Taking the (quantum) leap with Go

Mathilde Raynal[1] & Yolan Romailler[2]

[1] Kudelski Security
[2] SICPA (work done at Kudelski Security)

About us

Mathilde, research intern @ Kudelski & EPFL

- My work is 1 cup of crypto, 1 tbsp of privacy and a pinch of machine-learning
- when not geeking, I can be found at a bouldering gym
- linkedIn: mathilde.raynal

Yolan, cryptographer @ SICPA, prev. @ Kudelski

- Terrible cook: plain crypto sprinkled with some security engineering
- loves playing CTFs and (with) Go
- Twitter: @anomalroil

At Kudelski Security we:

- are actively involved in research;
- provide quantum-resistant security services;
- run a crypto blog with regular posts.

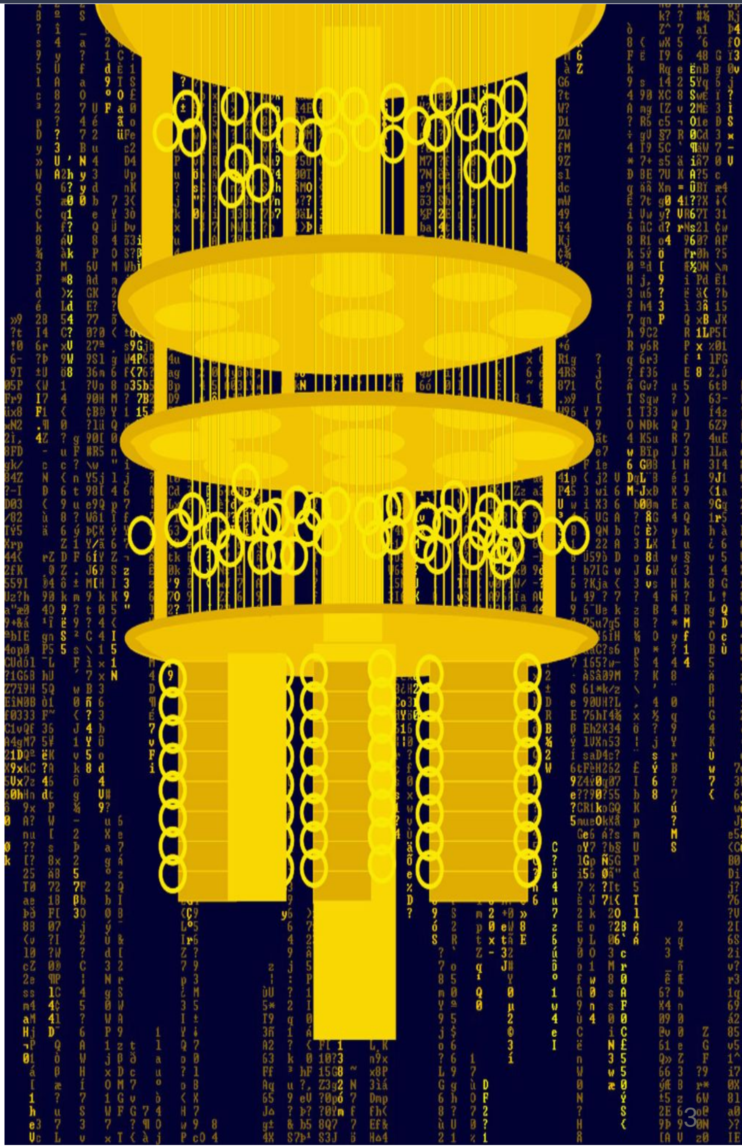
Introduction

The quantum computer threat is looming at the horizon for our cryptographic algorithms.

Quantum computers threaten the security of **public-key** schemes that we currently use.

They will **not** protect sensible information anymore.

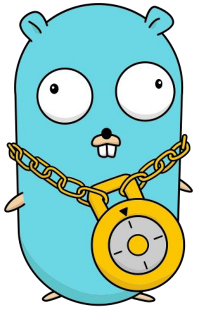
Symmetric-key cryptography and hash function are impacted, but not *broken*.



Source: MIT Technology Review

Public-key crypto... what for?

- Connecting to remote machines without relying on ~~weak~~ mere passwords with SSH
- TLS communication (basically the whole web)
- Sending PGP emails?
- Using modern VPNs such as **Wireguard**.
- Everywhere in blockchains.
- Encrypting passwords with tools such as **gopass**.
- ...



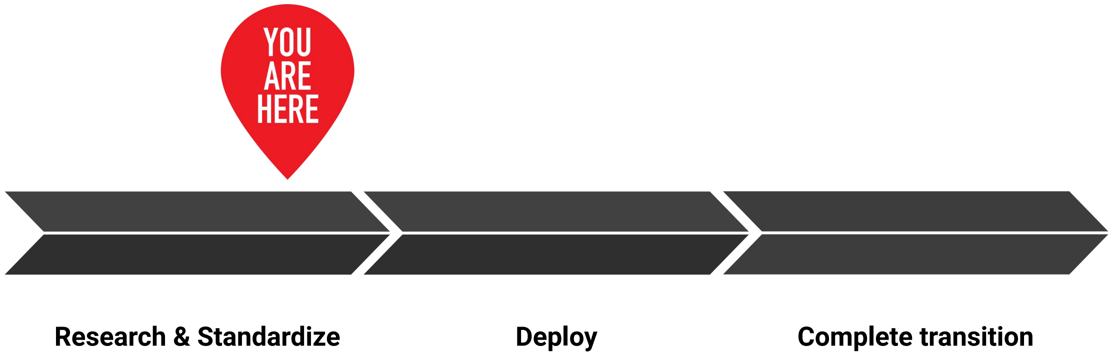
Post-quantum cryptography to the rescue

Cryptography that is **resistant** to attacks ran on both classical and quantum computers:

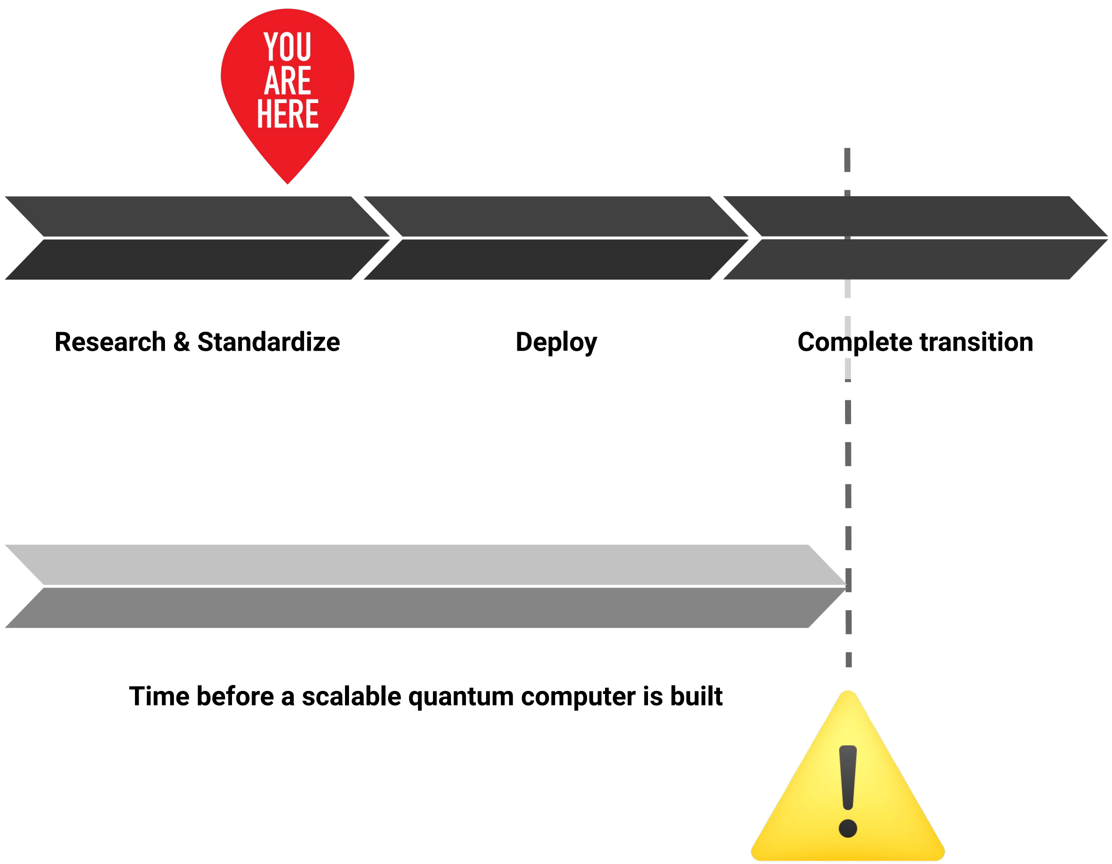
- Lattice-based
- Isogeny-based
- Code-based
- Hash-based
- Multivariate
- Symmetric.

Different trade-offs are available (runtime, bandwidth, …) so it is important to extract the requirements of your application to choose the best post-quantum tool.

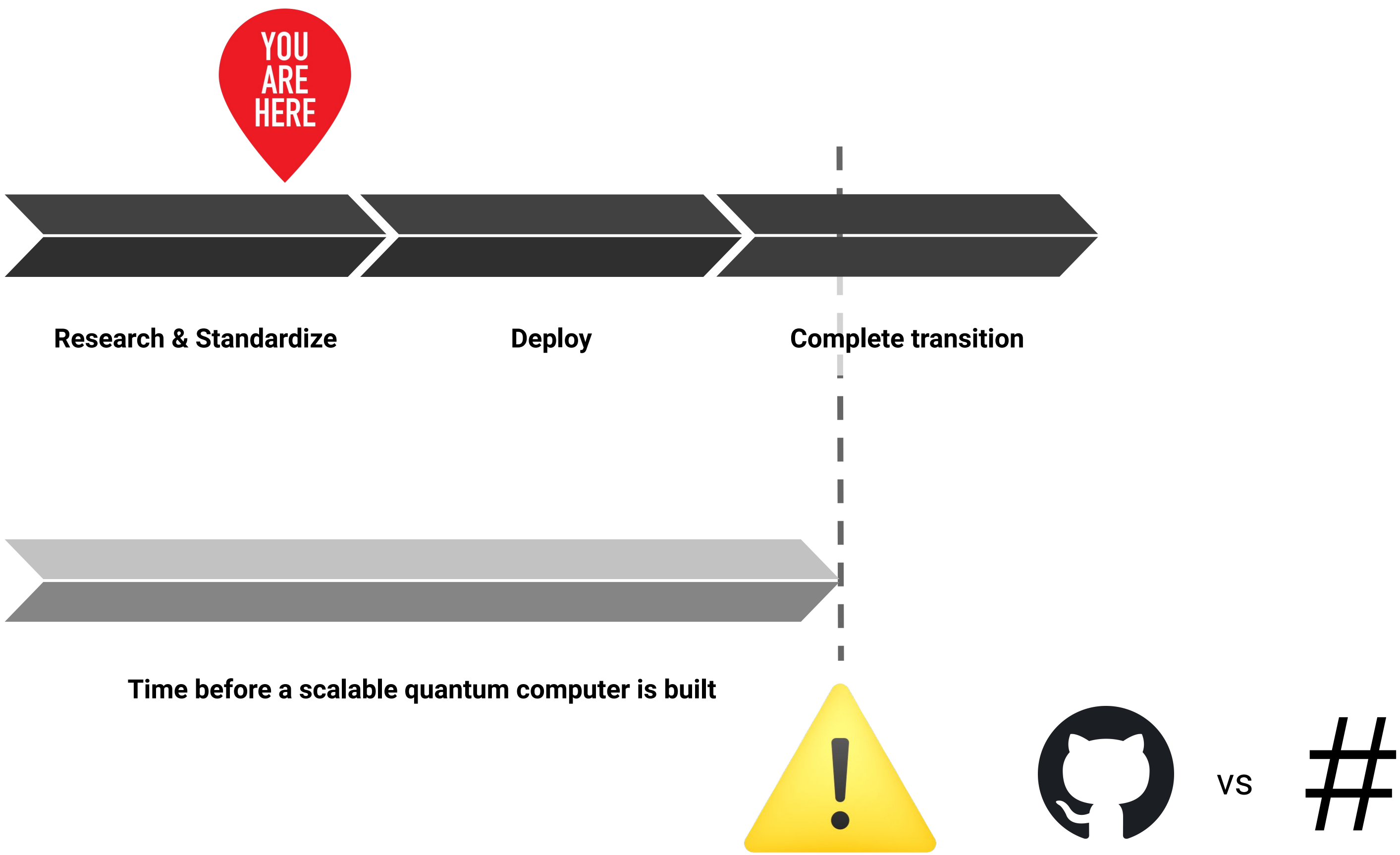
Why should you *already* want post-quantum cryptography in your project?



Why should you *already* want post-quantum cryptography in your project?



Why should you *already* want post-quantum cryptography in your project?



Why should you *already* want post-quantum cryptography in your project?

«Progress has been swift. In a few short years we now have over 20 of the world's most powerful quantum computers, accessible for free on the IBM Cloud.» - *IBM Quantum Experience*

Fully functioning quantum computers will arrive sooner than many have anticipated:

- we should not postpone as *crypto-agility* is a true challenge.

Increasing the key sizes of classical schemes is not a viable option regarding security or performance:

- a quantum resistant RSA protocol would require **1 TB** keys.

Crypto Refresher

- ★ Hybrid encryption
  - Public-key cryptography is **slow**, symmetric key crypto is usually **fast**.
  - People want *fast internet*.
  - We need to use as little public-key crypto as possible → hybrid encryption

  Uses a public-key algorithm to establish a shared secret, uses shared secret to do symmetric encryption

Most secure protocols (TLS, SSH, IPsec, ...) nowadays are relying on hybrid encryption: a public-key algorithm for **key-exchange** or **authentication,** and a symmetric key algorithm (AES) for actual data encryption.

Crypto Refresher

DSA

A Digital Signature Algorithm is used to produce a signature on a message using a secret signing key. The signature can be verified by anyone holding the associated public key.

★ Unforgeability → can be used as **authentication** mechanism

KEM

The goal of a Key Encapsulation Mechanism protocol is to securely exchange symmetric key material over an insecure channel using public key cryptography.

★ Confidentiality → can be used to **exchange key material**

Our library

The CRYSTALS suite is made of two algorithms:

- Dilithium, a DSA, and
- Kyber, a KEM.

Both are very promising alternatives for post-quantum cryptography, and are finalists in the post-quantum cryptography standardization competition organized by NIST.

They are **lattice-based**, and stand out for their simplicity, tight security and overall versatility.

They have a **great performance**, and have been shown to excel some of the widespread classical solutions. Their main drawback is their **relatively large outputs** size, which might impact the performance, but is never considered a bottleneck.

Our library

We ported the reference implementation of the CRYSTALS algorithms from C to Go.
It is open-source and available at: https://github.com/kudelskisecurity/crystals-go (QR code).

At Kudelski Security, our mission is to emphasize ***practical*** security, so we put a lot of efforts into integrating as many security features as possible.

Don't hesitate to open issues on our Github!

API

In two steps: first choose a security level, then invoke the core functions.

API

Dilithium:

Type

NewDilithium2() → d
NewDilithium3() → d
NewDilithium5() → d

Core

(d *Dilithium) KeyGen() → pk, sk
(d *Dilithium) Sign(sk, msg) → sig
(d *Dilithium) Verify(pk, sig, msg) → boolean

API

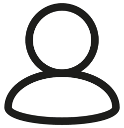
Dilithium:

Type

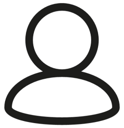
NewDilithium2() → d
NewDilithium3() → d
NewDilithium5() → d

Core

(d *Dilithium) KeyGen() → pk, sk
(d *Dilithium) Sign(sk, msg) → sig
(d *Dilithium) Verify(pk, sig, msg) → boolean



```
d := NewDilithium2()
```



```
d := NewDilithium2()
```

API

Dilithium:

Type

NewDilithium2() → d
NewDilithium3() → d
NewDilithium5() → d

Core

(d *Dilithium) KeyGen() → pk, sk
(d *Dilithium) Sign(sk, msg) → sig
(d *Dilithium) Verify(pk, sig, msg) → boolean

```
d := NewDilithium2()
pk, sk := d.KeyGen(seed)
```

pk →

```
d := NewDilithium2()
```

API

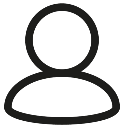
Dilithium:

Type

NewDilithium2() → d
NewDilithium3() → d
NewDilithium5() → d

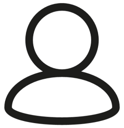
Core

(d *Dilithium) KeyGen() → pk, sk
(d *Dilithium) Sign(sk, msg) → sig
(d *Dilithium) Verify(pk, sig, msg) → boolean



```
d := NewDilithium2()
pk, sk := d.KeyGen(seed)

msg := []byte("A message")
sig := d.Sign(sk, msg)
```

pk



```
d := NewDilithium2()
```

API

Dilithium:

Type

NewDilithium2() → d
NewDilithium3() → d
NewDilithium5() → d

Core

(d *Dilithium) KeyGen() → pk, sk
(d *Dilithium) Sign(sk, msg) → sig
(d *Dilithium) Verify(pk, sig, msg) → boolean

```
d := NewDilithium2()
pk, sk := d.KeyGen(seed)

msg := []byte("A message")
sig := d.Sign(sk, msg)
```

pk →

sig, msg →

```
d := NewDilithium2()


ok := d.Verify(pk, sig, msg)
```

API

Kyber:

Type

NewKyber512() → k
NewKyber768() → k
NewKyber1024() → k

Core

(k *Kyber) KeyGen() → pk, sk
(k *Kyber) Encaps(pk, coins) → c, ss
(k *Kyber) Decaps(sk, c) → ss

API

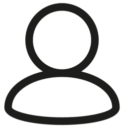
Type

Kyber:

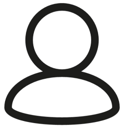
NewKyber512() → k
NewKyber768() → k
NewKyber1024() → k

Core

(k *Kyber) KeyGen() → pk, sk
(k *Kyber) Encaps(pk, coins) → c, ss
(k *Kyber) Decaps(sk, c) → ss



```
k := NewKyber512()
```



```
k := NewKyber512()
```

API

Kyber:

Type

NewKyber512() → k
NewKyber768() → k
NewKyber1024() → k

Core

(k *Kyber) KeyGen() → pk, sk
(k *Kyber) Encaps(pk, coins) → c, ss
(k *Kyber) Decaps(sk, c) → ss

```
k := NewKyber512()
pk, sk := k.KeyGen(seed)

ss := k.Decaps(sk, c)
```

pk →

← c

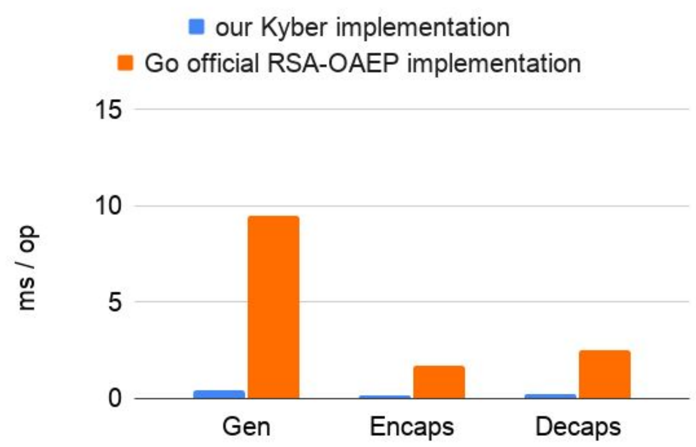
```
k := NewKyber512()

c, ss := k.Encaps(pk, coins)
```

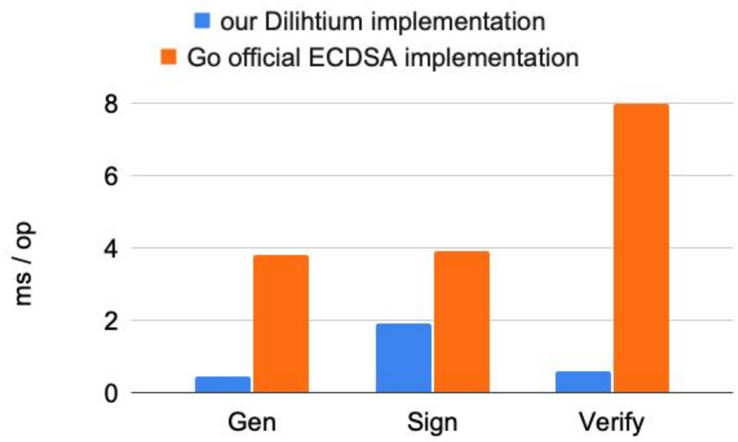
Performance Overview

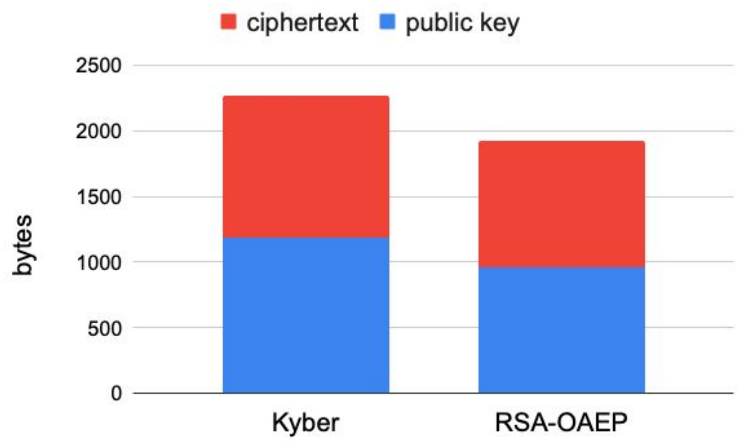
Security: We provide a library that is both theoretically and practically secure. We integrated countermeasures for many published implementation attacks (side-channel)

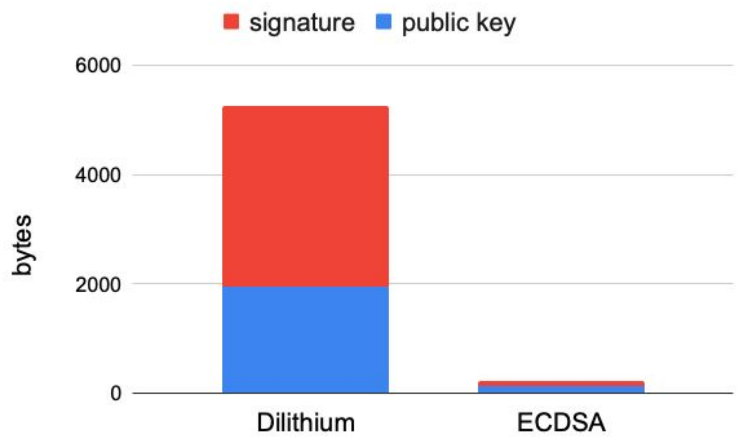
	Runtime (ms)			Size (B)	
Dilithium	*KeyGen*	*Sign*	*Verify*	*Public Key*	*Signature*
	0.4	1.8	0.5	1 952	3 293
Kyber	*KeyGen*	*Encaps*	*Decaps*	*Public Key*	*Ciphertext*
	0.4	0.2	0.3	1 184	1 088

crystals-go vs go/x/crypto









PQ-WireGuard

WIREGUARD
FAST, MODERN, SECURE VPN TUNNEL

&

crystals-go

PQ-WireGuard

WIREGUARD
FAST, MODERN, SECURE VPN TUNNEL

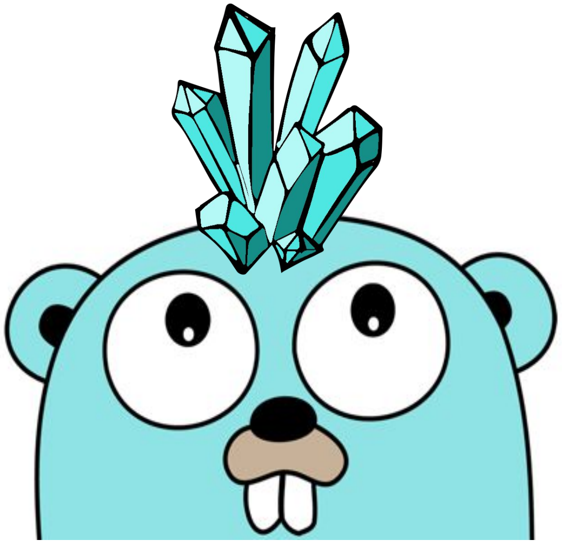
&

crystals-go

- \+ 2 IP packets
- \+ 0.2 ms

Attend our talk at the NIST 3rd PQC Standardization Conference for more details !

Conclusion



Our experimental results should be used as motivation to start the transition towards post-quantum alternatives !

Our library is **fast**, **secure**, and **easy** to use and integrate in your project, why wait?

Checkout our other material on quantum security: [Point of View Paper – Quantum Security]
Our research blog about the library: https://research.kudelskisecurity.com/?p=15394
About the integration in WireGuard: [Third PQC Standardization Conference | CSRC]

References

Léo Ducas et al., *CRYSTALS-Dilithium: A Lattice-Based Digital Signature Scheme*, 2017
Joppe Bos et al., *CRYSTALS-Kyber: A CCA-Secure Module-Lattice-Based KEM,* 2017

Crystal image by Tatyana from Noun Project

Q & A