Toward Boolean Functions in Post –Quantum Cryptography

This talk explores the role of Boolean functions in post-quantum cryptography, with a focus on their use in practical implementations. We focus on post-quantum cryptography and side-channel attacks focusing to KYBER and Dilithium algorithms. The presentation then examines the arithmetic operations underlying these schemes, emphasizing their Boolean structure, masking techniques, and transformations between arithmetic and Boolean domains. We discuss an approach to implementing arithmetic operations via superposition of low-bit-width partial products

Danila Gorodecky (Gorodetsky)

INESC-ID, Instituto Superior Tecnico, Universidade de Lisboa

Two parties can establish a secure channel over a fully public network, without sharing a secret in advance

RSA (Rivest–Shamir–Adleman) security relies on integer factorization of $A \cdot B = N (\mathrm{mod}\ P)$, where A, B, N and P are up to 4096-bit integers

ECC (Elliptic Curve Cryptography) security relies on $A \cdot k = Q (\mathrm{mod}\ P)$, A, k, Q and P are up to 512-bit integers

Implementation: HTTPS/TLS, cryptocurrency (Bitcoin, blockchain), Signal, WhatsApp, Telegram, VPNs, digital signatures, e-passports, SSH, etc.

P. Shor (1994) showed that a sufficiently large quantum computer can solve both factorization (RSA) and the discrete logarithm (ECC) in polynomial time

Today's quantum processors operate with hundreds to thousands of physical qubits.

Cryptographically relevant attacks require millions of error-corrected logical qubits (estimated 15–20 years away).

Post-quantum cryptography is the design of cryptographic algorithms that remain secure against both classical and quantum computers, based on mathematical problems for which no efficient quantum algorithm is known

Side-Channel Attack

An algorithm can be mathematically unbreakable yet physically exploitable

What leaks:

- **Power consumption** — an attacker connects an oscilloscope to a smart card. They send N known plaintexts and record N power traces
- **Timing** — execution time depends on secret-dependent branches and memory accesses
- **Electromagnetic emission** — radiation from the chip carries data-dependent patterns
- **Cache behavior** — memory access patterns reveal key-dependent indices

PQC protects against quantum attacks, but not against side-channel leakage

recovering a shared secret from a ciphertext using a secret key

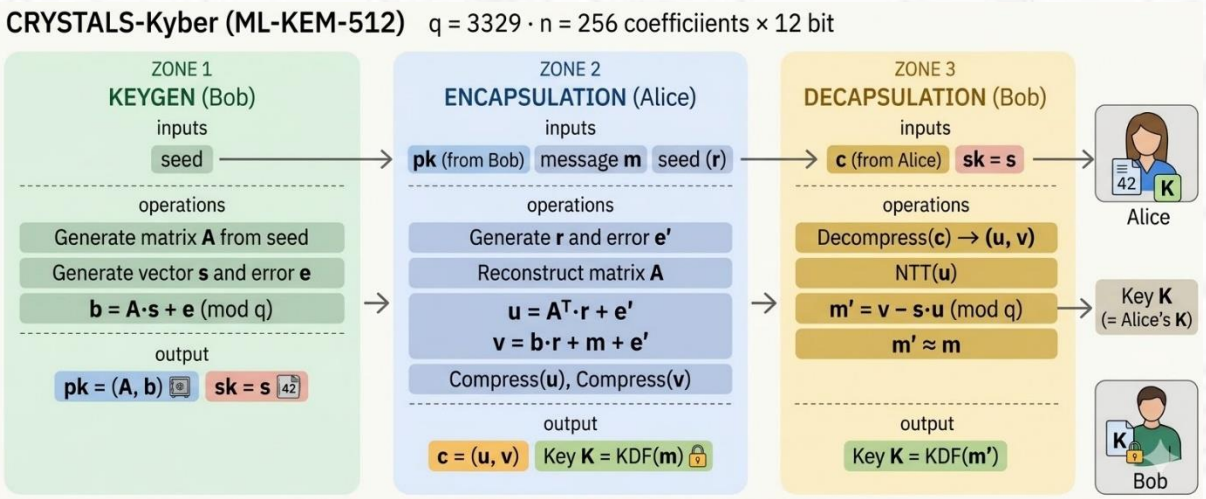
In decapsulation, the device processes a secret key together with attacker-controlled input, creating an scenario for power analysis attacks, making unprotected implementations as vulnerable as classical schemes like RSA.

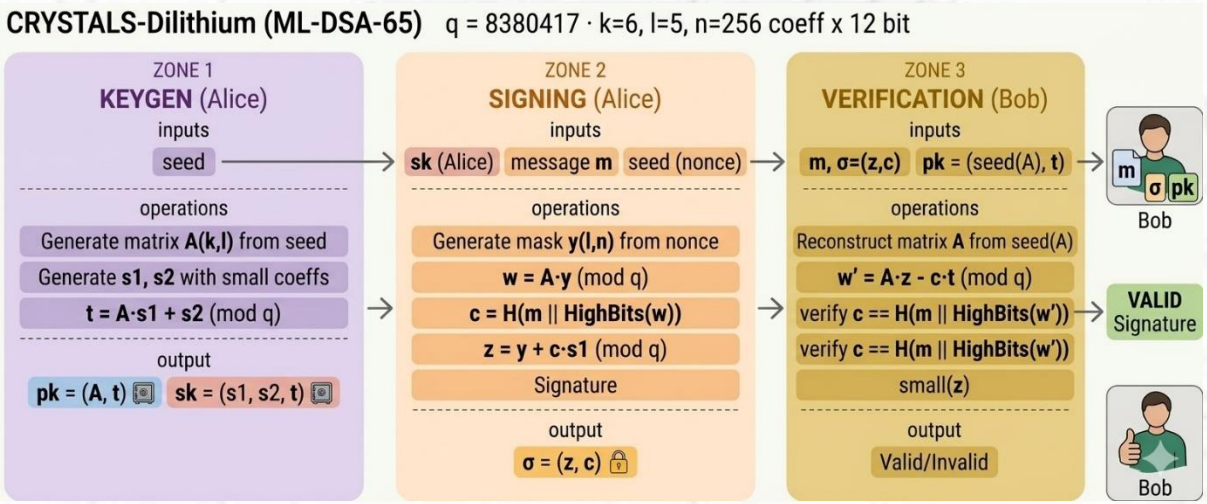
A 128-bit key is recovered in minutes from a few thousand traces.

Boolean masking — splitting every secret value into d random shares before any computation. Cost scales as $O(d)$ for XOR ($\oplus$) operations, $O(d^2)$ for AND operations.
Minimizing AND-gate count in arithmetic operations is the central problem!

NIST Post-Quantum Cryptography (PQC): KYBER and Dilithium

RSA and ECC are mathematically broken by quantum computing.
The transition to PQC is not optional, but it is a matter of timing





Operation	Comments
$A \cdot c \pmod q$	Matrix × vector: multiply 256-coeff polynomials entry by entry based on **NTT** (Number Theoretic Transform) — the fast algorithm for multiplying polynomials. Without it: 256² = 65 536 multiplications, with NTT: 256 · $\log_2(256)$ = 2 048. Cost: O(n log n) instead of O(n²)
Compress $[x \cdot 2^d/q]$	Round a coefficient from 12 bits down to d bits. Discards precision to shrink the ciphertext. Like rounding 3.14159 → 3.1
Decompress $[x \cdot q/2^d]$	Reconstruct approximate 12-bit value from d bits. Small error is tolerable — the noise e already introduced imprecision
HighBits $[w/a]$	Keep only the coarse part of a coefficient. Like keeping only "hours" from a time value, ignoring minutes
LowBits $w \pmod a$	Keep only the fine remainder. Like keeping only "minutes". Together High+Low = full value
B2A ↔ A2B $x_0 \oplus x_1 \leftrightarrow x_0 + x_1$	Convert secret shares from/to XOR-form to/from addition-form. Needed when switching between bitwise and arithmetic operations on a masked secret

1 The idea: **never process the secret directly**

`secret x` = `$x_0$ (random)` ⊕ `$x_1$ (random)` ⊕ `$x_2$ (random)`

The secret **x** is split into **d random shares**. Each share alone looks completely random. It reveals nothing about x. The device always works with shares, never with x itself. An attacker observing power consumption sees only random values.

2 Concrete example (d = 2 shares, 4-bit numbers)

secret:	$x = 1011_2 = 11$
random r:	$r = 0110_2 = 6$
share 0:	$x_0 = x \oplus r = 1011 \oplus 0110 = 1101$ (looks random)
share 1:	$x_1 = r = 0110$ (looks random)
restore:	$x_0 \oplus x_1 = 1101 \oplus 0110 = 1011$ $= 11$ ✓

3 Cost of operations on masked values

XOR ⊕ cheap

$z = x \oplus y$

$z_i = x_i \oplus y_i$

Each share computed independently — **no interaction between shares**.

Total cost: **d operations** → O(d)

AND · expensive

$z = x \cdot y$

$z_i = x_i \cdot y_i \oplus \Sigma\, r_{ij}$

Shares of x and y **must interact** — each pair (i,j) needs a fresh random bit r_{ij}.

Total cost: **d^2 operations** → $O(d^2)$

4 Why AND is expensive — the cross-term problem

XOR of two masked values

$(x_0 \oplus x_1) \oplus (y_0 \oplus y_1) = (x_0 \oplus y_0) \oplus (x_1 \oplus y_1)$

Each share only needs its own pair. No cross-terms. Safe to compute separately.

AND of two masked values

$(x_0 \oplus x_1) \cdot (y_0 \oplus y_1) = x_0y_0 \oplus x_0y_1 \oplus x_1y_0 \oplus x_1y_1$

Cross-terms x_0y_1 and x_1y_0 appear — they correlate with both x and y. Each must be randomized with a fresh r_{ij}.

$$Cost(d) = X \cdot d + A \cdot d^2$$

X = number of XOR gates · A = number of AND gates · d = masking order

Minimizing A directly reduces the cost of any masked implementation.

For d = 4: cutting A in half saves 8× more operations than cutting X in half.

Application Table

Optimized operation	Kyber	Dilithium	AES
modular multiplication *(A · B) mod q*	NTT butterfly: ω · b (mod 3329). ~1024 multiplications per NTT, multiple NTT calls per decapsulation	NTT butterfly: ω · b (mod 8380417). Same structure as ML-KEM. 23-bit modulus = larger Boolean functions, greater optimization potential	—
modular reduction *A mod q*	Internal to NTT, Compress, B2A/A2B mask conversions	Internal to NTT, rejection sampling.	Reduction mod m(x) in S-box tower-field decomposition
multiplication by constant (*C*) *(C · A) (mod q)*	NTT twiddle factors: $\omega^i \cdot b \pmod{3329}$. Constants known at design time	NTT twiddle factors: $\omega^i \cdot b \pmod{8380417}$. Constants known at design time	—
division by Constant (*d*) *A / d = {Q, R}*	Compress: $\lceil 2^d/q \cdot x \rceil$ Division by q = 3329	—	—
B2A and A2B mask conversion	Compress: A2B after NTT. Decompress: B2A before NTT. CCA comparison: A2B. ~256 conversions per polynomial	Similar conversions for rejection sampling and signature verification	—

high-level representation

logic design

“well-suited” algebraic formula

Boolean representations

splitting input and output operands into sub-words of smaller bit-width

representation of the formula as a superposition of “simple” arithmetic functions

properties of Boolean functions

Boolean representations

Boolean minimizations

Efficient representation of arithmetic operation

Arithmetic Operations

- Modular operations (for an arbitrary P)
  - constant multiplication $((A \cdot const)(\mathrm{mod}\ P) = R)$*
  - modular multiplication $((A \cdot B)(\mathrm{mod}\ P) = R)$*,**
  - modular addition $((A + B)(\mathrm{mod}\ P) = S)$
  - modular reduction $(A(\mathrm{mod}\ P) = R)$*
  - Modular division $((A \cdot x)(\mathrm{mod}\ P) \equiv 1)$
  - Residue Number System (RNS)
- Standard operations
  - constant multiplication $(A \cdot const = R)$****
  - multiplication $(A \cdot B = R)$*****
  - addition $(A + B = S)$
  - division by constant $(A = constant \cdot Q + residue)$**
- Floating-point calculations
  - multiplication $(A \cdot B = R)$
  - addition $(A + B = S)$
  - normalization

*D. Gorodecky, L. Sousa, “Modular Arithmetic Based on Boolean Functions: A Divide and Conquer Approach” // IEEE Access, Vol. 13, Oct., 2025, pp. 186383-186396.
**D. Gorodecky, L. Sousa, "Scalable architecture of constant division on FPGA" // Proceedings of the 30th IEEE Symposium on Computer Arithmetic, Sep. 4-6, 2023, Portland, Oregon, USA.
***D. Gorodecky, L.Sousa, “Two-Operand Modular Multiplication to Small Bit Ranges” // Advanced Boolean Techniques, Springer Nature Switzerland, 2023, p. 111-122.
****D. Gorodecky, P. Bibilo, “Constant Multiplication Based on Boolean Minimization” // Proceedings of the 14th International Workshop on Boolean Problems, Bremen, Germany, Sept. 24-25, 2020.
*****D. Gorodecky “Design of Multipliers Using Fourier Transformations” // Further Improvements in the Boolean Domain, Cambridge Scholars Publishing, UK, 2018, Section 3.4, pp. 240-252.

6-stage scalable methodology

split operation into partial products of smaller sub-words

Arithmetic Decomposition

$$F = A_1 \cdot B_1 \cdot C_1 + A_2 \cdot B_2 \cdot C_2 + \ldots + A_w \cdot B_w \cdot C_w$$

represent each partial product as a Boolean function $g_i(X)$

Boolean Mapping

$$A_1 \cdot B_1 \cdot C_1 = g_1(X_1),\ A_2 \cdot B_2 \cdot C_2 = g_2(X_2),\ \ldots,\ A_w \cdot B_w \cdot C_w = g_w(X_w)$$

Boolean Decompos./Minimiz.

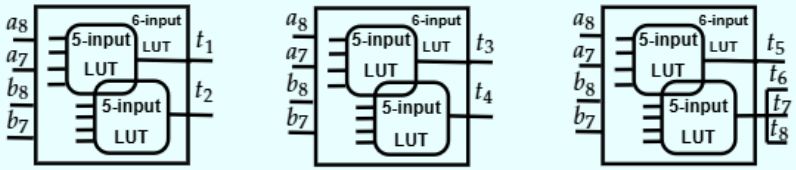
$$F = g_1(h_1(X_1), \ldots, h_v(X_v)) \circ g_2(h_1(X_1), \ldots, h_v(X_v)) \circ \ldots \circ g_w(h_1(X_1), \ldots, h_v(X_v))$$

$$\text{"}\circ\text{"} \in \{+, -, \cdot\}$$

minimize Boolean functions using Reed-Muller, Karnaugh, or technology-dependent methods

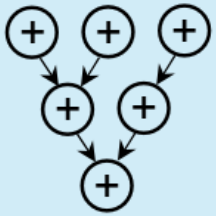
map minimized functions to FPGA LUTs or ASIC standard cells

LUT/Library Mapping



Adders Tree

ballanced parallel additions



combine partial results using balanced parallel addition

Result Integration

concatenation

$$R - P \quad if\ R \geq P$$

modular correction

when required

concatenate sub-results and apply modular correction if R ≥ P

* D. Gorodecky and L. Sousa, “Efficient Arithmetic on FPGA”, DATE, 2026, Verona, Italy

$$A = (a_{12}a_{11} \dots a_1) \qquad B = (b_{12}b_{11} \dots b_1)$$

$$A_1 = (a_4a_3a_2a_1) \qquad B_1 = (b_4b_3b_2b_1)$$
$$A_2 = (a_8a_7a_6a_5) \qquad B_2 = (b_8b_7b_6b_5)$$
$$A_3 = (a_{12}a_{11}a_{10}a_9) \qquad B_3 = (b_{12}b_{11}b_{10}b_9)$$

$A \cdot B = R(\text{mod } 3329) =$

$$A_1 \cdot B_1 + A_1 \cdot B_2 \cdot 2^4 + A_1 \cdot B_3 \cdot 2^8 +$$
$$A_2 \cdot B_1 \cdot 2^4 + A_2 \cdot B_2 \cdot 2^8 + A_2 \cdot B_2 \cdot 2^{12} +$$
$$A_3 \cdot B_1 \cdot 2^8 + A_3 \cdot B_2 \cdot 2^{12} + A_3 \cdot B_3 \cdot 2^{16} =$$

High – Level Representation

$$A_1 \cdot B_1 + A_1 \cdot B_2 \cdot 2^4 + A_1 \cdot B_3 \cdot 2^8 +$$
$$A_2 \cdot B_1 \cdot 2^4 + A_2 \cdot B_2 \cdot 2^8 + A_2 \cdot B_2 \cdot 767 +$$
$$A_3 \cdot B_1 \cdot 2^8 + A_3 \cdot B_2 \cdot 767 + A_3 \cdot B_3 \cdot 2285$$

Building blocks:
8 input Boolean functions

$A_1 \cdot B_1$ $\qquad$ $A_1 \cdot B_2 \cdot 2^4$

$A_1 \cdot B_3 \cdot 2^8$ $\qquad$ $A_2 \cdot B_2 \cdot 767$

$A_3 \cdot B_3 \cdot 2285$

$$A = (a_8 a_7 \dots a_1) \qquad B = (b_8 b_7 \dots b_1)$$

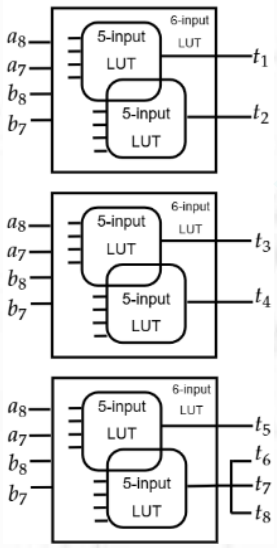
$$(A \cdot B)(\mathrm{mod}\ 241) =$$

$$((a_3a_2a_1)\cdot(b_3b_2b_1) + (a_3a_2a_1)\cdot(b_6b_5b_4)\cdot 2^3 + (a_3a_2a_1)\cdot(b_8b_7)\cdot 2^6 +$$

$$(a_6a_5a_4)\cdot(b_3b_2b_1)\cdot 2^3 + (a_6a_5a_4)\cdot(b_6b_5b_4)\cdot 2^6 + (a_6a_5a_4)\cdot(b_8b_7)\cdot 30 +$$

$$(a_8a_7)\cdot(b_3b_2b_1)\cdot 2^6 + (a_8a_7)\cdot(b_6b_5b_4)\cdot 30 + (a_8a_7)\cdot(b_8b_7)\cdot 240)\ (\mathrm{mod}\ 241) =$$

$$R = (r_8\, r_7 \dots r_1)$$



$$t_1 = a_8 \cdot \left(b_8 \cdot \overline{b_7} \vee \overline{a_7} \cdot b_7\right) \vee\ a_7 \cdot b_8 \cdot \overline{b_7} =$$
$$\overline{b_7} \oplus \overline{b_8} \cdot \overline{b_7} \oplus \overline{a_7} \oplus \overline{a_7} \cdot \overline{b_7} \oplus$$
$$\overline{a_8} \cdot \overline{a_7} \oplus \overline{a_8} \cdot \overline{a_7} \cdot \overline{b_8} \cdot \overline{b_7},$$

$$t_2 = a_8\ \cdot b_7 \cdot \left(\overline{a_7}\ \vee \overline{b_8}\ \right) \vee a_7 \cdot b_8 \cdot \left(\overline{a_8} \vee \overline{b_7}\right) =$$
$$a_7 \cdot b_8 \oplus a_8 \cdot b_7,$$

$$t_3 = a_8\ \cdot \left(\overline{b_8} \cdot\ b_7\ \vee \overline{a_7} \cdot b_8 \cdot \overline{b_7}\right) \vee \overline{a_8} \cdot a_7 \cdot b_8 =$$
$$a_7 \cdot b_8 \oplus a_8 \cdot b_7 \oplus a_8 \cdot b_8 \oplus a_8 \cdot b_8 \cdot b_8 \cdot b_8,$$

$$t_4 = a_8 \cdot (b_8\ \vee b_7) \vee a_7 \cdot b_8 =$$
$$a_7 \cdot b_8 \oplus a_8 \cdot b_7 \oplus a_8 \cdot b_8 \oplus a_8 \cdot b_8 \cdot b_7 \oplus a_8 \cdot a_7 \cdot b_8,$$

$$t_5 = \overline{a_8} \cdot a_7 \cdot \overline{b_8} \cdot b_7 =$$
$$a_7 \cdot b_7 \oplus a_7 \cdot b_8 \cdot b_7 \oplus a_8 \cdot a_7 \cdot b_7 \oplus a_8 \cdot a_7 \cdot b_8 \cdot b_7,$$

$$t_6 = t_7 = t_8 = (a_8\ \vee a_7) \cdot (b_8 \vee b_7) =$$
$$1 \oplus \overline{b_8} \cdot \overline{b_7}\ \oplus \overline{a_8} \cdot \overline{a_7} \oplus \overline{a_8} \cdot \overline{a_7} \cdot \overline{b_8} \cdot \overline{b_7}$$

$$((a_8a_7)\cdot(b_8b_7)\cdot 240)(\mathrm{mod}\ 241) = T(t_8t_7 \dots t_1)$$

decimal $(a_8a_7)\cdot(b_8b_7)\cdot 240$	a_8 a_7	b_8 b_7	t_8 t_7 t_6 t_5 t_4 t_3 t_2 t_1	decimal T
$0\cdot 0\cdot 240$	0 0	0 0	0 0 0 0 0 0 0 0	0
$0\cdot 1\cdot 240$	0 0	0 1	0 0 0 0 0 0 0 0	0
$0\cdot 2\cdot 240$	0 0	1 0	0 0 0 0 0 0 0 0	0
$0\cdot 3\cdot 240$	0 0	1 1	0 0 0 0 0 0 0 0	0
$1\cdot 0\cdot 240$	0 1	0 0	0 0 0 0 0 0 0 0	0
$1\cdot 1\cdot 240$	0 1	0 1	1 1 1 1 0 0 0 0	240
$1\cdot 2\cdot 240$	0 1	1 0	1 1 1 0 1 1 1 1	239
$1\cdot 3\cdot 240$	0 1	1 1	1 1 1 0 1 1 1 0	238
$2\cdot 0\cdot 240$	1 0	0 0	0 0 0 0 0 0 0 0	0
$2\cdot 1\cdot 240$	1 0	0 1	1 1 1 0 1 1 1 1	239
$2\cdot 2\cdot 240$	1 0	1 0	1 1 1 0 1 1 0 1	237
$2\cdot 3\cdot 240$	1 0	1 1	1 1 1 0 1 0 1 1	235
$3\cdot 0\cdot 240$	1 1	0 0	0 0 0 0 0 0 0 0	0
$3\cdot 1\cdot 240$	1 1	0 1	1 1 1 0 1 1 1 0	238
$3\cdot 2\cdot 240$	1 1	1 0	1 1 1 0 1 0 1 1	235
$3\cdot 3\cdot 240$	1 1	1 1	1 1 1 0 1 0 0 0	232

*D. Gorodecky, L. Sousa, “Modular Arithmetic Based on Boolean Functions: A Divide and Conquer Approach” // IEEE Access, Vol. 13, Oct., 2025, pp. 186383-186396.

Reed-Muller form:{XOR, AND}*,**,***

$$\mathbf{X(mod\ 5) = S^*}$$

$\mathbf{X} = (x_5x_4x_3x_2x_1)$ 5-bit number

$\mathbf{S} = (S_3S_2S_1)$ 3-bit number

VHDL	Reed-Muller form
S(1) < = x(1) xor (x(1) and x(3)) xor (x(2) and x(3)) xor (x(1)and x(2) and x(3)) xor x(4) xor (x(2) and x(4)) xor (x(3) and x(4)) xor (x(1) and x(3) and x(4)) xor x(5) xor (x(3) and x(5)) xor (x(1) and x(3) and x(5)) xor (x(2) and x(3) and x(5)) xor (x(1) and x(2) and x(3) and x(5)) xor (x(1) and x(4) and x(5)) xor (x(1) and x(2) and x(4) and x(5)) xor (x(3) and x(4) and x(5)) xor (x(1) and x(2) and x(3) and x(4) and x(5));	$S_1 = x_1 \oplus x_1 \cdot x_3 \oplus x_2 \cdot x_3 \oplus x_1 \cdot x_2 \cdot x_3 \oplus x_4 \oplus x_2 \cdot x_4 \oplus x_3 \cdot x_4 \oplus x_1 \cdot x_3 \cdot x_4 \oplus x_5 \oplus x_3 \cdot x_5 \oplus x_1 \cdot x_3 \cdot x_5 \oplus x_2 \cdot x_3 \cdot x_5 \oplus x_1 \cdot x_2 \cdot x_3 \cdot x_5 \oplus x_1 \cdot x_4 \cdot x_5 \oplus x_1 \cdot x_2 \cdot x_4 \cdot x_5 \oplus x_3 \cdot x_4 \cdot x_5 \oplus x_1 \cdot x_2 \cdot x_3 \cdot x_4 \cdot x_5$
S(2) <= x(2) xor (x(2) and x(3)) xor (x(1) and x(2) and x(3)) xor x(4) xor (x(1) and x(4)) xor (x(1) and x(2) and x(4)) xor (x(1) and x(3) and x(4)) xor (x(2) and x(3) and x(4)) xor (x(1) and x(5)) xor (x(1) and x(3) and x(5)) xor (x(2) and x(3) and x(5)) xor (x(1) and x(2) and x(3) and x(5)) xor (x(4) and x(5)) xor (x(2) and x(4) and x(5)) xor (x(3) and x(4) and x(5)) xor (x(1) and x(3) and x(4) and x(5));	$S_2 = x_2 \oplus x_2 \cdot x_3 \oplus x_1 \cdot x_2 \cdot x_3 \oplus x_4 \oplus x_1 \cdot x_4 \oplus x_1 \cdot x_2 \cdot x_4 \oplus x_1 \cdot x_3 \cdot x_4 \oplus x_2 \cdot x_3 \cdot x_4 \oplus x_1 \cdot x_5 \oplus x_1 \cdot x_3 \cdot x_5 \oplus x_2 \cdot x_3 \cdot x_5 \oplus x_1 \cdot x_2 \cdot x_3 \cdot x_5 \oplus x_4 \cdot x_5 \oplus x_2 \cdot x_4 \cdot x_5 \oplus x_3 \cdot x_4 \cdot x_5 \oplus x_1 \cdot x_3 \cdot x_4 \cdot x_5$
S(3) <= x(3) xor (x(1) and x(3)) xor (x(2) and x(3)) xor (x(1) and x(2) and x(3)) xor (x(1) and x(4)) xor (x(1) and x(2) and x(4)) xor (x(3) and x(4)) xor (x(1) and x(2) and x(3) and x(4)) xor (x(1) and x(2) and x(5)) xor (x(3) and x(5)) xor (x(1) and x(3) and x(5)) xor (x(2) and x(3) and x(5)) xor (x(4) and x(5)) xor (x(2) and x(4) and x(5)) xor (x(1) and x(2) and x(4) and x(5)) xor (x(2) and x(3) and x(4) and x(5)),	$S_3 = x_3 \oplus x_1 \cdot x_3 \oplus x_2 \cdot x_3 \oplus x_1 \cdot x_2 \cdot x_3 \oplus x_1 \cdot x_4 \oplus x_1 \cdot x_2 \cdot x_4 \oplus x_3 \cdot x_4 \oplus x_1 \cdot x_2 \cdot x_3 \cdot x_4 \oplus x_1 \cdot x_2 \cdot x_5 \oplus x_3 \cdot x_5 \oplus x_1 \cdot x_3 \cdot x_5 \oplus x_2 \cdot x_3 \cdot x_5 \oplus x_4 \cdot x_5 \oplus x_2 \cdot x_4 \cdot x_5 \oplus x_1 \cdot x_2 \cdot x_4 \cdot x_5 \oplus x_2 \cdot x_3 \cdot x_4 \cdot x_5$

The (minimized) Reed-Muller representation expresses every output bit using **ONLY** AND and XOR operations

- In Boolean masking, XOR is free (linear, applied to each share independently)
- AND is expensive (requires ISW gadget, cost scales as d^2 per gate)
- The number of AND gates in the Reed-Muller form directly determines the cost of the masked implementation
- Optimization reduces the number of AND gates → directly reduces masking cost

*D. Gorodecky, "Reed-Muller Realization of X (mod P)", 2015, arXiv:1504.04773 [cs.DM].
**D. Gorodecky, "Reed-Muller Representation in Arithmetic Operations" // Reed-Muller Workshop 2019, May 24, 2019, Fredericton, New Brunswick, Canada, pp. 53-58.
***D. Gorodecky, "X (mod 2^k − 1) calculation based on Reed-Muller expansions of symmetric Boolean functions" // Reed-Muller Workshop 2023, May 24, 2023, Matsue, Shimane Prefecture, Japan

Experimental Results for FPGA and ASIC

Vivado, Synopsys, Cadence are the main commercial Electronic Design Automation (EDA) tools

RISC-V and **ARM Cortex-M** processors are built from the same **ASIC** cell libraries

Blocks of ASIC standard cells (AND, OR, XOR, MUX, INV, etc) correspond to basic processor operations.

Optimizing gate-level circuits for ASIC libraries simultaneously optimizes for software implementations on RISC-V and ARM Cortex-M

FPGA (Vivado)*

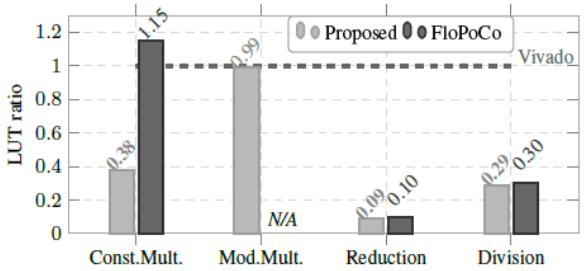


Fig. 1: LUT utilization comparison (normalized to Vivado = 1.0, geometric mean). Lower is better.

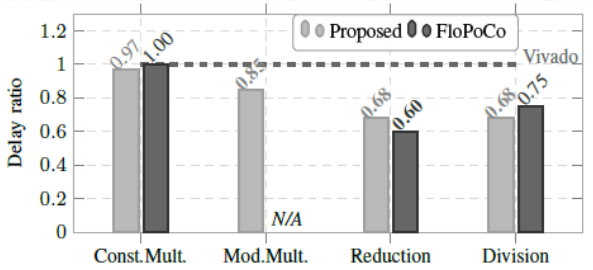


Fig. 2: Critical path delay comparison (normalized to Vivado = 1.0, geometric mean). Lower is better.

ASIC (Synopsys)

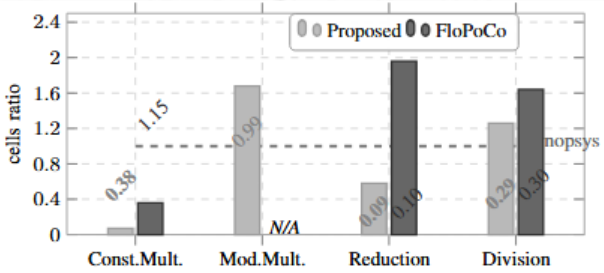


Fig. 1: cells utilization comparison (normalized to Synopsys = 1.0, geometric mean). Lower is better.

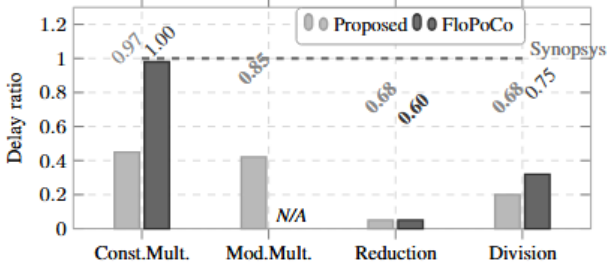


Fig. 2: Critical path delay comparison (normalized to Synopsys = 1.0, geometric mean). Lower is better.

ASIC (Cadence)

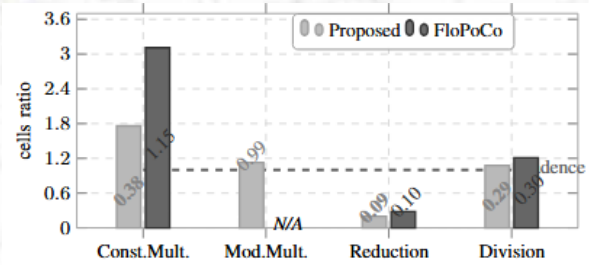


Fig. 1: cells utilization comparison (normalized to Cadence = 1.0, geometric mean). Lower is better.

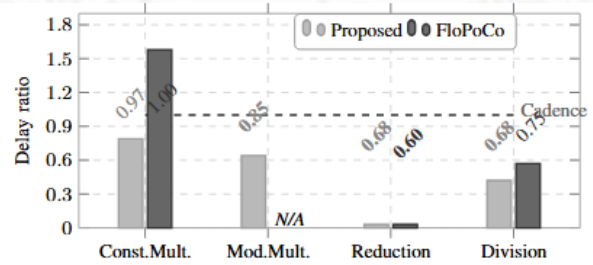


Fig. 2: Critical path delay comparison (normalized to Cadence = 1.0, geometric mean). Lower is better.

* D. Gorodecky and L. Sousa, “Efficient Arithmetic on FPGA”, DATE, 2026, Verona, Italy