Mask Compression: High-Order Masking on Memory-Constrained Devices

Markku-Juhani O. Saarinen [1]   Mélissa Rossi [2]

[1] PQShield, UK and Tampere University, Finland   [2]ANSSI, France

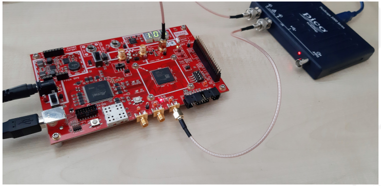
16 August 2023
Selected Areas in Cryptography (SAC) 2023
Fredericton, New Brunswick, Canada

Outline of the Talk

1. **Intro: Side-Channel Attacks & Masking Countermeasures**
2. The Basic “Trick” of Mask Compression
3. In Practice: Order-31 Masked Signatures on FPGA

Side-Channel Attacks

- **Side-Channel Attacks (SCA)** use external measurements such as latency (TA), power consumption (SPA/DPA), or electromagnetic emissions ([S/D]EMA) to extract secrets.
- *SCA resistance is important for PC, IoT, and mobile device "platform security" (secure boot, firmware updates, attestation), authentication tokens, smart cards, HSMs / secure elements..*
- Common compliance & market requirement for hardware (Common Criteria / AVA_VAN, FIPS 140-3 / ISO 17825).



- **Post-Quantum Cryptography (PQC)** implementations – e.g. lattice-based schemes **Dilithium** and **Kyber** inherit all of the security and compliance requirements of Elliptic Curve or RSA based solutions in applications.

Masking: Non-Invasive SCA Security

- **Masking:** Secret data $[\![\mathbf{s}]\!]$ is processed in d randomized shares $\mathbf{s}_i$.

$$\begin{aligned} \text{Boolean Masking:} \quad & [\![\mathbf{s}]\!] = \mathbf{s}_1 \oplus \mathbf{s}_2 \oplus \cdots \oplus \mathbf{s}_d \\ \text{Arithmetic Masking:} \quad & [\![\mathbf{s}]\!] = \mathbf{s}_1 + \mathbf{s}_2 + \cdots + \mathbf{s}_d \pmod{q}. \end{aligned}$$

- Individually each share $\mathbf{s}_i$ is uniformly random, as is any combination if $d - 1$ shares.

- A bit like d-of-d secret sharing: Even full knowledge of $d - 1$ shares $\sum_{i=1}^{d-1} \mathbf{s}_i$ reveals nothing about $[\![\mathbf{s}]\!] = \sum_{i=1}^{d} \mathbf{s}_i$. You need all d shares. We call $d - 1 = t$ the *masking order*.

- If you only have partial or "noisy" measurements (traces), it has been shown that the number of such observations required to learn $[\![\mathbf{s}]\!]$ grows *exponentially* with d.

  (Chari et al. 1999 – a lot of subsequent theoretical and experimental work supports this.)

Masking Gadgets, Models, and Proofs

Computation on masked shares must be arranged so that intermediate variables have no statistical correlation with the actual secret variables. They need to appear random too.

→ **Gadgets:** Common approach is to first design a set of "gadgets" for simple operations (logical AND, selection, bit shift, etc.) and compose larger algorithms from them.

→ **Refreshing:** Masking security generally requires that a particular secret sharing of variable $[\![\mathbf{s}]\!]$ can only be used once; after that, it needs to be *refreshed* (re-randomized).

→ **Proofs:** The proofs can be made in several models; the Ishai-Sahai-Wagner (ISW) *t*-probing security requires that any *t* internal intermediate values don't reveal secrets.

The noisy leakage model is an alternative; links have been proven between *t*-probing security, noisy leakage model, and information-theoretic attack complexity bounds.

Computing on Shares: Linear, Quadratic, Quasilinear

- Linear operations only need **linear** $O(d)$ effort to mask:
  - *Addition / subtraction / XOR of masked variables* ($[\![\mathbf{s}]\!] + [\![\mathbf{r}]\!]$).
  - *Multiplication (or Boolean AND, OR) with a scalar constant or a public variable* ($\mathbf{c} \cdot [\![\mathbf{s}]\!]$).
  - *Share-independent linear operations such as NTT (Number Theoretic Transform.)*

- Non-linear operations generally require **quadratic** $O(d^2)$ effort:
  - *Multiplication (Boolean AND, OR) between secret variables* ($[\![\mathbf{s}]\!] \cdot [\![\mathbf{r}]\!]$).
  - *Conversions between Arithmetic and Boolean masking representations (A2B and B2A).*
  - *Symmetric cryptography like AES or SHA3. Especially these may benefit from Threshold Implementation (TI) technique, requiring additional share(s) but less randomness.*

- But some non-linear operations can be done with **quasilinear** $O(d \log d)$ effort:

  Practical quasilinear techniques are known only for a limited number of computational tasks.

Masking in Lattice Crypto: Long Secret Vectors

Example. Most MLWE-based algorithms are built on arithmetic in rings $\mathbb{Z}_q[X]/(X^n + 1)$:

- → Kyber: $q = 3329$, $n = 256$ with $k \in \{2, 3, 4\}$ rings (different security levels) in secret $\hat{\mathbf{s}}$.
- → Dilithium: $q = 8380417$, $n = 256$ with $(k + \ell) \in \{8, 11, 15\}$ rings in secret $(\mathbf{s}_1, \mathbf{s}_2)$.
- → Raccoon: $q = 549824583172097$, $n = 512$ with $\ell \in \{4, 5, 7\}$ rings in secret $[\![\mathbf{s}]\!]$.

(Note that there are other sensitive variables that also require masking, this is just an example.)

Each ring requires at least $n \cdot \lceil \log_2 q \rceil$ bits. At PQC Category 5, a single share of Kyber's $\hat{\mathbf{s}}$ requires 12288 bits, Dilithium's $(\mathbf{s}_1, \mathbf{s}_2)$ is 88320 bits, and Raccoon's $[\![\mathbf{s}]\!]$ is 175616 bits.

Multiply this with d, the number of shares: First-order masking requires twice the amount of storage, and a potential order-9 ($d = 10$) implementation would require 10× bits, etc.

However, masked implementations access the secret shares independently of each other; much of secret key computation is performed serially, first to share 1, then to share 2, etc.

Outline of the Talk

1. Intro: Side-Channel Attacks & Masking Countermeasures
2. **The Basic “Trick” of Mask Compression**
3. In Practice: Order-31 Masked Signatures on FPGA

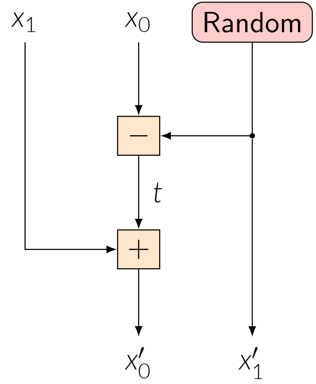
Consider the Basic Two-Share NI Refresh Gadget

1. **Input:** (x_0, x_1), two full shares of $[\![x]\!]$.
2. Pick a new, uniform random x'_1.
3. Intermediate variable $t = x_0 - x'_1$.
4. Compute output $x'_0 = t + x_1$.
5. **Output:** Refreshed shares (x'_0, x'_1).

Correctness: We maintain correct masking $[\![x]\!] = x_0 + x_1 = x'_0 + x'_1$.

First Order: Each input, output, and intermediate variable x_0, x_1, x'_0, x'_1, t is statistically uncorrelated with secret $[\![x]\!]$.

Have: (x_0, x_1)



Refreshed: (x'_0, x'_1)

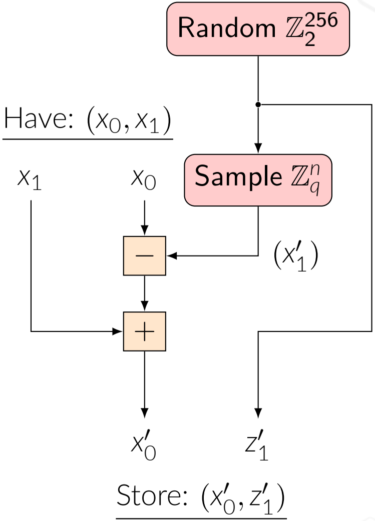
Basic (first-order NI) MaskCompress Trick

1. **Input:** (x_0, x_1), two full shares of $[\![x]\!]$.
2. Generate a new short seed $z'_1 \leftarrow \$$.
3. Expand seed to share $x'_1 = \mathsf{Sample}_G(z'_1)$.
4. Subtract new random: $t = x_0 - x'_1$.
5. Update stored share: $x'_0 = t + x_1$.
6. **Output:** Compressed pair (x'_0, z'_1).

Correctness: The compressed representation (x'_0, z'_1) satisfies $x'_0 + \mathsf{Sample}_G(z'_1) = x_0 + x_1$.

First Order: Uniform, uncorrelated variables.

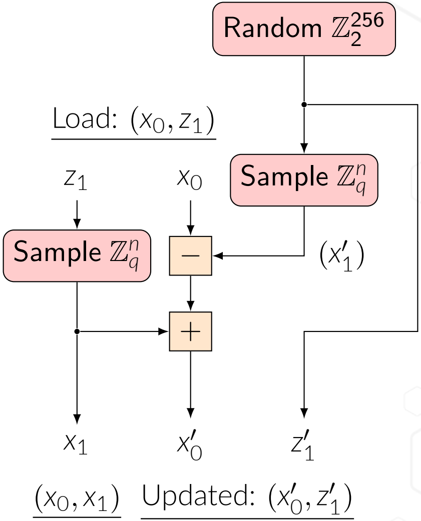
Almost Half Size: The second full share x'_1 can be discarded. Its 256-bit seed z'_1 suffices.



Basic (first-order, NI) LoadShare

1. **Input:** Compressed masking (x_0, z_1): One full share x_0 and one seed z_1.
2. Generate a new short seed $z_1' \leftarrow \$$.
3. Expand it to share $x_1' = \textsf{Sample}_G(z_1')$.
4. Subtract new random: $t = x_0 - x_1'$.
5. Expand stored seed $x_1 = \textsf{Sample}_G(z_1)$.
6. Update stored share: $x_0' = t + x_1$.
7. **Output:** Expanded shares $[\![x]\!] = (x_0, x_1)$, refreshed "compressed" pair (x_0', z_1').

First Order: Each input, output, and intermediate variable is statistically uniform and uncorrelated with secret $x = x_0 + x_1$.



Properties

- These basic **MaskCompress**() and **LoadShare**() gadgets generalize to $d = t + 1$ shares (Alg. 1 and 2 in the paper.) Complexity is linear: d shares can be accessed in $O(d)$ time.

- Secure memory shrinks from $d \cdot |G|$ to $|G| + (d - 1)\lambda$ bits, where $|G|$ is the share size, (e.g., 5888 bits for the Dilithium ring) and λ is a security parameter (e.g., 256 bits.)

- These gadgets are shown to be t-Non-Interfering (NI), which also indicates t-probing security in the Ishai-Sahai-Wagner (ISW) model.

- Strong Non-Interference (SNI) is a property that allows gadgets to be combined more freely with other gadgets while maintaining security ("composability".) We also present SNI gadgets **SNIMaskCompress**() and **SNILoadShare**() (Algorithms 4 and 5). Unfortunately, these require quadratic $O(d^2)$ time to access d shares.

How to use it? “Share access API.”

- Implement a “share access gadget library” with an API for compressing / uncompressing shares. Modify masked implementation to access shares with this API.

- Hardware implementation of Kyber, Dilithium (and Raccoon) needs a fast, deterministic seed expander like $\mathsf{Sample}_G(\texttt{seed})$ anyway – to expand the public “lattice” matrix $\mathbf{A}$.

- You don’t need to compress everything. Often mask compression makes sense only for longest sensitive variables, and only outside computationally intensive loops.

- **It’s really generic:** We have described arithmetic masking, but one can equally well use Boolean masking. Especially code-based and multivariate cryptography has large algebraic objects (vectors, matrices over finite fields) where the technique is applicable.

Outline of the Talk

1. Intro: Side-Channel Attacks & Masking Countermeasures

2. The Basic “Trick” of Mask Compression

3. **In Practice: Order-31 Masked Signatures on FPGA**

Masked Raccoon: Side-Channel Secure Signatures

- **Raccoon** is a lattice-based signature scheme on the NIST PQC "On-Ramp."

  (The paper discusses a little bit earlier Raccoon version from IEEE SP 2023.)

- Similar to Dilithium but designed for efficient (quasilinear-time) high-order masking. Parameters up to $d = 32$.

- Raccoon has SNI proofs, but for the demonstration, we just use the NI gadgets to achieve overall quasilinear speed (we know the composition..)

You may ask: Why is Dilithium Hard To Mask?

- **Dilithium** requires a masked SHAKE; mixes bit manipulations with (mod q) arithmetic, requiring A2B and B2A; has masked comparisons / rejection sampler.

  (For these non-linear operations only quadratic O(d^2) gadgets are known.)

- **Raccoon** avoids quadratic operations.

  The cost of additional shares is nearly constant. *(Cycles/share even decreases initially due to a small constant overhead.)*

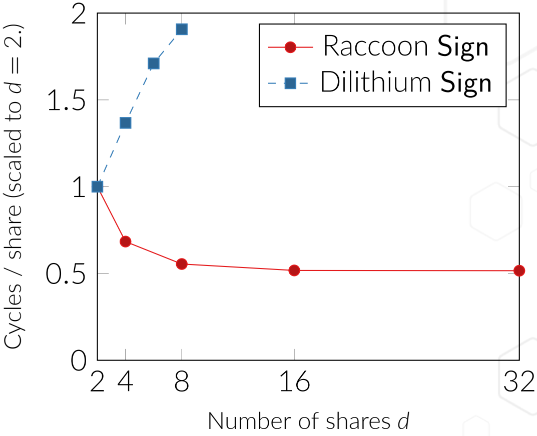


Figure 1: Cost of masking: Signing cycle count divided by d, normalized to a common start at 1 for $d = 2$. Dilithium data from [24, Table 3].

Raccoon Masking Proof: Composition of Gadgets

- Cryptanalytic sensitivity analysis: Which variables need to be protected?
- Raccoon signature and key generation functions are composed of **Masking Gadgets** that are individually t-non-intefering ($t -$ NI) or t-strong non-interfering ($t -$ SNI).
- The scheme is designed to be "masking friendly," so the proofs are quite standard.

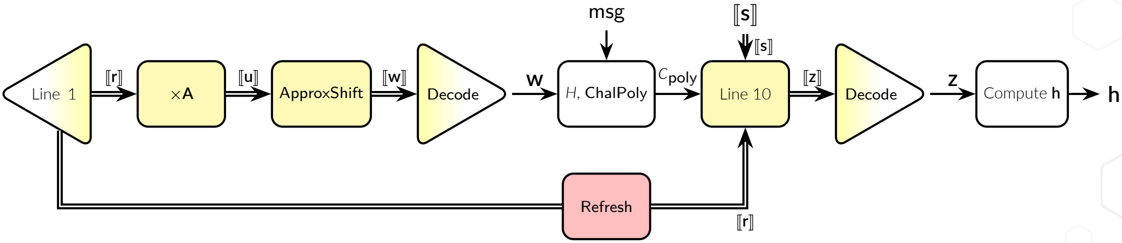


Figure 2: Raccoon signature function. Colors: gadget proven $t -$ NI, gadget proven $t -$ SNI, gadget unmasked. Single arrows (⟶) and double arrows (⟹) represent plain and masked values.

Raccoon Signing With Mask Compression “API Calls”

Input: A masked key $[\![\mathbf{sk}]\!]$, a message $\mathbf{msg}$
Output: A signature $\mathbf{sig}$ of $\mathbf{msg}$ under $\mathbf{sk}$

```
1:  $[\![\mathbf{r}]\!] \leftarrow (\mathcal{R}_q^\ell)^d$                                ▷ A random mask set.
2:  $[\![\mathbf{u}]\!] := \mathbf{A} \cdot [\![\mathbf{r}]\!]$                          ▷ LoadShare, MaskCompress.
3:  $[\![\mathbf{u}]\!] \leftarrow \mathsf{Refresh}([\![\mathbf{u}]\!])$                   ▷ LoadShare (not SNI).
4:  $[\![\mathbf{w}]\!] := \mathsf{ApproxShift}_{q \to q_w}([\![\mathbf{u}]\!])$           ▷ MaskCompress.
5:  $\mathbf{w} := \mathsf{Decode}([\![\mathbf{w}]\!])$                                  ▷ FullLoadShare.
6:  $\mathbf{c}_{\mathsf{hash}} := H(\mathbf{w}, \mathbf{msg})$                           ▷ Not masked.
7:  $\mathbf{c}_{\mathsf{poly}} := \mathsf{ChalPoly}(\mathbf{c}_{\mathsf{hash}})$            ▷ Not masked.
8:  $[\![\mathbf{s}]\!] \leftarrow \mathsf{Refresh}([\![\mathbf{s}]\!])$                   ▷ LoadShare (not SNI).
9:  $[\![\mathbf{r}]\!] \leftarrow \mathsf{Refresh}([\![\mathbf{r}]\!])$                   ▷ LoadShare (not SNI).
10: $[\![\mathbf{z}]\!] := \mathbf{c}_{\mathsf{poly}} \cdot [\![\mathbf{s}]\!] + [\![\mathbf{r}]\!]$   ▷ MaskCompress.
11: $\mathbf{z} := \mathsf{Decode}([\![\mathbf{z}]\!])$                                  ▷ FullLoadShare.
12: $\mathbf{y} := \mathbf{A} \cdot \mathbf{z} - p_t \cdot \mathbf{c}_{\mathsf{poly}} \cdot \mathbf{t}$
        ▷ Unmasked to the end.
13: $\mathbf{y}^{\mathsf{top}} := \lfloor \mathbf{y} \rceil_{q \to q_w}$
14: $\mathbf{h} := \mathbf{w} - \mathbf{y}^{\mathsf{top}}$
15: if $(\|\mathbf{h}\|_2 > B_2)$ or
        $(\|\mathbf{h}\|_\infty > B_\infty)$ then
16:     goto 1
17: return $\mathbf{sig} := (\mathbf{c}_{\mathsf{hash}}, \mathbf{z}, \mathbf{h})$
```

Annotated in comments with mask compression gadgets for loading and storing shares where needed. Access to shares is in order $0, 1, .., d-1$ at every masked step 1–11.

System Design

- The project was built by modifying and expanding components from an existing (commercial) first-order masked Kyber & Dilithium hardware implementation.

- We already had fast seed expansion:
  For SHAKE-128/256 based $\textbf{Sample}_G$ the unit has a 24-cycle Keccak permutation, connected to uniform (mod q) rejection sampling. DMA output to work memory.

- Other: RV32C control core, masking random number generator, communication peripherals, and a lattice unit with direct memory access via a 64-bit bus. Fast, native support for Raccoon's mod q and NTT arithmetic, as well as masking.

- High-level algorithm was implemented in C to run on the RISC-V core on the target. The cryptographic hardware is accessed via memory-mapped control registers.

Hardware Realization and Performance

- On an XC7A100T FPGA: 10,638 Slice LUTs (16.78%), 4,140 Slice registers / Flip Flops, (3.26%) and only 3 DSPs (as logic was used for multipliers – ASIC-oriented design). Rated for 78.3 MHz, and ran with 24ns (41.7 MHz) clock cycle for trace acquisition.

- Operated well with 128 kB of SRAM (Block RAM), while at least 2000 kB would have been required at $d = 32$ without compression. Artix 7 doesn't have such memory resources. The secret key $[\![\mathbf{s}]\!]$ alone shrunk from 294 kB to 12.1 kB.

Table 1: FPGA cycle counts at various side-channel security levels.

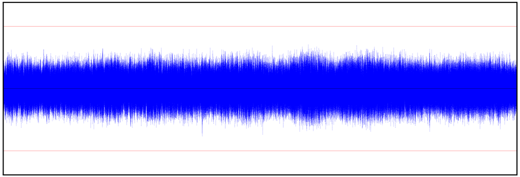
Algorithm	Shares	Keygen()	Sign()	Verif()
Raccoon-128	$d = 2$	1,366,000	2,402,000	1,438,000
Raccoon-128	$d = 4$	2,945,000	3,714,230	1,433,034
Raccoon-128	$d = 8$	6,100,000	6,345,000	1,389,000
Raccoon-128	$d = 16$	12,413,000	11,605,000	1,389,000
Raccoon-128	$d = 32$	25,073,000	22,160,000	1,393,000

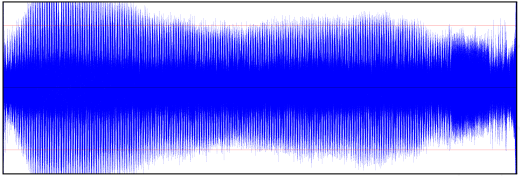
Leakage Assessment (“TVLA Sign-Off”)

- The FPGA target was on a ChipWhisperer CW305 board for good-quality power trace acquisition. PicoScope 2208B oscilloscope at the 24ns sampling frequency (same as target clock); more than 22 million samples per Raccoon-128 trace at $d = 32$.

- At N=20,000 traces, the maximum t-value was 5.55, well under the threshold and corresponding to P-value 0.47. At N=10,000 traces, the test result was $t = 5.43$. TVLA only detects first-order leakage so we tried $d = 2$ too ($N = 200,000$ traces.)

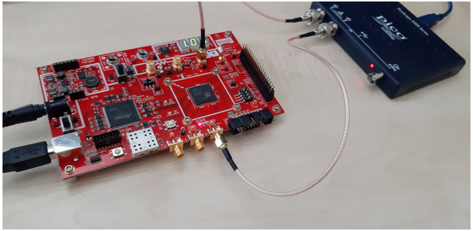
- We also verified that leakage detection is functional by disabling countermeasures in various ways; spikes rapidly appear in those cases.

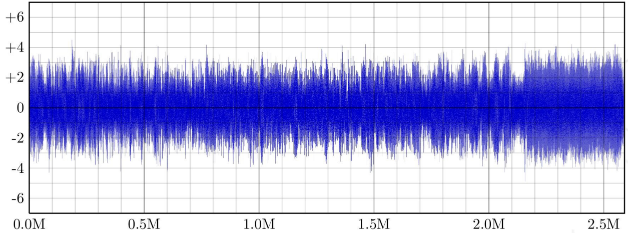




Conclusions

- Mask Compression is a technique where one stores random "seed" for some large masking shares and expands the seeds only when needed by computation.
- Mask Compression allows provable side-channel security properties to be retained.
- Practical – sometimes necessary – especially for high-order masking of PQC.
- Experiments: FPGA implementation of Order-31 Raccoon-128 signature function, with significant resource savings. ISO 17825 / "TVLA" style leakage assessment.





200,000 TVLA t-traces of Raccoon-128 $d = 2$ with mask compression.