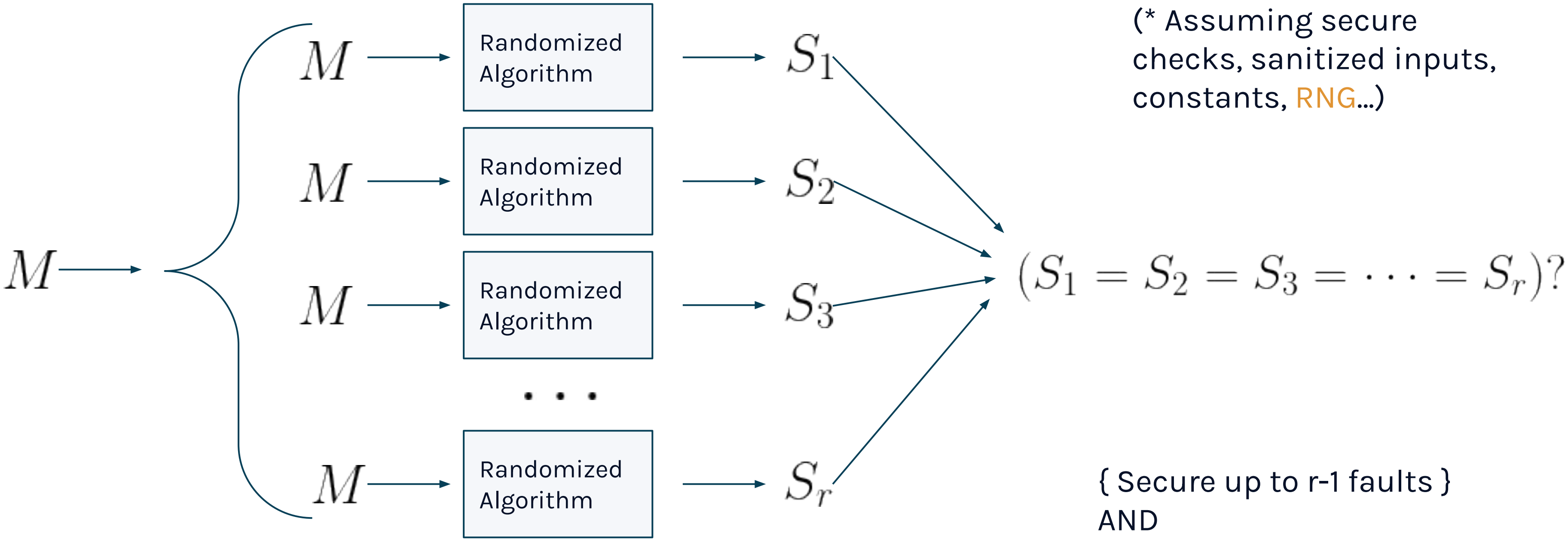
	F	**H**	**PRF**	$T_{\texttt{len}}$
Key Generation	$2^{h/d}w\texttt{len}$	$2^{h/d}-1$	$2^{h/d}\texttt{len}$	$2^{h/d}$
Signing	$kt+d(2^{h/d})w\texttt{len}$	$k(t-1)+d(2^{h/d}-1)$	$kt+d(2^{h/d})\texttt{len}$	$d2^{h/d}$
Verification	$k+dw\texttt{len}$	$k\log t+h$	–	d

Attacker model

Attacker has a scope: they can recognize patterns on operations, but not their operands
=> can distinguish the operations based on the nb of input words

Comparisons are protected: the attacker needs to perturbate the SLH-DSA execution
=> must inject twice the same fault (consider no collision)

Redundancy + randomization



$$(S_1 = S_2 = S_3 = \cdots = S_r)?$$

(* Assuming secure checks, sanitized inputs, constants, RNG…)

{ Secure up to r-1 faults }
AND
{ Probably secure to > r-1 faults }

Randomization

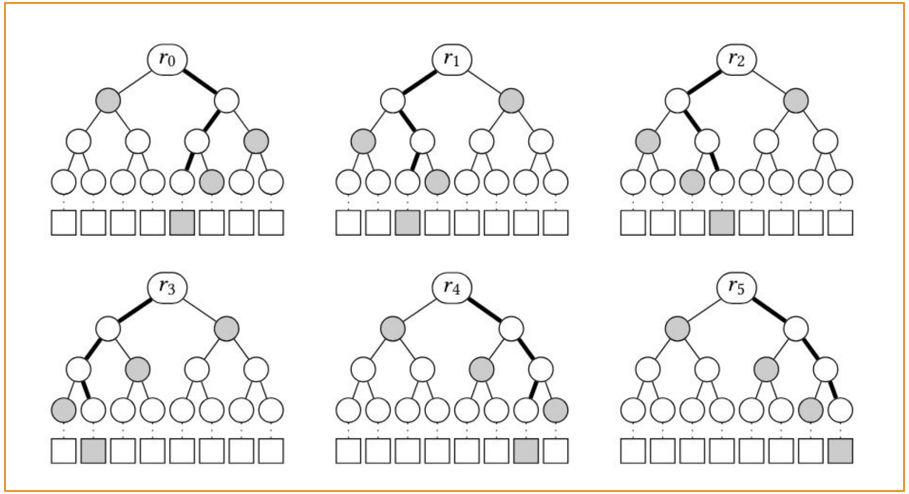
Execute operations in a random order

- For example: 16 S-boxes in AES ⇒ **16!** possible orders

In SLH-DSA, many operations can be performed in parallel:

- at every level of the FORS (leaves)
- at every level of the hypertree
- at every step of a WOTS chain
- (optimizations possible)

For example, bottom layer of FORS ⇒ **(12*2^14)!** possible orders



Randomization

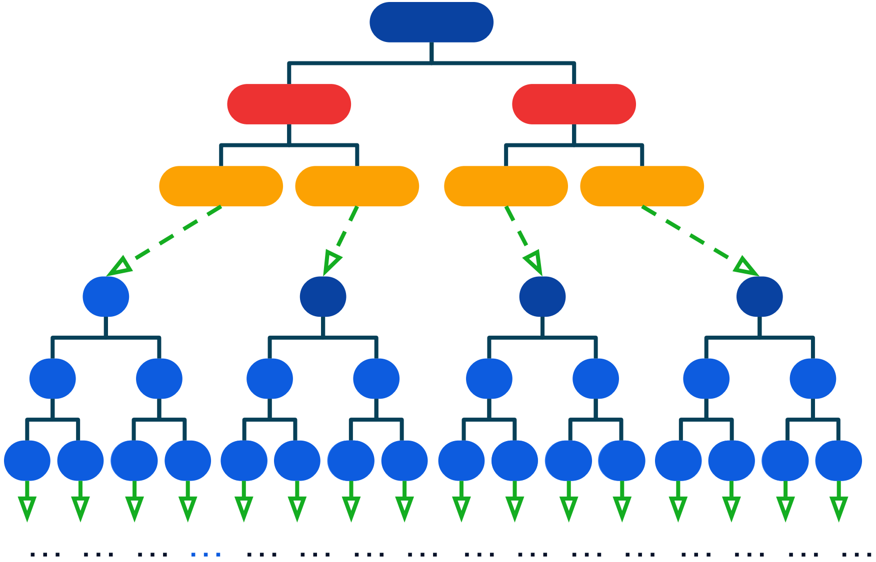
Execute operations in a random order

- For example: 16 S-boxes in AES ⇒ **16!** possible orders

In SLH-DSA, many operations can be performed in parallel:

- at every level of the FORS (leaves)
- at every level of the hypertree
- at every step of a WOTS chain
- (optimizations possible)

For example: bottom layer of FORS ⇒ **(12*2^14)!** possible orders



Randomization

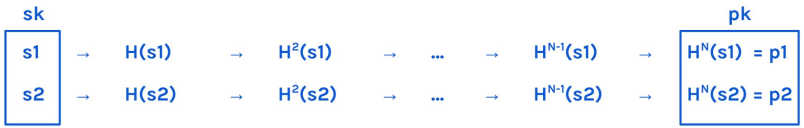
Execute operations in a random order

- For example: 16 S-boxes in AES ⇒ **16!** possible orders

In SLH-DSA, many operations can be performed in parallel:

- at every level of the FORS (leaves)
- at every level of the hypertree
- at every step of a WOTS chain
- (optimizations possible)

For example, bottom layer of FORS ⇒ **(12*2^14)!** possible orders



$$\begin{tikzcd}
\text{sk} & & & & & & \text{pk} \\
s1 \arrow[r] & H(s1) \arrow[r] & H^2(s1) \arrow[r] & \dots \arrow[r] & H^{N-1}(s1) \arrow[r] & H^N(s1) = p1 \\
s2 \arrow[r] & H(s2) \arrow[r] & H^2(s2) \arrow[r] & \dots \arrow[r] & H^{N-1}(s2) \arrow[r] & H^N(s2) = p2
\end{tikzcd}$$

Randomization

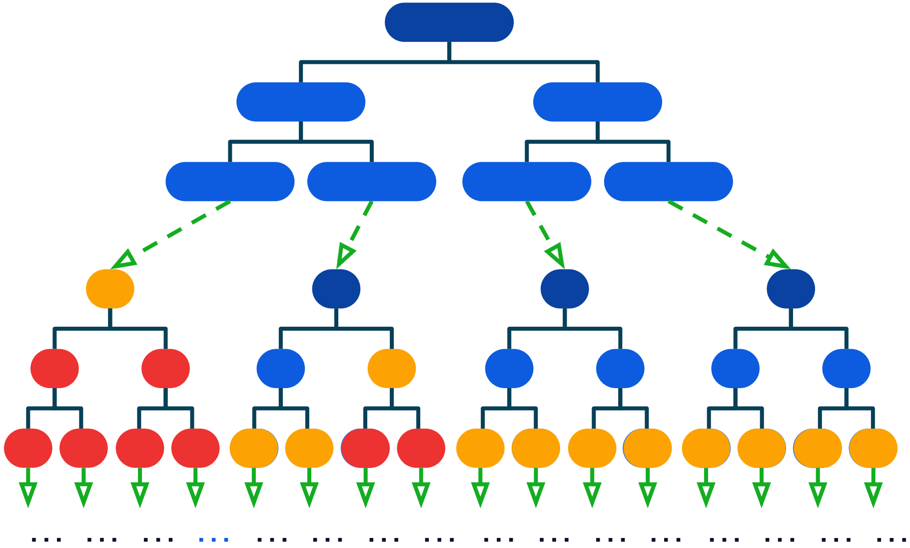
Execute operations in a random order

- For example: 16 S-boxes in AES ⇒ **16!** possible orders

In SLH-DSA, many operations can be performed in parallel:

- at every level of the FORS (leaves)
- at every level of the hypertree
- at every step of a WOTS chain
- (optimizations possible)

For example, bottom layer of FORS ⇒ **(12*2^14)!** possible orders

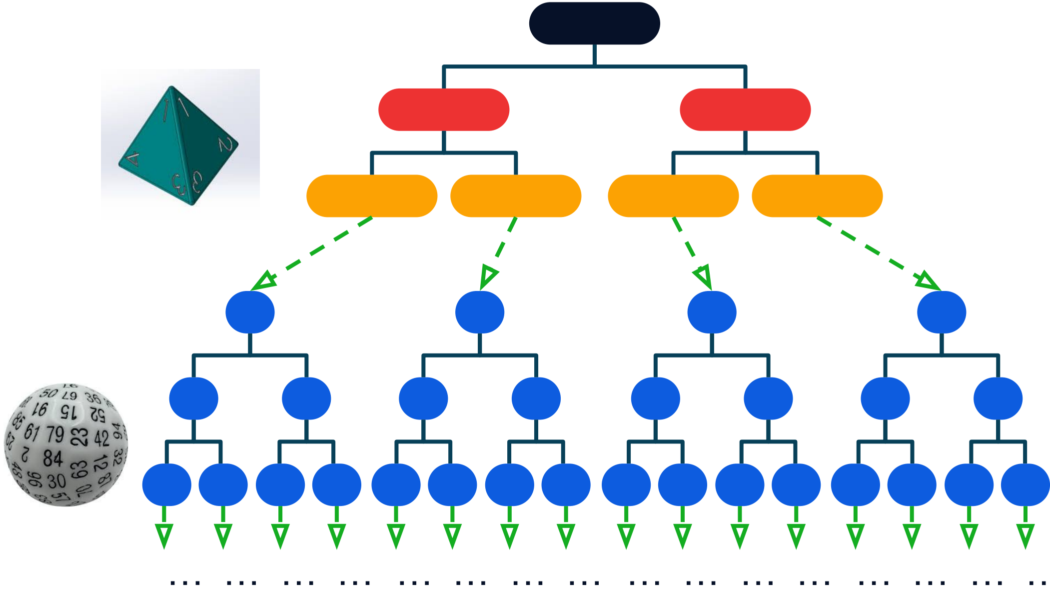


Decaying entropy

Climbing in each subtree lowers the number of possible orders, up to the root, where no randomness can occur.

Depending on the constraints:

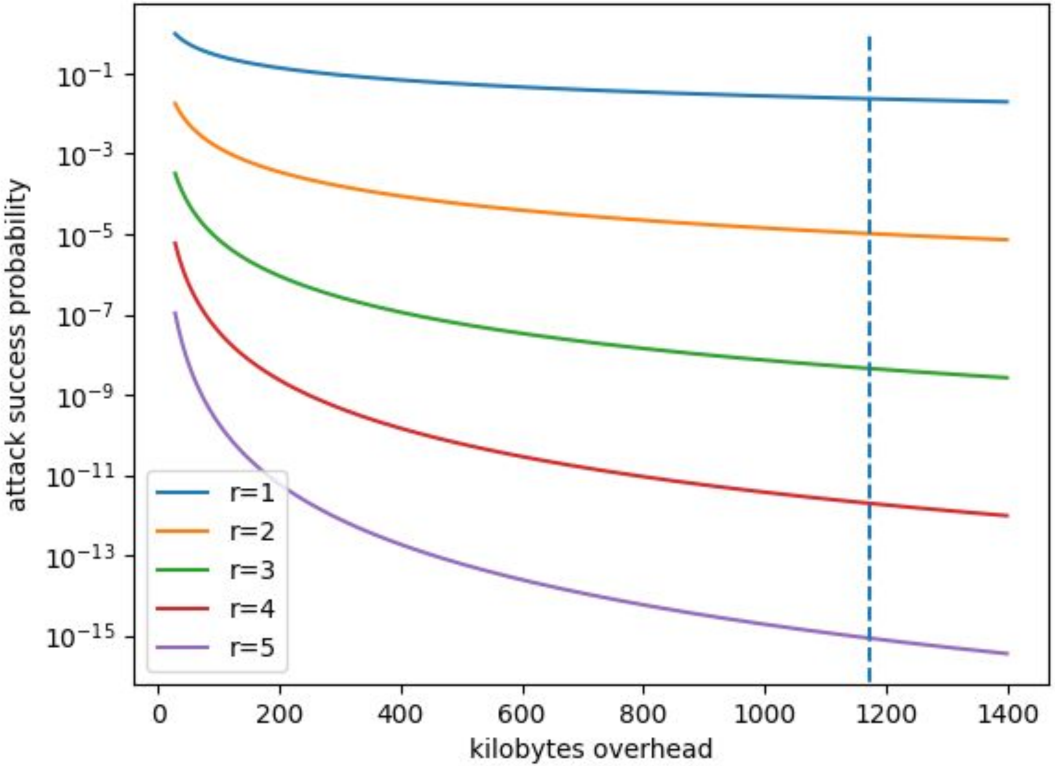
- **Add dummy operations**
  ⇒ artificially raise entropy and decreases success probability
- **Locally duplicate the operation**
  ⇒ perfect security but need to be carefully made (eg duplicate inputs)



Attack success probability (no dummies)

128s	r=1	r=2	r=3	r=4	r=5
PRF	1.00e+00	5.47e-06	2.99e-11	1.64e-16	8.96e-22
F-FORS	1.00e+00	1.74e-05	3.04e-10	5.30e-15	9.25e-20
F-i	1.00e+00	7.97e-06	6.36e-11	5.07e-16	4.04e-21
Tlen	8.57e-01	2.39e-04	6.67e-08	1.86e-11	5.19e-15
H0	9.52e-01	4.54e-02	2.16e-03	1.03e-04	4.90e-06
Hmax	1.00e+00	6.98e-05	4.87e-09	3.39e-13	2.37e-17

Asymptotic security (dummies on most sensitive pool)



Quick PoC

Ran simulations on open source “sloth” implementation by Markku (https://github.com/slh-dsa/sloth), slightly modified to get:

- { Compiled in -O0 } & { r executions and final comparisons }
- { Compiled in -O0 } & { r executions and final comparisons w/ randomization of F leaves }

Implementation allows for easy and immediate randomization of 14*12 operations (modifying a bit more would allow for much better, but time constraints…)

gdb scripting to stuck at 0 the same register at the exact same time:

- Redundancy ⇒ 100% success rate
- Redundancy + randomization:
  - r = 2 ⇒ 55 successes on 10k (p=0.0055, expected 0.0059)
  - r = 3 ⇒ 2 successes on 200k (p=0.00001, expected 0.0000354)

Conclusion

Fault injection attacks

- SLH-DSA is particularly vulnerable to fault injection attacks
    - Easy to mount
    - Easy to exploit
    - Not detectable by default

Countermeasures

- **Caching** ⇒ seems too expensive
- **Pure redundancy** ⇒ works but expensive
- **Redundancy + dummies + shuffling** ⇒ tolerates faults **beyond the redundancy threshold**

Questions?