PQ Secure

Ensuring Security of Post-Quantum Cryptography on Embedded Devices: Formal Verification and Side-Channel Protection Challenges

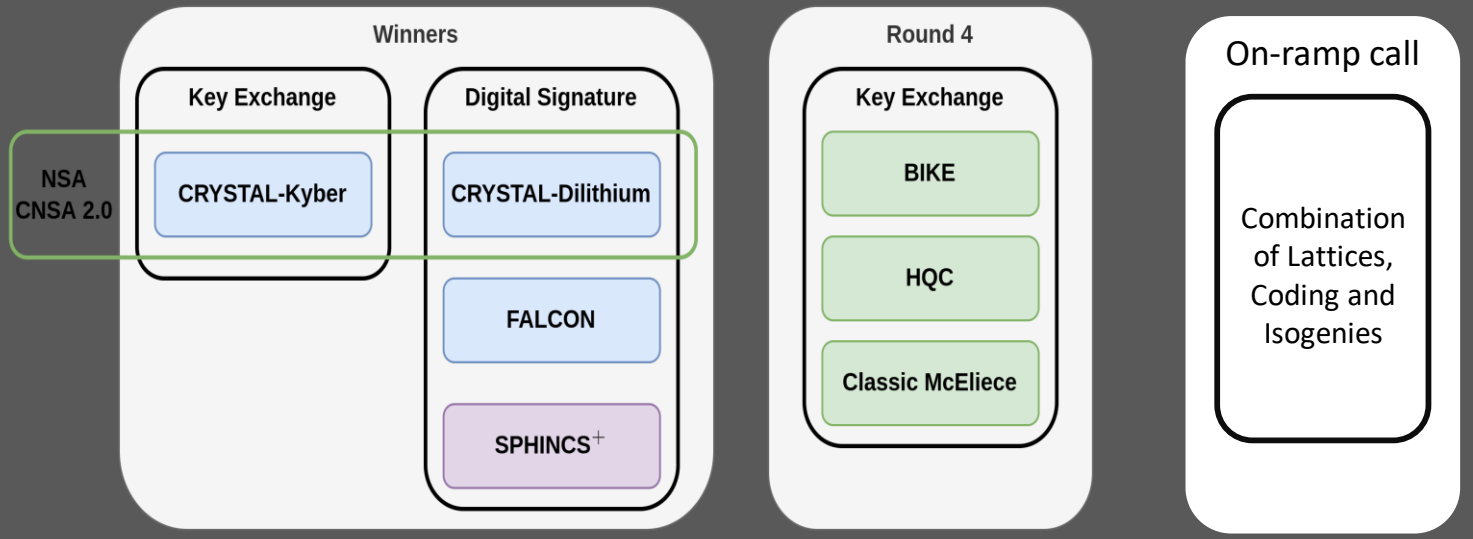
Reza Azarderakhsh

Professor FAU
CEO PQSecure

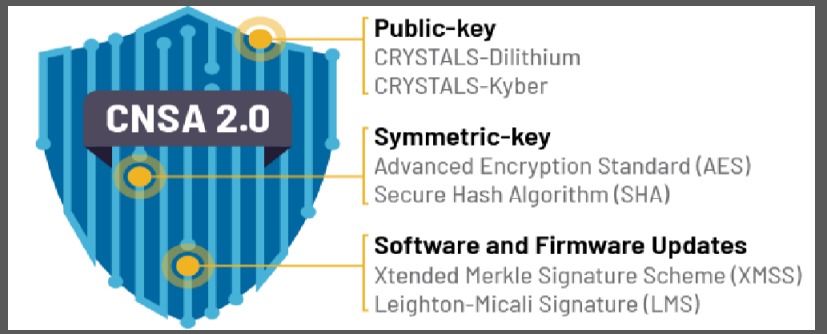
OpenSSL Conference
Prague 2025

NIST Announcement for PQC (August 2023)

PQ Secure



NIST Competition Status



NSA Recommendations for Cryptography
[From CNSA 2.0]

FIPS-203: Module-Lattice-based Key-Encapsulation Mechanism Standard (CRYSTALS-Kyber)
FIPS-204: Module-Lattice-Based Digital Signature Standard (CRYSTALS-Dilithium)
FIPS 205: Stateless Hash-Based Digital Signature Standard (SPHINCS+)
NIST SP 800-208: Recommendation for Stateful Hash-Based Signature Schemes (XMSS/LMS)

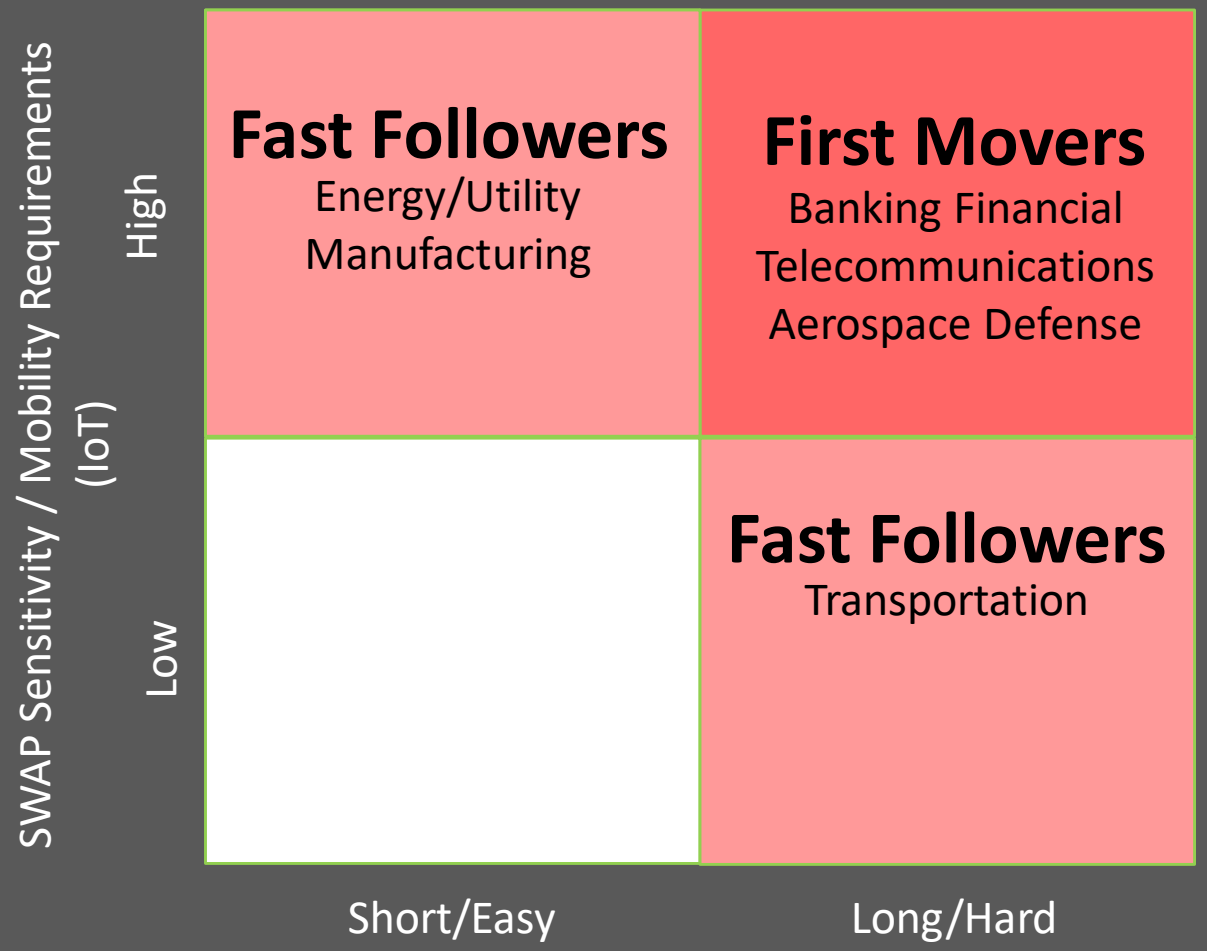
PQC Agility

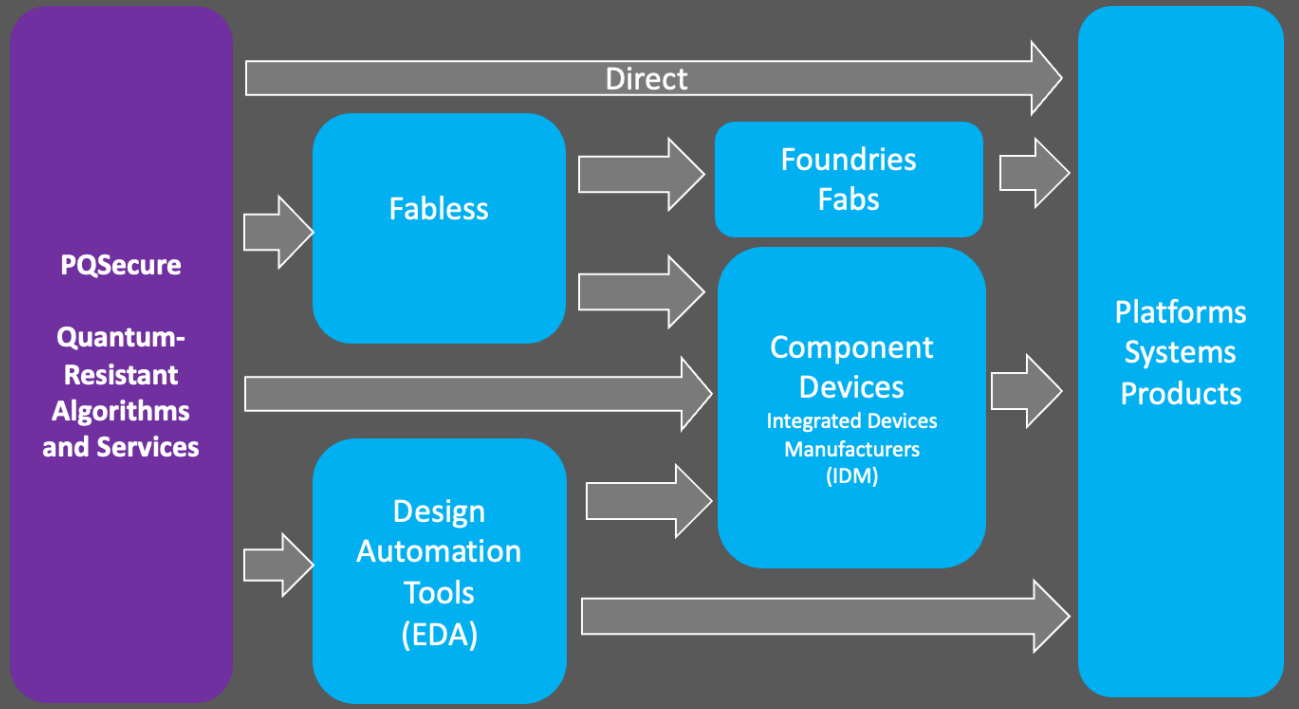
PQ Secure

Type of Agility	Definition
Implementation	The Capability to swiftly configure interfaces and implement updates across various systems or applications
Compliance	The capacity to adapt cryptographic configurations in accordance with compliance requirements.
Security Strength	The capability to dynamically adjust the level of security strength based on configuration, allowing for scalable security measures.
Migration	The capability to transition and convert between cryptographic algorithms seamlessly.
Retirement	Ability to retire obsolete or insecure cryptographic algorithms
Composability	The capability to securely integrate multiple cryptographic primitives for composability.
Platform	Ability to use assured cryptographic algorithms across different platform types
Context	Ability to use a derived cryptographic algorithm policy with the flexibility from system attributes

PQC to Final Products

PQ Secure





CHIPS Act: Agility and HW/SW for PQC

PQ Secure

AUGUST 09, 2022

FACT SHEET: CHIPS and Science Act Will Lower Costs, Create Jobs, Strengthen Supply Chains, and Counter China

Key Strategy 1.1.5: Prioritize hardware integrity and security as an element in co-design strategies across the stack.

In the face of threats from nation-state and criminal adversaries, the potential for the insertion of malicious alterations into components ranging from circuits to software combined with the need to prepare for a post-quantum-computing world make it essential that integrity and cybersecurity be a foundational component of system design.[30,31] Co-design of hardware with software is needed to meet this challenge in a way that provides maximum protection while minimizing the impact on system performance.[32] The design process must allow for iteration between hardware, software, and security constraints. To meet economic and national security needs, security must be incorporated in co-design R&D as a design constraint at the same level as performance.

16

[30] Cybersecurity R&D challenges and goals for hardware and software are described in NITRD's *Federal Cybersecurity Research and Development Strategic Plan* , https://www.nitrd.gov/pubs/Federal-Cybersecurity-RD-Strategic-Plan-2019.pdf.

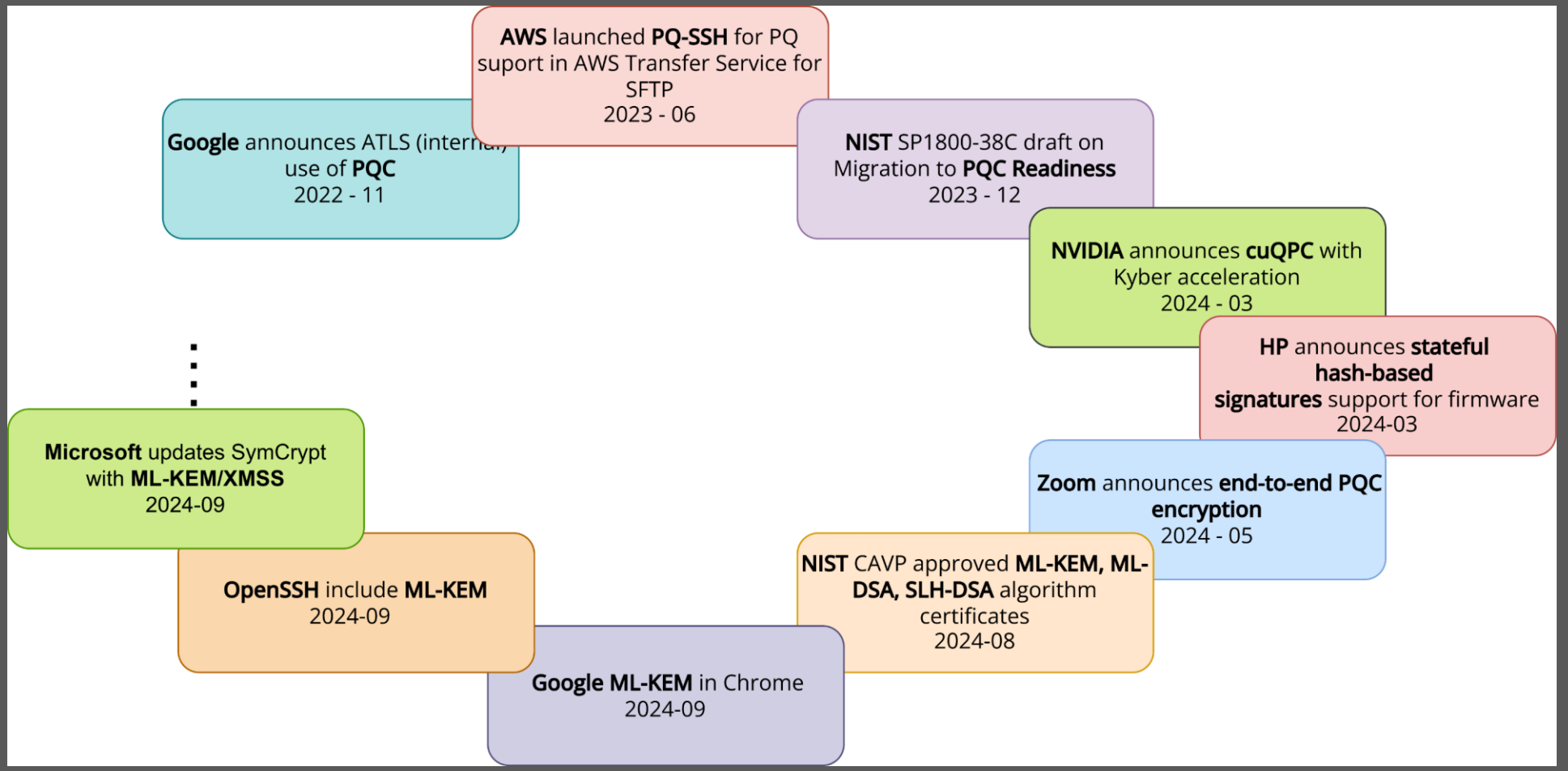
[31] https://www.whitehouse.gov/briefing-room/statements-releases/2022/05/04/national-security-memorandum-on-promoting-united-states-leadership-in-quantum-computing-while-mitigating-risks-to-vulnerable-cryptographic-systems/

[32] See, for example, D. Dangwai et al., *SoK: Opportunities for Software-Hardware-Security Codesign for Next Generation Secure Computing*, arxiv.org/abs/2105.00378.

– 12 –

Pre-finalized PQC Standard Deployments

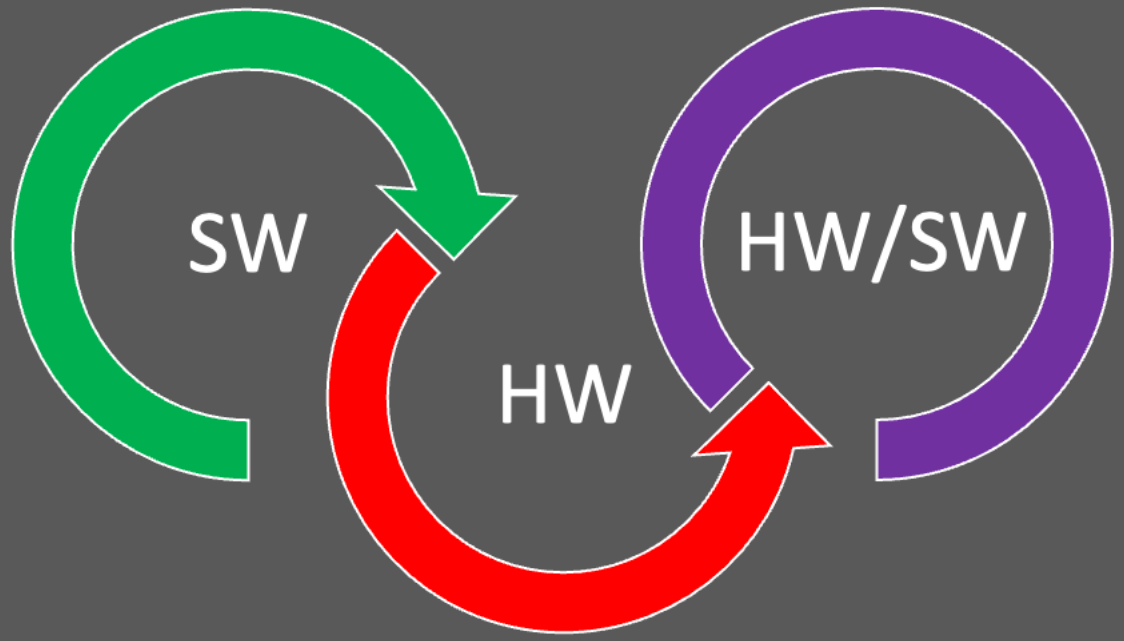
PQ Secure



SW-First and then HW/SW Co-design

PQ Secure

- Reference Codes
- ANSI C
- Device level optimized ASM
- C-to-Rust
- Side-channel protection (?)
- Formal Verification

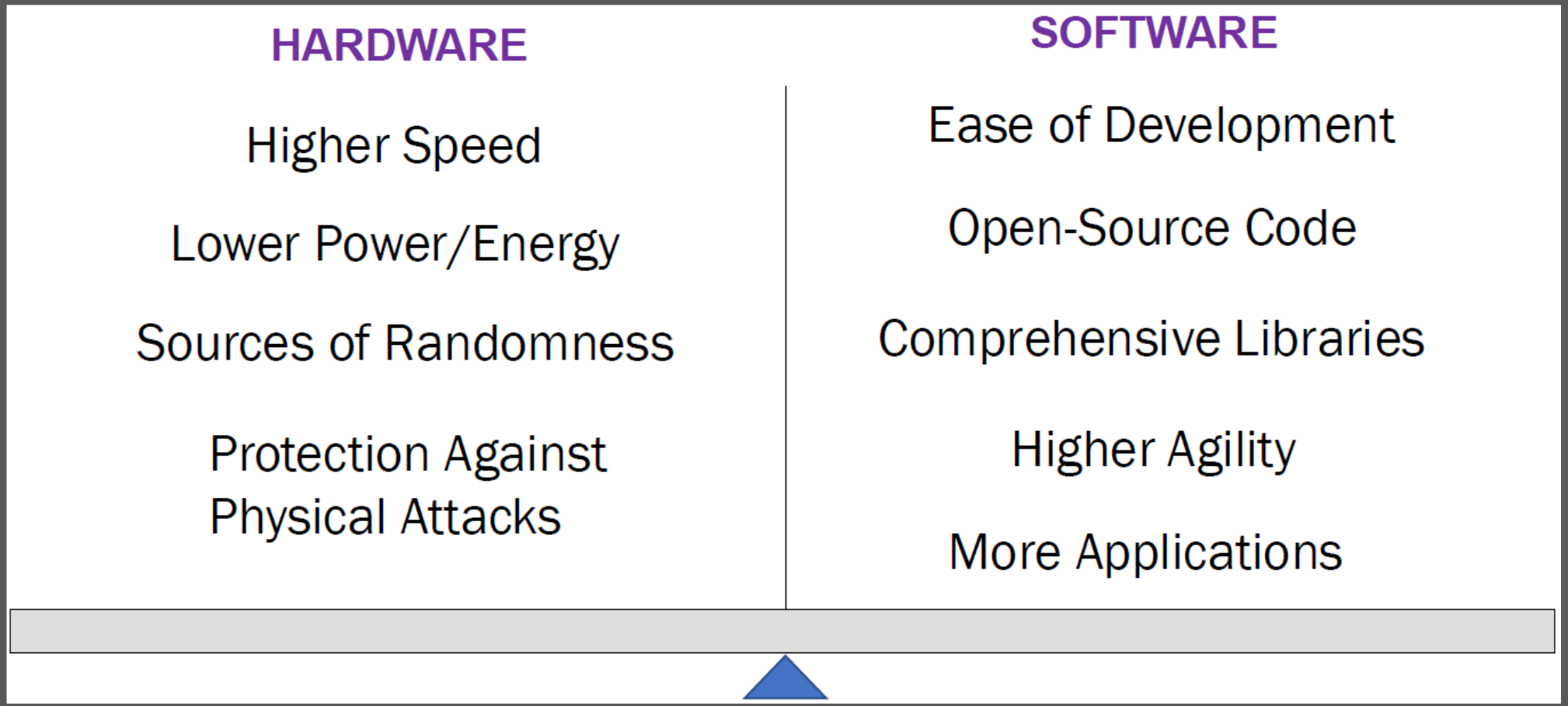


- CPU/FPGA co-design
- RISC-V/FPGA co-design
- ASIC with embedded co-design
- Side-channel protection
- Testing and Verification
- Assurance

- Reference HDL
- FPGA Implementations
- ASIC Synthesis kGE
- Side-Channel Protection
- Verification and Assurance

HW vs. SW

PQ Secure



Everyone Does Cryptography

PQ Secure

Software Libraries: Provide flexible cryptographic functionality for general use.

Dedicated Hardware Instructions: Enhance cryptographic operations with specialized processor commands.

Trusted Platform Modules (TPMs): Secure hardware solutions that store encryption keys and perform cryptographic tasks.

Hardware Security Modules (HSMs): Purpose-built hardware devices designed to protect cryptographic keys and processes.

Secure Enclaves: Isolated, secure environments within processors to handle sensitive data ensure secure execution.

Not Everyone Does Good Cryptography

PQ Secure

- **Incorrect Implementations**:
  - **CVE-2022-0778 in OpenSSL** allowed denial-of-service by exploiting a flaw in certificate parsing.
- **Weak Protocols**:
  - **ROCA (2022)** affected Infineon TPMs, making RSA keys vulnerable to factorization attacks.
- **Bad Random Number Generation**:
  - **CVE-2023-23946 in GitHub Actions**, where weak random numbers led to easily guessable session tokens.
- **Information Leakage Through Side-Channels**:
  - **PLATYPUS attack (2020)** on Intel CPUs exploited power consumption data to extract cryptographic keys.

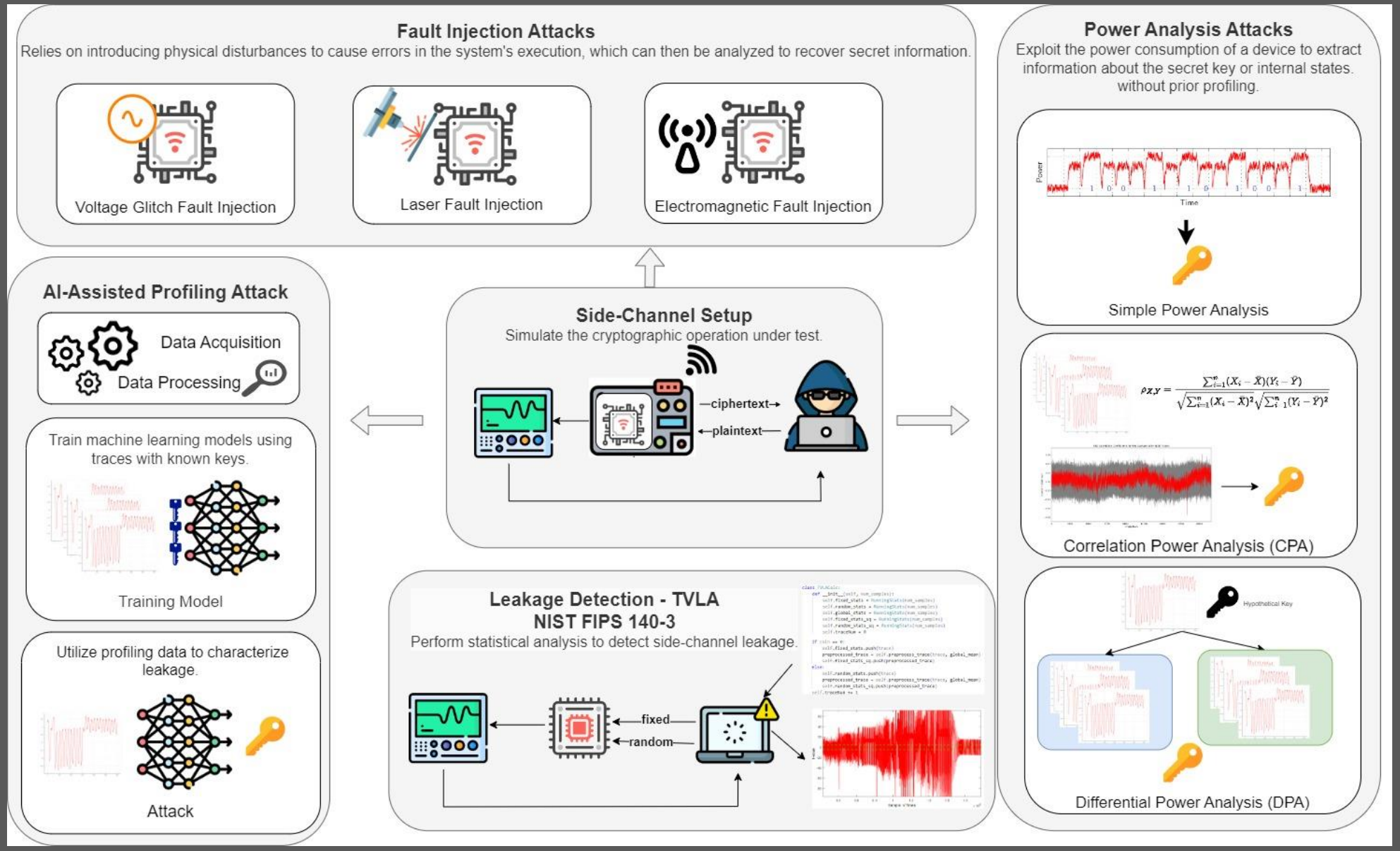
Cryptographic Engineering

PQ Secure

- **I can do it**
  - Experienced in designing SoC with crypto accelerators.
  - Only need IP blocks from PQSecure.
- **Do it for me**
  - Limited resources or expertise, need full support.
  - Require design, integration, and verification assistance.
- **Give me a starting point**
  - Need a secure solution tailored to specific needs.
  - Require customizable IP as a base to avoid reinventing the wheel.

SCA Technologies

PQ Secure



Many Stages for PQC Migration

PQ Secure

Perform risk assessment

Explore PQC algorithms

Study algorithms for the users

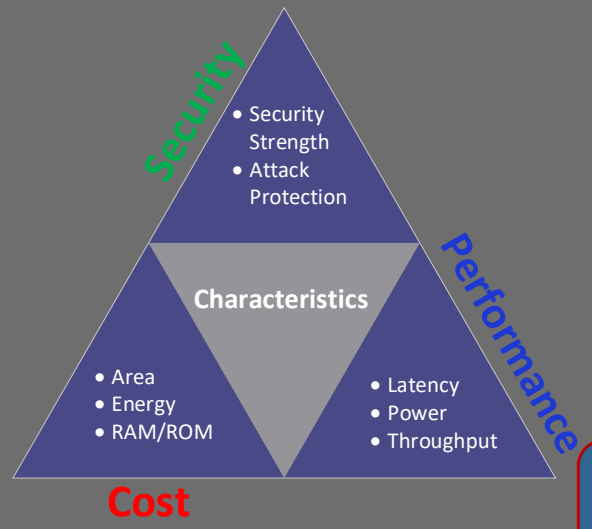
Study implementations on real HW

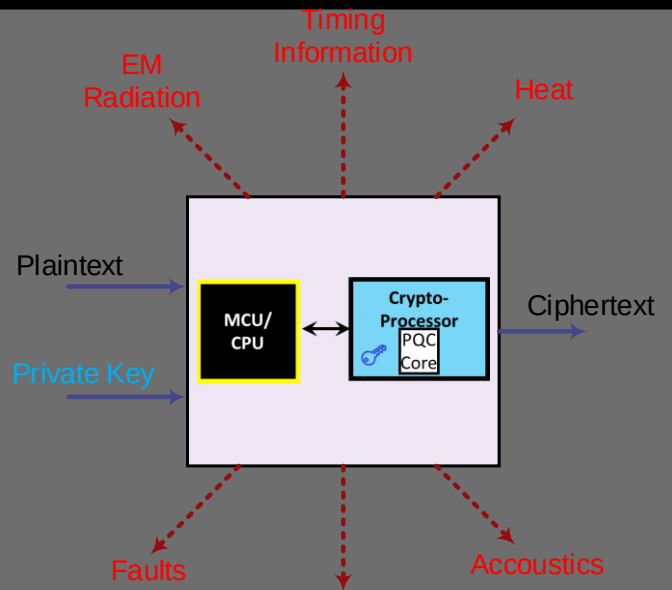
Study Physical security attacks SCA, FI, etc.

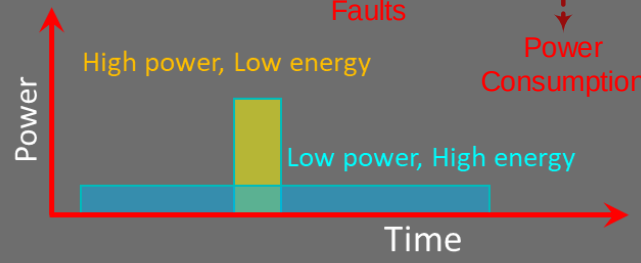
Focus on deployment for target performance

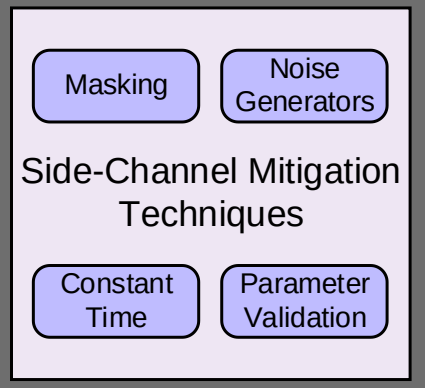
Integrate securely on read-world apps

Study reliable and high assurance









FIPS 140-3 “non-invasive SCA”

Implementing PQC primitives is more complex than implementing ECC/RSA, and the community has many years fewer experience with what could go wrong.

Formal Verification

PQ Secure

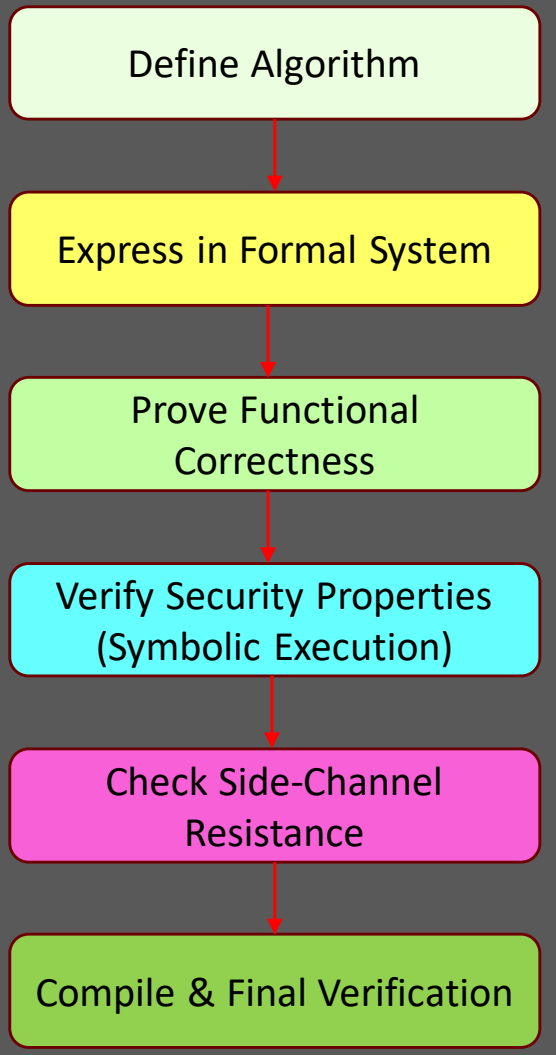
- Consider the field multiplication over $\mathbb{F}p$ with p = 2^255 − 19.
  - Number of inputs: 2^255 × 2^255
  - How many of them can be tested?
  - What about those inputs which are never tested?

- Formal verification aims to prove the absence of bugs through logical or mathematical reasoning.
  - Mathematical proof of correctness *for all possible inputs.*
  - Complements testing by providing exhaustive confidence.

What is Formal Verification?

PQ Secure

- Mathematical proof-based approach
- Ensures cryptographic implementations behave as intended
- Proves correctness across all possible inputs
- Why is Formal Verification Critical for PQC?
  - PQC is new and complex – traditional testing is insufficient
  - Verification prevents future vulnerabilities
  - Meets compliance standards (FIPS 140-3)



Challenges of Formal Verification in PQC

PQ Secure

1. Algorithmic Complexity

- Lattice-based schemes like Kyber or Dilithium use complex structures (e.g., NTT, rejection sampling).
- Verifying math-heavy operations (polynomial arithmetic, mod reductions) is non-trivial.

2. Low-Level Optimizations

- PQC implementations are heavily optimized for performance (e.g., AVX2, Cortex-M).
- Hand-optimized assembly is hard to formally verify due to lack of structure and documentation.

3. Correctness vs. Security

- Formal tools often verify *functional correctness*, not *cryptographic security*.
- Need for tools that can bridge functional correctness (e.g., "does NTT work") with security properties (e.g., IND-CCA2).

4. Tooling Gaps (Many formal tools are not yet adapted to post-quantum constructs)

- Most tools assume classical, simple arithmetic and logic.
- E.g., no native support for ring-based lattice arithmetic in most hardware formal tools.

5. Compositional Reasoning

- PQC schemes have multi-stage operations: keygen → encode → sample → transform.
- Verifying each step in isolation is hard without modular or compositional frameworks.

6. Side-Channel Resistance

- Formally verifying constant-time execution or masking countermeasures is still evolving.
- Side-channel models are complex and not fully integrated into most formal workflows.

Why Formal Verification Tools Needed for PQC?

PQ Secure

- Ensuring correctness in PQC algorithms requires rigorous verification *beyond* traditional testing
- PQC algorithms too complex for manual proofs → automation through formal tools **necessary**
- Verifies correctness, memory safety, and protocol security *before* deployment
- Key Areas of PQC Verification:
  - Mathematical Proof Verification → Theorem Proving
  - Protocol-level Security Checking → Model Checking
  - Software Execution Analysis → Symbolic Execution
  - Ensuring Compiler Security → PCC

What Tools Are Available for PQC?

PQ Secure

Category	Tools Used	Key Focus
Theorem Proving	Coq, Isabelle/HOL, EasyCrypt	Proving mathematical correctness of cryptographic algorithms
Model Checking	Tamarin, ProVerif	Verifying protocol security (e.g., PQC key exchange)
Symbolic Execution	SAW, KLEE, Angr	Checking functional correctness and memory safety
Compiler & Low-level Execution	Jasmin, CompCert, LLVM Verified Backend	Ensuring compiled cryptographic implementations remain secure and correct

Formosa Crypto: Advancing High-Assurance Cryptography

PQ Secure

- Collaborative research initiative focused on formally verifying cryptographic implementations
- Goal: Ensure formal verification, security, and efficiency in PQC
- Provides users with tools to advance formal verification
- Key Projects:
  - EasyCrypt → Theorem-proving framework
  - Jasmin → Formally verified programming language and compiler
  - Libjade → Cryptographic library, including PQC implementations

FORMOSA CRYPTO

EasyCrypt

PQ Secure

- Formal verification framework
- Uses game-based security proofs to model adversaries
- Employs interactive theorem proving to verify correctness
- Ensures PQC algorithms (e.g., Kyber) are mathematically secure
- Security Proofs in EasyCrypt (Kyber Example):
  - Goal: Show adversary cannot distinguish between two encapsulated secrets (IND-CPA)
  - Method: Define a security experiment where adversary tries to to guess which shared secret was encapsulated

Formal Security Proof in EasyCrypt

PQ Secure

- Define Kyber's KEM operations in EasyCrypt

- The adversary chooses two shared messages: *m0, m1*
- A random one (*b*) is encapsulated and the ciphertext is given to the adversary
- The attacker can query the decapsulation oracle (but not on the challenge ciphertext)
- If the adversary guesses *b* correctly with probability >50%
  - → *Kyber is insecure!*
- Security is proven via game reduction
- EasyCrypt provides an interactive theorem prover that checks every step of the proof formally

Formal Verification: Programming Languages

PQ Secure

- C has been standard language for writing crypto implementations
- Reminder: classical crypto standards should NOT apply to PQC
- So, should we still use C for PQC algorithms?
- C is not ideal for PQC:
  - Lack of built-in memory safety
  - Unpredictable optimizations by compilers
  - NO built-in formal verification

"Security engineers have been fighting with C compilers for years"

--Simon, Chisnall, Anderson, 2018

"We argue that we must stop fighting the compiler, and instead make it our ally"

C/C++ versus Rust for Secure cryptography

PQ Secure

Features	C/C++	Rust
Memory Safety	✗	✓
Clear, Defined Semantics	✗	✓
Mandatory Initialization	✗	✓
Built-in Runtime Checks	✗	✓
Secret vs Public Data Separation	✗	✓
Microarchitectural Attack Protection	✗	✓
Secure Data Erasure (Zeroization)	✗	✓
Strong Type Safety	✗	✓
Concurrency Safety	✗	✓
High Performance	✓	✓

C-to-Rust Cryptography

PQ Secure

C source code

```c
#define KYBER_INDCPA_MSGBYTES 123
#define KYBER_N 256
#define KYBER_Q 3329

#include <stdint.h>

void poly_tomsg(uint8_t msg[KYBER_INDCPA_MSGBYTES], const poly *a)
{
  unsigned int i,j;
  uint32_t t;

  for(i=0;i<KYBER_N/8;i++) {
    msg[i] = 0;
    for(j=0;j<8;j++) {
      t  = a->coeffs[8*i+j];
       t += ((int16_t)t >> 15) & KYBER_Q;
       t  = (((t << 1) + KYBER_Q/2)/KYBER_Q) & 1;
      msg[i] |= t << j;
    }
  }
}
```

c2rust.com

Generated Rust source code

```rust
#[no_mangle]
pub unsafe extern "C" fn poly_tomsg(mut msg: *mut uint8_t, mut a: *const libc::c
    let mut i: libc::c_uint = 0;
    let mut j: libc::c_uint = 0;
    let mut t: uint32_t = 0;
    i = 0 as libc::c_int as libc::c_uint;
    while i < (256 as libc::c_int / 8 as libc::c_int) as libc::c_uint {
        *msg.offset(i as isize) = 0 as libc::c_int as uint8_t;
        j = 0 as libc::c_int as libc::c_uint;
        while j < 8 as libc::c_int as libc::c_uint {
            t = (t as libc::c_uint)
                .wrapping_add(
                    (t as int16_t as libc::c_int >> 15 as libc::c_int
                        & 3329 as libc::c_int) as libc::c_uint,
                ) as uint32_t as uint32_t;
            t = (t << 1 as libc::c_int)
                .wrapping_add((3329 as libc::c_int / 2 as libc::c_int) as libc::
                .wrapping_div(3329 as libc::c_int as libc::c_uint)
                & 1 as libc::c_int as libc::c_uint;
            let ref mut fresh0 = *msg.offset(i as isize);
            *fresh0 = (*fresh0 as libc::c_uint | t << j) as uint8_t;
            j = j.wrapping_add(1);
        }
        i = i.wrapping_add(1);
```

- Unsafe C converts to unsafe Rust
- Translation uses Raw pointer instead of Rust native type
- Non idiomatic Rust, relies on C type

=> Less safe, less readable, and more error-prone than idiomatic Rust code written from scratch.

Pure Rust implementation

PQ Secure

```rust
fn poly_tomsg(msg: &mut [u8; KYBER_INDCPA_MSGBYTES], a: &Poly) {
    for (msg_byte, coeffs) in msg.iter_mut().zip(a.coeffs.chunks(8)) {
        *msg_byte = coeffs.iter().enumerate().fold(0, |acc, (j, &coeff)| {
            let t =
                ((((coeff as u32) << 1)
                .wrapping_add(1665)
                .wrapping_mul(80635) >> 28) & 1)
                as u8;
            acc | (t << j)
        });
    }
}
```

Idiomatic Rust implementation:

- Bound checks, explicitly handle arithmetic overflow
- Rust native type safety: Memory safety, thread safety
- No NULL pointer, no UNDEFINED behavior
- Side-channel resistance

Final Notes

PQ Secure

- PQC's complexity requires rigorous security guarantees
- Formal Verification ≠ academic concept
- …it is essential to securing next-gen crypto implementations
- Tools like Jasmin and EasyCrypt are advancing verification in PQC
- Continued advancements must be made to ensure correctness, security, and efficiency in PQC
- AI-driven automation and increased industry adoption promises a bright future for PQC and formal verification!

Questions?

PQ Secure

Reza Azarderakhsh
CEO PQSecure
razarder@pqsecurity.com