Hybrid Post-quantum Signatures in Hardware Security Keys

Diana Ghinea[1,2], Fabian Kaczmarczyck[2(✉)], Jennifer Pullman[2], Julien Cretin[2], Stefan Kölbl[2], Rafael Misoczki[2], Jean-Michel Picod[2], Luca Invernizzi[2], and Elie Bursztein[2]

[1] ETH, Zürich, Switzerland
ghinead@ethz.ch
[2] Google, Zürich, Switzerland
{dianamin,kaczmarczyck,jpullman,cretin,kste,
jmichel,invernizzi,elieb}@google.com

Abstract. Recent advances in quantum computing are increasingly jeopardizing the security of cryptosystems currently in widespread use, such as RSA or elliptic-curve signatures. To address this threat, researchers and standardization institutes have accelerated the transition to quantum-resistant cryptosystems, collectively known as Post-Quantum Cryptography (PQC). These PQC schemes present new challenges due to their larger memory and computational footprints and their higher chance of latent vulnerabilities.

In this work, we address these challenges by introducing a scheme to upgrade the digital signatures used by security keys to PQC. We introduce a hybrid digital signature scheme based on two building blocks: a classically-secure scheme, ECDSA, and a post-quantum secure one, Dilithium. Our hybrid scheme maintains the guarantees of each underlying building block even if the other one is broken, thus being resistant to classical and quantum attacks. We experimentally show that our hybrid signature scheme can successfully execute on current security keys, even though secure PQC schemes are known to require substantial resources.

We publish an open-source implementation of our scheme at https://github.com/google/OpenSK/releases/tag/hybrid-pqc so that other researchers can reproduce our results on a nRF52840 development kit.

Keywords: PQC · FIDO · Dilithium · Embedded

1 Introduction

Recent advances in quantum computing are increasingly jeopardizing the security of cryptosystems currently in widespread use, such as RSA [34] and DSA [21]. For example, even the comparatively-newer ECDSA, based on elliptic curve cryptography [20], is vulnerable to quantum attacks (i.e., attacks that leverage quantum computers).

To address this threat, researchers and standardization institutes have accelerated the transition to quantum-attack-resistant cryptosystems, collectively

J. Zhou et al. (Eds.): ACNS 2023 Workshops, LNCS 13907, pp. 480–499, 2023.
https://doi.org/10.1007/978-3-031-41181-6_26

known as Post-Quantum Cryptography (PQC). These PQC schemes rely on a new set of underlying hard problems, that researchers believe to be impervious to quantum attacks. However, these schemes present new challenges due to their substantial memory and computational footprints and their higher chance of latent vulnerabilities due to the schemes' novelty, such as the one recently discovered by Castryck and Decru [9] against SIKE. To mitigate the potential damage, researchers are pursuing hybrid signature schemes [6], which maintain the classical scheme's security against classical attackers.

One class of protocols that needs upgrading to PQC is security-key-based authentication protocols, such as FIDO's CTAP and WebAuthn [15,22]. Through these protocols, a user can prove their identity with a hardware token (commonly called a *security key*), either as a first-factor or second-factor authentication.

In this work, we address this challenge by introducing a PQC digital signature scheme for hardware security keys, focusing on both the theoretical and practical aspects of this scheme. Specifically, we introduce a hybrid scheme based on two building blocks: a classically-secure scheme, ECDSA, and a post-quantum secure one, Dilithium. We picked Dilithium [13] as it is one of the schemes recently selected by United States National Institute of Science and Technology (NIST) [28] as the PQC standard for digital signature schemes and because of the fast speed of its signing—the most frequent operation in our use case.

On the practical aspect, we show that our hybrid signature scheme can be executed by current security keys, even though secure PQC schemes require substantial resources. Specifically, we implement our hybrid scheme in OpenSK [31], an open-source firmware for security keys written in Rust. We provide our implementation as open-source software with an Apache2 license at https://github.com/google/OpenSK/releases/tag/hybrid-pqc.

Our contributions are as follows:

- We prove the strong unforgeability of a previously proposed hybrid signature scheme [6] in the context of security key authentication. Our hybrid scheme maintains the guarantees of each underlying scheme even if the other one is broken, thus being resistant to classical and quantum attacks.
- We release an implementation of our hybrid scheme with ECDSA and Dilithium as underlying components. We have implemented this scheme in Rust on top of the security-key firmware OpenSK. To allow deployment on diverse hardware, we do not take advantage of any hardware-specific acceleration and we ensure that the memory footprint of our hybrid scheme fits in 64 kB of RAM. This requirement leads us to reduce Dilithium's memory footprint.

1.1 Related Work

Hybrid Cryptosystems. For the transition period to prevalent quantum computers, hybrid cryptosystems provide security against future quantum attackers, while mitigating potential design or implementation bugs in the face of classical attackers. Hybrid solutions exist for e.g., authenticated key exchange [2,12], public-key encryption [25], and digital signatures [6].

The hybrid scheme we are implementing in this work follows one of the designs presented in [6]. The difference is that we investigate the feasibility of achieving post-quantum security under the constraints of embedded hardware. Furthermore, we show that this design actually achieves stronger security definitions than shown in [6] (i.e., strong existential unforgeability versus existential unforgeability). In addition, we show a natural extension to a larger class of hybrid signature schemes of the non-separability property introduced by Bindel et al., and we show how it can be applied in the context of passwordless authentication.

PQC for FIDO. In a concurrent work, Bindel et al. [5] have analyzed whether FIDO can provide PQC guarantees from a theoretical point of view and formally verify parts of the FIDO protocols. They argue how PQC can be integrated to maintain theoretical guarantees. Our work complements theirs in the following aspects:

- They focus on the PIN protocol, and mostly omit registration and authentication. We propose a signing scheme for this use case.
- They formally prove parts of the FIDO protocols. We prove the security guarantees of a hybrid signing primitive.
- They propose changes to the FIDO protocols, leaving the cryptographic primitives as an open choice. We implement a PQC primitive that works on embedded hardware, and open-source fully-working firmware.

Dilithium vs Other PQC. Dilithium [13] and Falcon [24] both won NIST's post-quantum signature algorithm standardization challenge. We compare the two schemes in Fig. 1. As will be discussed in Sect. 5, for security keys, we are mainly interested in the signing speed and the private key size, whereas verification is not performed on embedded hardware. The private key size affects how many credentials can be stored on the security key.

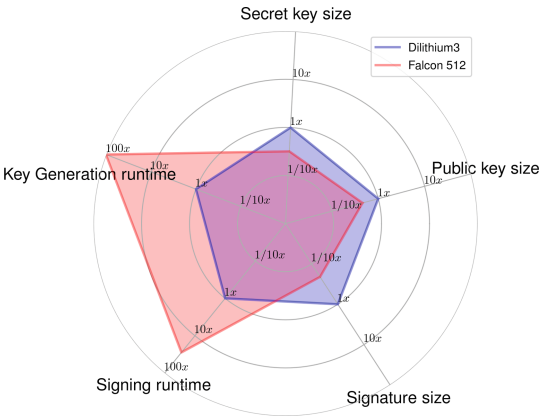


Fig. 1. Relative performance of Falcon 512 compared to the reference implementation of Dilithium3 as the baseline for the security key use case, as reported by Raavi et al. [33] in Figure 6.

We optimize Dilithium in Sect. 5 to get closer to Falcon in key sizes. Our implementation maintains competitive signing speeds, while having favorable properties overall. We use some of the optimization techniques in Bos et al. [7], and implement them in Rust for added memory safety.

- *Private key size:* We store the 256-bit secret that we use to regenerate the private key on the fly. Given the amount of entropy we need, this is near-optimal in terms of required storage.
- *Public key and signature size:* Public keys and signatures have to be small enough for transmission over USB and NFC. Falcon has 2× smaller public keys and 3× smaller signatures than Dilithium for comparable security levels. Dilithium is still compatible with the CTAP protocol's constraints.
- *Key generation speed:* Dilithium is 100x faster for key generation [18] than Falcon. That is why we can regenerate private keys on the fly from the stored random seed.
- *Memory:* Falcon has smaller requirements, but Dilithium can be optimized to fit embedded devices.

Outside of lattice-based signatures, there are stateless hash-based signature schemes like SPHINCS+ [3] which has been selected as a standard by NIST. However, their much larger signature size is infeasible for our use case, and the performance cost of signing compared to lattice schemes is significantly worse.

Improvements to Dilithium's Performance. Multiple works have focused on improving Dilithium's reference implementation [32] in terms of speed and memory footprint. Another direction is vulnerability against side-channel attacks, Migliore et al. [27] analyze Dilithium on an ARM Cortex-M3 microcontroller, and remove unexpected leakages through masking. Greconici et al. [17] obtain a constant-time Dilithium implementation on the Cortex-M3, which is necessary for limiting potential side-channel attacks. In addition, they present different strategies for the signing procedure that allow trading between the stack and flash memory usage and speed, which can be applied for Cortex-M3 and Cortex-M4. Abdulrahman et al. [1] focused on further improvements of Dilithium's speed on the ARM Cortex-M4.

Concurrently to our work, Bos et al. [7] have focused on reducing Dilithium's memory footprint to less than 9 kB. While our implementation requires less than 22 kB, one can choose to compile different optimizations according to the use case. Minimizing for binary size shrinks it to 9.3 kB compared to 9.8 kB for Bos et al. Letting the compiler optimize for speed makes our code 6% to 18% faster than them instead. Other advantages are: it ensures memory safety as it is written in Rust, it is open-source and ready to use for security keys, even in the hybrid setting.

2 Background

In this section, we introduce the relevant cryptography, describe our use case of security keys and explain our hardware and firmware stack for embedded development.

2.1 Digital Signatures

We first recall the definition of a digital signature scheme. For our scope, we only consider signature schemes deployed on classical computers.

Definition 1 *(Digital signature scheme).* *A digital signature scheme Σ is a triple of polynomial time algorithms $(\Sigma.\texttt{KeyGen}, \Sigma.\texttt{Sign}, \Sigma.\texttt{Verify})$ such that:*

- $\Sigma.\texttt{KeyGen}(1^\kappa)$ *is a probabilistic algorithm that takes the security parameter 1^κ as input and outputs a public verification key* pk *and a secret signing key* sk.
- $\Sigma.\texttt{Sign}(m, \mathsf{sk})$ *is a probabilistic algorithm that takes a message m and a secret key* sk *as inputs and outputs a signature σ.*
- $\Sigma.\texttt{Verify}(m, \sigma, \mathsf{pk})$ *is a deterministic algorithm that takes a message m, a signature σ and a public key* pk *as inputs. It outputs* true *or* false*, where* true *means that σ is accepted as a signature for the message m and public key* pk*, and* false *means that the signature is not accepted.*

Digital signature schemes must achieve two properties: *correctness* and *security*. Correctness requires that for every key pair $(\mathsf{sk}, \mathsf{pk}) \leftarrow\!\!\$\ \Sigma.\texttt{KeyGen}(1^\kappa)$, every possible message m, and any possible $\sigma \leftarrow\!\!\$\ \Sigma.\texttt{Sign}(m, \mathsf{sk})$, it holds that $\Sigma.\texttt{Verify}(m, \sigma, \mathsf{pk}) = \mathsf{true}$. In terms of security, there are multiple definitions, and we present the ones relevant for our context below.

The security of digital signatures is often defined through a security game where an adversary tries to forge a valid signature while interacting with a challenger who holds the secret key. For the scope of our paper, we only consider classical challengers (i.e., the signing oracle runs on a classical computer). We will use the notation C-adversary to refer to a classical adversary, and Q-adversary to refer to a quantum adversary with classical access to the signing oracle.

Security guarantees for digital signatures are often defined through the goals and constraints of the adversary. We only work with two security definitions, presented below: EUF-CMA (Existential Unforgeability under Chosen Message Attacks) and SUF-CMA (Strong Unforgeability under Chosen Message Attacks).

Definition 2 *(EUF-CMA security).* *We consider the EUF-CMA security game for a signature scheme Σ, where the adversary $\mathcal{A}$ interacts with a challenger $\mathcal{C}$ as follows:*

1. *The challenger $\mathcal{C}$ obtains a pair of keys from the key generation algorithm $(\mathsf{sk}, \mathsf{pk}) \leftarrow\!\!\$\ \Sigma.\texttt{KeyGen}(1^\kappa)$ and sends* pk *to $\mathcal{A}$.*
2. *$\mathcal{A}$ may adaptively send a polynomial (in κ, the security parameter) number of queries m_i to the challenger $\mathcal{C}$. For each such query, $\mathcal{C}$ obtains $\sigma_i = \Sigma.\texttt{Sign}(m, \mathsf{sk})$ and sends σ_i to the adversary. Note that $\mathcal{A}$ may send query m_{i+1} after receiving σ_i.*
3. *$\mathcal{A}$ may send a message-signature pair (m^*, σ^*). $\mathcal{A}$ wins the EUF-CMA security game if $m^* \notin \{m_i$ queried by the adversary$\}$ and $\Sigma.\texttt{Verify}(m^*, \sigma^*, \mathsf{pk})$ holds.*

We say that Σ is C-EUF-CMA secure if any classical $\mathcal{A}$ wins the EUF-CMA security game with negligible probability ($\mathsf{negl}(\kappa)$). Similarly, Σ is Q-EUF-CMA secure if any (possibly quantum) $\mathcal{A}$ that interacts with the signing oracle classically wins the EUF-CMA security game with negligible probability ($\mathsf{negl}(\kappa)$).

Definition 3 *(SUF-CMA security). Similar to Definition 2, but replace EUF with SUF and step 3 with:*

3. *$\mathcal{A}$ may send a message-signature pair (m^*, σ^*). $\mathcal{A}$ wins the SUF-CMA security game if $(m^*, \sigma^*) \notin \{(m_i, \sigma_i) \mid m_i \text{ queried}\}$ and $\Sigma.\mathtt{Verify}(m^*, \sigma^*, \mathsf{pk})$ holds.*

Achieving SUF-CMA security implies achieving EUF-CMA security, and achieving the quantum variant of a definition implies achieving the classical variant.

We recall the security guarantees of the concrete digital signature schemes that we use:

- ECDSA achieves C-EUF-CMA security in the random bijection model [14].
- Dilithium achieves Q-SUF-CMA security in the quantum random oracle model [19] (which implies all the weaker security definitions introduced above).

2.2 Post-quantum Cryptography

Dilithium is a signature scheme without known weaknesses to quantum computers. Different parameter sets of Dilithium are called modes and correspond to estimated security levels. The cryptographic strengths of Dilithium are shown in Table 1. The hardness is quantified with respect to the underlying mathematical problems Learning With Errors (LWE) and Short Integer Solution (SIS). The LWE and SIS problems are conjunctured to be hard to solve. NIST made an attempt to translate these hardness levels to classical cryptography [29]. While classical and quantum security levels are hard to directly compare, we add these estimates to the table as an approximation.

Table 1. Cryptographic strength of Dilithium modes, as of NIST standardization round 3 (see [10], Table 1). The classical equivalent refers to NIST's estimation.

Mode	LWE	SIS (for SUF-CMA)	Classical equivalent
Dilithium2	112	112 (110)	SHA256 collision
Dilithium3	165	169 (159)	AES-192 key search
Dilithium5	229	241 (230)	AES-256 key search

2.3 Security Keys

Security keys allow user authentication through digital signatures. They are often implemented on embedded hardware to protect secret key material from extraction.

FIDO. Fast IDentity Online (*FIDO*, see [15]) is a set of standards to allow online authentication through asymmetric cryptography. This exchange of messages involves two protocols: the Client to Authenticator Protocol (*CTAP*, see [11]), which enables the communication between the user's *Authenticator* and their *Client* (such as their browser, or their computer), and WebAuthn, which ensures the communication between the client and the server (*Relying Party*). Security keys act as authenticators and therefore implement CTAP.

CTAP. As part of a CTAP registration, the user generates a key pair, and sends the public key to the server. For a CTAP authentication, the user then proves possession of the private key (*Credential*) being stored on an authenticator. A credential can be stored in one of two ways: Either it is encrypted and sent to the relying party for storage, or it is stored locally in flash. We call these cases server-side key and resident key, respectively.

We describe the cryptographic commands in CTAP below (see Fig. 2). The CTAP protocol started with U2F [22], and since then evolved to its current version 2.1. The most important commands are Make Credential for registration and Get Assertion for authentication. Depending on usage of server-side or resident credentials, these commands use the following cryptographic operations:

R1) During registration, the security key generates a key pair.
R2) Registration returns the public key of the credential, and may return the encrypted private key (server-side key).
R3) Registration returns the public key of the credential, and may store the private key on flash (resident key).
A1) Authentication returns a signature over a response derived from the Relying Party's message.
A2) Authentication returns a signature, and may return an encrypted private key (server-side vs resident key).

3 Attacker Model

Security keys' main goal is to defend against remote attackers and phishing. Defense against local attackers with physical possession of the device are an explicit non-goal. Adversaries can attack the protocol on different levels: cryptographically, on CTAP level, or against the hardware device. In our cryptography analysis in Sect. 4, we consider different extended capabilities for attackers:

– Possession of a Cryptographically-Relevant Quantum Computer;
– Knowledge of a Dilithium weakness.

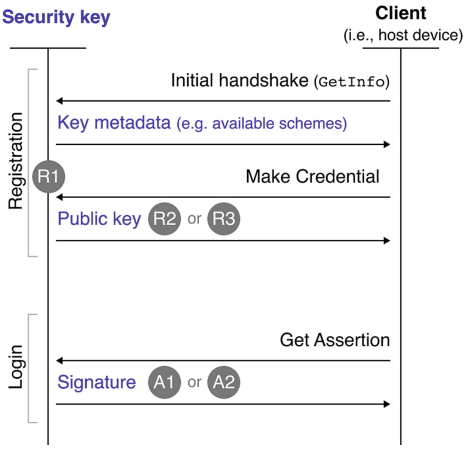


Fig. 2. Cryptographic operations in the CTAP protocol.

We acknowledge that FIDO's protocols mitigate downgrading the protocol already. They transmit the used algorithm over a channel that is considered secure in their attacker model.

Cryptographic Strength. We want our implementation to support all modes of Dilithium, to allow applications with strong security requirements. In particular, security keys are an important line of defense against account hijacking.

Non-goals. Local attacks against the hardware itself or faulty implementations are out of scope for this work. That includes local side-channel attacks. Indeed, Dilithium has been successfully attacked locally on e.g., the power side-channel [26]. We follow FIDO's security assumptions, listed in their Security Reference [16]. The two most important for our threat model are the following:

SA-3 Applications on the user device are able to establish secure channels that provide trustworthy server authentication, and confidentiality and integrity for messages (e.g., through TLS).
SA-4 The computing environment on the FIDO user device and the applications involved in a FIDO operation act as trustworthy agents of the user.

4 Hybrid Signatures

A hybrid signature scheme combines a classical signature algorithm with a post-quantum secure signature algorithm (in a construction commonly known as a *combiner*). Before discussing the design of our hybrid scheme, we explain why such an approach is relevant instead of simply replacing classically secure schemes with post-quantum secure schemes. We present the assumptions below:

1. Cryptographically-Relevant Quantum Computers (i.e., with enough qubits to break ECDSA) are not available yet.
2. Classical signature algorithms withstand attacks from classical computers.
3. The post-quantum secure signature algorithm might be breakable by classical computers due to design or implementation bugs.

The first two assumptions present today's reality. As soon as one of these two assumptions fails, post-quantum security becomes a requirement. On the other hand, post-quantum cryptography is still young, and attacks are still being discovered. One such example is a recent attack against Rainbow [4], one of the NIST standardization finalists. Our third assumption reflects this, and motivates the transition to post-quantum secure schemes through hybrid schemes.

We can now discuss the informal requirements a hybrid scheme H should satisfy:

1. If a quantum computer becomes available, and hence H's underlying classical scheme is broken, H should maintain the security of its underlying post-quantum scheme.
2. If a classical attack for H's underlying post-quantum secure scheme is discovered, H should maintain the security of its underlying classical scheme.

There are multiple natural options for designing a hybrid scheme that satisfies such guarantees. An example is obtaining a hybrid signature by concatenating a classical signature with a post-quantum secure signature. Although simple, this approach indeed maintains the existential unforgeability of the underlying schemes [6].

On the other hand, for our concrete instantiation, Dilithium is strongly unforgeable, while ECDSA is existentially but not strongly unforgeable [14]. Concatenation unfortunately would not maintain Dilithium's strong unforgeability: one could simply replace the ECDSA part of the hybrid signature with another valid ECDSA signature.

Intuitively, this issue can be solved by first signing a given message m with the X-EUF-CMA secure scheme, obtaining σ_1, and afterwards obtaining σ_2 by signing (m, σ_1) with the X-SUF-CMA secure scheme. The hybrid signature for the message m is $\sigma = (\sigma_1, \sigma_2)$. This approach is called *Strong Nesting* [6] and is the basis of our hybrid scheme.

Replacing ECDSA with Ed25519 is another possible fix, as the latter is strongly unforgeable [8]. However, as all security keys already implement ECDSA, a hybrid protocol that uses ECDSA benefits from code reuse.

We make use of the suggestion of Bindel et al. [6] of prepending a constant label to the message to be signed. As an alternative construction with the same properties, one could choose a random label during key generation and store it in both the secret and public key. We decided to use the constant label approach because it leads to smaller key size. This label essentially encodes an algorithm identifier bound to the scheme, and restricts an adversary from trivially deriving partial signatures for messages not having the chosen label as a prefix (if

the underlying schemes are secure). We exclude this label from the theoretical analysis and consider it part of the message for the rest of this chapter.

We now formally present our hybrid signature scheme. Given two signature schemes Σ_1 and Σ_2, we define the secret and public keys as pairs of their counterparts in the given underlying schemes. Below we present the pseudocode of the `KeyGen`() function.

$\mathcal{H}(\Sigma_1, \Sigma_2).\texttt{KeyGen}(1^\kappa)$

$\mathsf{sk}_1, \mathsf{pk}_1 \leftarrow\!\!\$\ \Sigma_1.\texttt{KeyGen}(1^\kappa)$

$\mathsf{sk}_2, \mathsf{pk}_2 \leftarrow\!\!\$\ \Sigma_2.\texttt{KeyGen}(1^\kappa)$

return $\mathsf{sk} = (\mathsf{sk}_1, \mathsf{sk}_2), \mathsf{pk} = (\mathsf{pk}_1, \mathsf{pk}_2)$

When signing a message m, the signer obtains a Σ_1-signature for m, followed by a Σ_2-signature for (m, σ_1). The hybrid signature is then the pair (σ_1, σ_2). In practice, these pairs can be implemented as a simple concatenation if σ_1 has predictable length, or a concatenation with a separator. The pseudocode of the signing and verifying functions is presented below.

$\mathcal{H}(\Sigma_1, \Sigma_2).\texttt{Sign}(m, \mathsf{sk} = (\mathsf{sk}_1, \mathsf{sk}_2))$

$\sigma_1 \leftarrow \Sigma_1.\texttt{Sign}(m, \mathsf{sk}_1)$

$\sigma_2 \leftarrow \Sigma_2.\texttt{Sign}((m, \sigma_1), \mathsf{sk}_2)$

return $\sigma = (\sigma_1, \sigma_2)$

$\mathcal{H}(\Sigma_1, \Sigma_2).\texttt{Verify}(m, \sigma, \mathsf{pk})$

$(\sigma_1, \sigma_2) = \sigma$ and $(\mathsf{pk}_1, \mathsf{pk}_2) = \mathsf{pk}$

return $(\Sigma_1.\texttt{Verify}(m, \sigma_1, \mathsf{pk}_1) \wedge$
$\Sigma_2.\texttt{Verify}((m, \sigma_1), \sigma_2, \mathsf{pk}_2))$

We include the result below, which can be proven immediately using the fact that the underlying schemes are correct.

Lemma 1. *If Σ_1 and Σ_2 are correct, then $\mathcal{H}(\Sigma_1, \Sigma_2)$ is also correct.*

Security Analysis. We now show that our hybrid scheme $\mathcal{H}(\Sigma_1, \Sigma_2)$ maintains the security guarantees of its underlying components. In the statements, we use $X \in \{C, Q\}$ to specify whether we consider classical or post-quantum security.

The following Lemma 2 can be derived from the work of Bindel et al. [6] (Theorem 7).

Lemma 2. *If Σ_1 is X-EUF-CMA secure, then $\mathcal{H}(\Sigma_1, \Sigma_2)$ is X-EUF-CMA secure as well.*

We add the result below, stating that $\mathcal{H}(\Sigma_1, \Sigma_2)$ does not only maintain EUF-CMA security, but also maintains Σ_2's SUF-CMA security guarantees. We include the proof in the appendix.

Lemma 3. *If Σ_2 is X-EUF-CMA secure (resp. X-SUF-CMA) secure, it follows that $\mathcal{H}(\Sigma_1, \Sigma_2)$ is X-EUF-CMA secure (resp. X-SUF-CMA) as well.*

We recall that ECDSA achieves C-EUF-CMA security in the random bijection model, while Dilithium achieves Q-SUF-CMA security in the quantum random oracle model, and we note that our hybrid scheme and proofs use the underlying components as black-boxes. Then, the above shows that our scheme $\mathcal{H}(\text{ECDSA}, \text{Dilithium})$ at least maintains the security guarantees of ECDSA and Dilithium (in their corresponding models).

5 A SK-Friendly Implementation

Security keys often run on embedded hardware devices with tight performance constraints. Our work is based on the open source security key *OpenSK* [31]. OpenSK is a firmware that implements CTAP 2.1. It works as an application on top of the embedded operating system TockOS [23]. For this work, we run OpenSK on a Nordic nRF52840 development kit [30] with a 64 MHz ARM Cortex-M4F MCU. The nRF52840 comes with a TRNG for randomness, and we run all CTAP communication over USB.

To support different hardware targets, we want our firmware including Dilithium, namely the key generation and signing algorithm, to fit 64 kB of RAM. For embedded hardware, we discuss various trade-offs between speed, memory usage and key sizes. We describe our changes to Dilithium compared to the reference implementation. We focus on obtaining a hardware security key-friendly Dilithium implementation for all Dilithium modes.

5.1 CTAP Requirements

Time to login affects usability. In addition, there are some limits for FIDO operations in the specification:

- User presence and user verification tokens usually timeout after 30 s (i.e., see 5. in [11]), but are guaranteed to be valid for at least 10 s. We therefore aim for commands to finish within 10 s.
- The size of a CTAP message over USB cannot exceed 7609 B (see 11.2.4. Message and packet structure in [11]).

Following the command naming from Sect. 2.3, this yields the following priorities:

R1 ⇒ Key generation must finish in less than 10 s.
R2 ⇒ Key pairs must be smaller than 7 kB.
R3 ⇒ The private key should be small to allow storing additional credentials.
A1 ⇒ The login operation is more frequent than registration. Signing should be as fast as possible.
A2 ⇒ A private key and signature together must be smaller than 7 kB.

The Dilithium modes 3 and 5 achieve the desired security requirements. However, for the reference implementation, they fail some requirements due to the large sizes of the key pair and signature. Namely, both miss requirements R3 and A2, and Dilithium5 misses requirement R2.

In the following sections, we describe how we achieve these requirements within the memory limits of embedded hardware. Our main focus consists of reducing the private key size and the memory footprint significantly. The experiments for speed benchmarks can be found in Sect. 6.

5.2 Dilithium Optimizations

Our implementation offers two modes: first, a high speed mode, which follows the original implementation with the exception that we reduce the key size. Second, a low memory footprint mode. To reduce Dilithium's memory footprint, we used known optimization tricks, similarly to [7]. We recompute some intermediate values and effectively trade additional computations (performance) to reduce memory usage.

Both the key generation and the signing algorithm of Dilithium require computations on vectors and matrices of polynomials stored on the stack memory, together with intermediate results. The signing algorithm of the reference implementation of Dilithium [32] keeps on stack 49 such polynomials for Dilithium2, 76 for Dilithium3, and 118 for Dilithium5. Each such polynomial requires 1 kB. The secret key and the array of bytes used to compute the signature are stored on the stack at the same time, which leads to a stack usage of at least 53 kB for Dilithium2, 83 kB for Dilithium3, and 127 kB for Dilithium5. This makes the reference implementation of Dilithium infeasible for our RAM target, especially since we aim for the security levels of Dilithium3 and Dilithium5.

As a first measure, we take into account the life cycle of variables and we arrange the code into multiple blocks, such that the polynomials are only stored on the stack when needed, and afterwards the memory can be recycled. This is still not enough to meet our requirements.

Fortunately, the computations on polynomials are done sequentially (polynomial by polynomial), and OpenSK does not have parallel execution. This enables us to only store a few polynomials at a time, instead of a significant number of large structures. While using this approach reduces the stack usage, it requires some of the intermediate results to be recomputed, and hence it increases the runtime significantly.

Discarding Information from the Secret Key. Dilithium's secret key is an array of bytes comprising the encoding of an array of polynomials, t_0, and the information necessary for computing the array t_0. At a high level, from the secret key, one can derive a matrix of polynomials, A, and two vectors of polynomials, s_1 and s_2. The array t_0 is obtained by reducing each coefficient of $t = A \cdot s_1 + s_2$ modulo 2^d, where d is a parameter. Then, storing the encodings of t_0 is not necessary. To further decrease Dilithium's memory footprint, we can simply recompute these polynomials when signing instead.

Encoding a single polynomial of t_0 into the secret key takes 416 B. In the case of Dilithium2, the encoding of t_0 requires 1664 B. For Dilithium3 and Dilithium5, the encoding requires 2496 B and 3328 B respectively. Then, the size of the secret key gets reduced significantly: from 2528 B to 864 B in the case of Dilithium2, from 4000 B to 1504 B in Dilithium3, and from 4864 B to 1536 B in the case of Dilithium5.

Recomputing t_0 every time we sign helps us decrease Dilithium's memory footprint even more, with a caveat in terms of runtime since this recomputation needs to be done whenever we sign a message. Indeed, this change negatively

affects the performance of Dilithium, but it remains reasonable (see experiments in Sect. 6).

Only Storing a 32 B Seed. Dilithium's key generation uses a 32 B seed as source of randomness, which is then expanded to compute the components of the secret and public keys. We can store only this seed and recompute the secret key deterministically based on the stored seed whenever we sign. This adds a small runtime overhead, while saving a significant amount of storage space. For Dilithium5, we reduce the private key size from 4864 kB to 32 kB. From our benchmarks in Sect. 6.2, we can see that the speed overhead is 8.2%.

We want to note that discarding information from the secret key during computation is still useful: recomputing the polynomials in the vector t_0 when needed requires less stack memory than storing its encoding.

5.3 CTAP Implementation

To indicate support for $\mathcal{H}(\text{ECDSA}, \text{Dilithium})$, we added a new algorithm identifier *Hybrid*. When a relying party requests a Hybrid credential, we follow the CTAP procedure as usual. For simplicity, the only change of the encoding of public keys compared to that in ECDSA is the addition of an extra field with the bytes of the Dilithium public key. For registration of server-side credentials, we only add a 32 B seed of the Dilithium private key that is part of the Hybrid credential. All data is encoded as a CBOR map.

During authentication, the signature is computed with $\mathcal{H}(\text{ECDSA}, \text{Dilithium})$. The partial ECDSA signature is ASN.1 DER encoded like standard ECDSA signatures in CTAP.

5.4 Side-Channel Resilience

As per our attacker model, local attackers are out of scope, and we consider time-based remote side-channels only. Our Dilithium implementation should not leak information about its secret key through the computation time as measurable from outside the device.

The paper introducing Dilithium (Section 5.4 in [13]) explains that their implementation does not leak information about the secret key. Indeed, as Dilithium's signing algorithm may attempt to generate multiple signatures until one that satisfies a set of conditions is found, an adversary can gain information about the number of attempts, or about the conditions previous attempts did not meet. The reasons why a signature attempt is rejected do not depend on the secret key, instead they are based on pseudorandom information. Hence determining which conditions where the reason why a signature attempt was rejected does not help the adversary derive information about the secret key.

Our modifications to Dilithium indeed change the computation compared to the reference implementation. However, our implementation still does not branch depending on the secret data, and hence we maintain the same guarantees.

6 Experiments

We benchmark Dilithium on different target architectures, compare the 3 modes, and evaluate the speed difference of the stack optimized version.

6.1 Dilithium Reference Implementation

Achieving higher security levels demands a higher run time and space usage. Table 2 states the average speed of the Dilithium key generation and signing algorithms over 1000 executions on an x86-64 architecture[1], and the size of the keys and the signature.

Table 2. Average run times on an x86-64 architecture and the key and signature sizes of the reference implementation of Dilithium.

Scheme	Runtime (ms) of		Key size (bytes) of		
	KeyGen	Sign	Private	Public	Sign.
Dilithium2	0.08	0.31	2528	1312	2420
Dilithium3	0.15	0.53	4000	1952	3293
Dilithium5	0.22	0.61	4864	2592	4595

6.2 Dilithium Embedded

The changes from Sect. 5.2 enabled us to execute Dilithium in all modes on the Nordic nRF52840 development kit [30]. The performance was measured on the device and the elapsed time printed out via the debugging interface. Table 3 shows the performance we have obtained. In what we call speed mode, we selectively apply some stack optimizations to be able to sign messages with Dilithium2 on embedded hardware. This allows evaluating the impact of the recomputations only applied to stack mode. To measure the computational cost of our hybrid scheme, we ran the equivalent experiment to the Dilithium benchmarks, but we use the full hybrid scheme.

If not stated otherwise, all binaries are compiler-optimized for size. To compare our runtime to other benchmarks, we also show results compiled for speed in Table 4. Note that the code runs as an application on top of an operating system. Therefore, performance benchmarks don't directly compare to other implementations, as some time is spent inside i.e., syscalls. For an estimate of our relative performance when compiled for speed, we convert the measured time to clock cycles by multiplying with the processor speed of 32768 kHz. Those numbers are reported with their relative performance compared to Bos et al. [7].

The binary size of an application running Dilithium on TockOS is 9.3 kB with compiler optimization level -Oz. This size increases to 26.8 kB using -O3.

[1] We have used a MacBook Pro (13-inch, 2020), with processor 2.3 GHz Quad-Core Intel Core i7, and memory 16 GB 3733 MHz LPDDR4X.

Table 3. We show the performance obtained by our Optimized Stack mode implementation of Dilithium on the Nordic nRF52840 development kit [30]. The runtime in milliseconds is averaged over 1000 executions, and the stack usage is measured with stack painting. We added runtime speed for ECDSA as a baseline, and to explain the difference between pure Dilithium and Hybrid measurements. Signing with Dilithium2 speed mode exceeds our target memory usage.

Key generation	Stack (in kB)	Runtime (ms)		Signing	Stack (in kB)	Runtime (ms)	
		Pure	Hybrid			Pure	Hybrid
ECDSA	0.3	115.7		ECDSA	3.0	188.0	
Dilithium2 (speed mode)	41.6	70.3	192.0	Dilithium2 (speed mode)	77.1	420.4	687.8
Dilithium2	14.4	82.3	207.5	Dilithium2	17.0	1053.1	1417.5
Dilithium3	19.4	142.4	258.5	Dilithium3	17.9	2077.3	2420.7
Dilithium5	21.4	271.4	393.1	Dilithium5	19.2	3305.1	3378.5

Table 4. We repeated the Dilithium benchmarks from Table 3 with the compiler optimizing for speed rather than binary size (-O3 instead of -Oz) to compare them. We also compare the speed to Bos et al. [7]. Since they report clock cycles, we estimate ours by multiplying our runtimes with our clock frequency.

Key generation	Runtime -O3 (ms)	Relative to		Signing	Runtime -O3 (ms)	Relative to	
		-Oz	Bos			-Oz	Bos
ECDSA	51.9	45%		ECDSA	73.3	39%	
Dilithium2 (speed mode)	63.2	90%	71%	Dilithium2 (speed mode)	363.0	86%	64%
Dilithium2	72.3	88%	81%	Dilithium2	956.1	91%	170%
Dilithium3	129.5	91%	83%	Dilithium3	1955.0	94%	176%
Dilithium5	223.6	82%	85%	Dilithium5	2723.8	82%	201%

We highlight that our Dilithium implementation runs solely on the stack; no heap is required. This benefits embedded devices that don't support heap allocation. The memory footprint was measured with stack painting: Before entering the function that we want to measure, we write a fixed byte pattern into the unused stack. After the function returns, we read back the stack to see where the byte pattern was overwritten.

With this method, we can measure the actual stack usage of each function. Therefore, our reported numbers represent our implementation and depend e.g., on the compiler version used. This explains why our numbers are higher than reported theoretical optima (see [7]). The stack usage is deterministic and does not depend on the inputs' concrete values. Our measurement method also implies

that input messages for signing and the RNG are not counted for its memory usage, but outputs are.

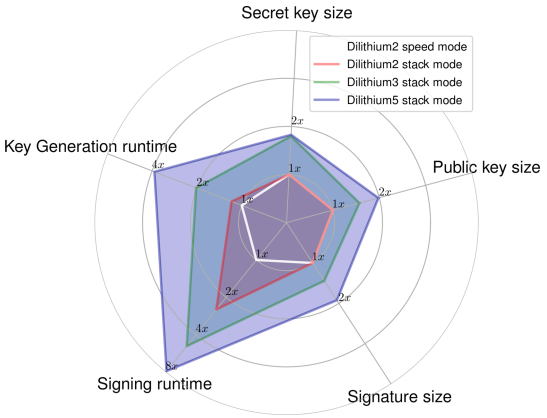


Fig. 3. Comparison of sizes and speeds of Dilithium modes on embedded hardware. The reference in white is Dilithium2 without recomputating parts of the key to save memory. To set the computation speed into perspective, we compare the scaling with the key and signatures sizes. Note that the shown key sizes are after restoring from the 32 byte seed.

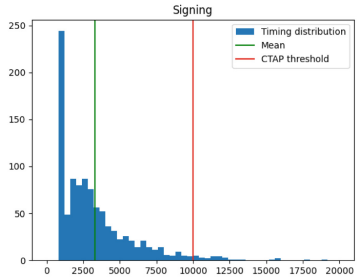
Figure 3 summarizes how Dilithium modes scale, and how our stack optimizations impact the speed of operations.

Since Dilithium's signing has a retry loop, its signing speed has a long tail. The distribution of measurements for our Dilithium5 signing benchmark is shown in Fig. 4. To not cause timeouts, CTAP operations should be faster than 10 s. Signing with Dilithium5 achieves that in 97% of the operations. Key generation is faster and more predictable, taking 271 ms on average, with 1 ms standard deviation.

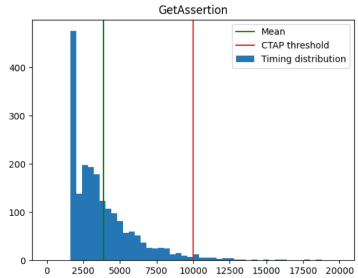
6.3 Register and Authenticate Speed

Different from pure cryptography measurements above, the performance measurements for the CTAP commands MakeCredential and GetAssertion were measured on the USB host, and include a full message exchange. All measurements use server-side keys (see Sect. 5.1). MakeCredential takes 792 ms with 2 ms standard deviation. GetAssertion has the same long-tail timing distribution as signing (see Fig. 4).

We simulated 2000 calls to the security key to register and login. MakeCredential calls took between 786 and 797 milliseconds, whereas GetAssertion has much more variance, due to its signing retry logic described above (see Fig. 4). The time distribution shows that 20% of all calls finish within 2 s. On average, a command takes 3.9 s to complete. 97% of all authentication attempts finished within the CTAP timeout of 10 s, as stated in our requirements in Sect. 5.1.



(a) The sign operation has a retry loop that discards insecure parameters. The signing speed is therefore highly non-deterministic.



(b) GetAssertion commands have a similar long tail, depending on the number of retries when signing with Dilithium.

Fig. 4. Timing distributions of signing and the CTAP command GetAssertion, using the Dilithium5 mode.

7 Conclusion

In this paper, we proposed a practical way to upgrade security-key authentication via FIDO's CTAP to PQC. To do so, we have designed and evaluated a hybrid digital-signature scheme that combines a classical scheme, ECDSA, with a PQC one, Dilithium. This hybrid scheme ensures that the security guarantees of each underlying scheme are maintained even when one of the scheme becomes insecure

To demonstrate the practicality of this scheme, we have implemented it in the open-source security-key firmware OpenSK, benchmarked its performance, and released our contribution as open-source software with an Apache2 license. This way, we encourage other researchers to reproduce our results on a nRF52840 chip.

Our implementation is designed to overcome the intrinsic resource limitations of current security key hardware platforms while maintaining reasonable runtimes. Our evaluation of this implementation has demonstrated its feasibility even when using Dilithium's highest security mode, which comes with the highest resource requirements.

A Appendix

We include the formal proof that was omitted in the main body of the paper.

Lemma 3. *If Σ_2 is X-EUF-CMA secure (resp. X-SUF-CMA) secure, it follows that $\mathcal{H}(\Sigma_1, \Sigma_2)$ is X-EUF-CMA secure (resp. X-SUF-CMA) as well.*

Proof. We show that, for every X-adversary $\mathcal{A}$ that wins the X-EUF-CMA (resp. X-SUF-CMA) security game for $\mathcal{H}(\Sigma_1, \Sigma_2)$ with probability $p_{\mathcal{A}}$, there is an adversary $\mathcal{B}$ that wins Σ_2's X-EUF-CMA (resp. X-SUF-CMA) security game with probability $p_{\mathcal{B}} \geq p_{\mathcal{A}}$.

The adversary $\mathcal{B}$ can be constructed as follows:

- $\mathcal{B}$ receives the Σ_2 public key pk_2 from the challenger $\mathcal{C}_{\Sigma_2}$.
  Since $\mathcal{A}$ expects a hybrid public key, $\mathcal{B}$ generates its own pair of Σ_1 keys $(\mathsf{sk}_1, \mathsf{pk}_1) \leftarrow\!\!\$\ \Sigma_1.\mathtt{KeyGen}(1^\kappa)$, and then sends $\mathsf{pk} = (\mathsf{pk}_1, \mathsf{pk}_2)$ to $\mathcal{A}$.
  Note that the keys received by $\mathcal{A}$ are generated from the same probability distribution as in $\mathcal{H}(\Sigma_1, \Sigma_2)$'s security game.
- When receiving a message query m_i from $\mathcal{A}$, $\mathcal{B}$ uses its own secret key sk_1 to compute the first part of the hybrid signature: $\sigma_{1,i} \leftarrow\!\!\$\ \Sigma_1.\mathtt{Sign}(m_i, \mathsf{sk}_1)$.
  Afterwards, to obtain the Σ_2-component of the hybrid signature, $\mathcal{B}$ sends $m'_i := (m_i, \sigma_1)$ as a signing query to the challenger $\mathcal{C}_{\Sigma_2}$ and obtains $\sigma_{2,i} = \Sigma_2.\mathtt{Sign}(m'_i, \mathsf{sk}_2)$.
  When receiving $\sigma_{2,i}$ from $\mathcal{C}_{\Sigma_2}$, $\mathcal{B}$ computes $\sigma_i := (\sigma_{1,i}, \sigma_{2,i})$ and sends it to $\mathcal{A}$. Note that σ_i is a valid hybrid signature:
  $\mathcal{H}(\Sigma_1, \Sigma_2).\mathtt{Verify}(m_i, \sigma_i, \mathsf{pk}) = \mathsf{true}$
- When receiving the forgery $\left(m^*, \sigma^* = (\sigma_1^*, \sigma_2^*)\right)$ from the adversary $\mathcal{A}$, $\mathcal{B}$ obtains its own forgery $\left((m^*, \sigma_1^*), \sigma_2^*\right)$ and sends it to $\mathcal{C}_{\Sigma_2}$.

Since $\mathcal{B}$ simulates the X-EUF-CMA (resp. X-SUF-CMA) security game for $\mathcal{H}$ perfectly towards $\mathcal{A}$, $\mathcal{A}$ maintains its success probability $p_\mathcal{A}$.

We show that, whenever $\mathcal{A}$ wins the simulated game, $\mathcal{B}$ wins the X-EUF-CMA (resp. X-SUF-CMA) security game for Σ_2.

If $\mathcal{A}$ wins the simulated X-EUF-CMA security game, $m^* \notin \{\text{queries } m_i\}$. It immediately follows that $(m^*, \sigma_1^\star) \notin \{\text{queries } m'_i\}$.

If $\mathcal{A}$ wins the simulated X-SUF-CMA security game, $(m^*, \sigma^*) \notin \{(m_i, \sigma_i) \mid m_i \text{ query}, \sigma_i \text{ response}\}$. If this is the case, we need to show that $\mathcal{B}$ has never received σ_2^* as a response from $\mathcal{C}_2$ to the signing query (m^*, σ_1^*). Assuming that $\left((m^*, \sigma_1^*), \sigma_2^*\right) = (m'_i, \sigma_{2,i})$ for some query-response pair $(m'_i, \sigma_{2,i})$ in $\mathcal{B}$'s interaction with $\mathcal{C}_2$, we obtain that $(m^*, \sigma^*) = (m_i, \sigma_i)$, which contradicts that $\mathcal{A}$'s forgery was successful.

Both in the X-EUF-CMA case and in the X-SUF-CMA case, if $\mathcal{A}$ has sent a successful forgery, then $\mathcal{H}(\Sigma_1, \Sigma_2).\mathtt{Verify}(m^*, \sigma^*, \mathsf{pk}) = \mathsf{true}$ holds, and hence $\Sigma_2.\mathtt{Verify}((m^*, \sigma_1^*), \sigma_2^*, \mathsf{pk}_2)$ holds as well.

It follows that $\mathcal{B}$ wins the X-EUF-CMA (resp. X-SUF-CMA) security game for Σ_2 with probability $p_\mathcal{B} \geq p_\mathcal{A}$.

Finally, as Σ_2 is X-EUF-CMA (resp. X-SUF-CMA) secure, $p_\mathcal{B} \in \mathsf{negl}(\kappa)$, and therefore $p_\mathcal{A} \in \mathsf{negl}(\kappa)$.

Since $\mathcal{A}$ was chosen arbitrarily, we obtain that every X-adversary has negligible probability in winning $\mathcal{H}(\Sigma_1, \Sigma_2)$'s X-EUF-CMA (resp. X-SUF-CMA) security game. □

References

1. Abdulrahman, A., Hwang, V., Kannwischer, M.J., Sprenkels, D.: Faster kyber and dilithium on the cortex-M4. In: Ateniese, G., Venturi, D. (eds.) ACNS 2022. LNCS, vol. 13269, pp. 853–871. Springer, Cham (2022). https://doi.org/10.1007/978-3-031-09234-3_42

2. Azarderakhsh, R., Elkhatib, R., Koziel, B., Langenberg, B.: Hardware deployment of hybrid PQC: SIKE+ECDH. In: Garcia-Alfaro, J., Li, S., Poovendran, R., Debar, H., Yung, M. (eds.) SecureComm 2021. LNICST, vol. 399, pp. 475–491. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-90022-9_26
3. Bernstein, D.J., Hülsing, A., Kölbl, S., Niederhagen, R., Rijneveld, J., Schwabe, P.: Leveraging secondary storage to simulate deep 54-qubit sycamore circuits. In: Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, pp. 2129–2146 (2019)
4. Beullens, W.: Improved cryptanalysis of UOV and rainbow. In: Canteaut, A., Standaert, F.-X. (eds.) EUROCRYPT 2021, Part I. LNCS, vol. 12696, pp. 348–373. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-77870-5_13
5. Bindel, N., Cremers, C., Zhao, M.: FIDO2, CTAP 2.1, and WebAuthn 2: provable security and post-quantum instantiation. In: 2023 IEEE Symposium on Security and Privacy (SP), Los Alamitos, CA, USA, pp. 674–693. IEEE Computer Society (2023). https://doi.org/10.1109/SP46215.2023.00039
6. Bindel, N., Herath, U., McKague, M., Stebila, D.: Transitioning to a quantum-resistant public key infrastructure. In: Lange, T., Takagi, T. (eds.) PQCrypto 2017. LNCS, vol. 10346, pp. 384–405. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-59879-6_22
7. Bos, J.W., Renes, J., Sprenkels, A.: Dilithium for memory constrained devices. In: Batina, L., Daemen, J. (eds.) AFRICACRYPT 2022. LNCS, vol. 13503, pp. 217–235. Springer, Cham (2022). https://doi.org/10.1007/978-3-031-17433-9_10
8. Brendel, J., Cremers, C., Jackson, D., Zhao, M.: The provable security of Ed25519: theory and practice. Cryptology ePrint Archive, Paper 2020/823 (2020). https://eprint.iacr.org/2020/823
9. Castryck, W., Decru, T.: An efficient key recovery attack on SIDH. In: Hazay, C., Stam, M. (eds.) EUROCRYPT 2023. LNCS, vol. 14008, pp. 423–447. Springer, Cham (2023). https://doi.org/10.1007/978-3-031-30589-4_15
10. CRYSTALS-Dilithium Algorithm Specifications and Supporting Documentation. https://pq-crystals.org/dilithium/data/dilithium-specification-round3-20210208.pdf. Accessed 08 Feb 2023
11. Client to Authenticator Protocol (CTAP). https://fidoalliance.org/specs/fido-v2.1-ps-20210615/fido-client-to-authenticator-protocol-v2.1-ps-errata-20220621.html. Accessed 05 Feb 2023
12. Dowling, B., Hansen, T.B., Paterson, K.G.: Many a mickle makes a muckle: a framework for provably quantum-secure hybrid key exchange. In: Ding, J., Tillich, J.-P. (eds.) PQCrypto 2020. LNCS, vol. 12100, pp. 483–502. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-44223-1_26
13. Ducas, L., et al.: CRYSTALS-dilithium: a lattice-based digital signature scheme. IACR Trans. Cryptogr. Hardw. Embed. Syst. **2018**(1), 238–268 (2018). https://doi.org/10.13154/tches.v2018.i1.238-268. https://tches.iacr.org/index.php/TCHES/article/view/839
14. Fersch, M., Kiltz, E., Poettering, B.: On the provable security of (EC)DSA signatures. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, CCS 2016, pp. 1651–1662. Association for Computing Machinery, New York (2016). https://doi.org/10.1145/2976749.2978413
15. FIDO Alliance. https://fidoalliance.org/. Accessed 05 Feb 2023
16. FIDO Alliance security reference. https://fidoalliance.org/specs/fido-v2.0-id-20180227/fido-security-ref-v2.0-id-20180227.html. Accessed 05 Feb 2023

17. Greconici, D.O.C., Kannwischer, M.J., Sprenkels, D.: Compact dilithium implementations on Cortex-M3 and Cortex-M4. IACR Trans. Cryptogr. Hardw. Embed. Syst. **2021**(1), 1–24 (2020). https://doi.org/10.46586/tches.v2021.i1.1-24
18. Kannwischer, M.J., Rijneveld, J., Schwabe, P., Stoffelen, K.: pqm4: testing and benchmarking NIST PQC on ARM Cortex-M4. Cryptology ePrint Archive, Paper 2019/844 (2019). https://eprint.iacr.org/2019/844
19. Kiltz, E., Lyubashevsky, V., Schaffner, C.: A concrete treatment of Fiat-Shamir signatures in the quantum random-oracle model. In: Nielsen, J.B., Rijmen, V. (eds.) EUROCRYPT 2018. LNCS, vol. 10822, pp. 552–586. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-78372-7_18
20. Koblitz, N.: Elliptic curve cryptosystems. Math. Comput. **48**, 203–209 (1987)
21. Information Technology Laboratory: Digital Signature Standard (DSS). Technical report, National Institute of Standards and Technology (2013). https://doi.org/10.6028/nist.fips.186-4
22. Lang, J., Czeskis, A., Balfanz, D., Schilder, M., Srinivas, S.: Security keys: practical cryptographic second factors for the modern web. In: Grossklags, J., Preneel, B. (eds.) FC 2016. LNCS, vol. 9603, pp. 422–440. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-662-54970-4_25
23. Levy, A., et al.: Multiprogramming a 64kB computer safely and efficiently. In: Proceedings of the 26th Symposium on Operating Systems Principles, SOSP 2017, pp. 234–251. ACM, New York (2017). https://doi.org/10.1145/3132747.3132786
24. Li, S., et al.: FALCON: a Fourier transform based approach for fast and secure convolutional neural network predictions. CoRR abs/1811.08257 (2018). http://arxiv.org/abs/1811.08257
25. Lipp, B.: An analysis of hybrid public key encryption. Cryptology ePrint Archive, Paper 2020/243 (2020). https://eprint.iacr.org/2020/243
26. Marzougui, S., Ulitzsch, V., Tibouchi, M., Seifert, J.P.: Profiling side-channel attacks on dilithium: a small bit-fiddling leak breaks it all. Cryptology ePrint Archive, Paper 2022/106 (2022). https://eprint.iacr.org/2022/106
27. Migliore, V., Gérard, B., Tibouchi, M., Fouque, P.-A.: Masking dilithium. In: Deng, R.H., Gauthier-Umaña, V., Ochoa, M., Yung, M. (eds.) ACNS 2019. LNCS, vol. 11464, pp. 344–362. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-21568-2_17
28. NIST Announces First Four Quantum-Resistant Cryptographic Algorithms. https://www.nist.gov/news-events/news/2022/07/nist-announces-first-four-quantum-resistant-cryptographic-algorithms. Accessed 07 Feb 2023
29. NIST Post-Quantum Cryptography FAQs. https://csrc.nist.gov/Projects/post-quantum-cryptography/faqs. Accessed 13 Feb 2023
30. Nordic nrf52840. https://www.nordicsemi.com/Products/Development-hardware/nrf52840-dk. Accessed 05 Feb 2023
31. OpenSK. https://github.com/google/OpenSK. Accessed 05 Feb 2023
32. PQCrystals: Dilithium. https://github.com/pq-crystals/dilithium. Accessed 10 Feb 2023
33. Raavi, M., Wuthier, S., Chandramouli, P., Balytskyi, Y., Zhou, X., Chang, S.-Y.: Security comparisons and performance analyses of post-quantum signature algorithms. In: Sako, K., Tippenhauer, N.O. (eds.) ACNS 2021. LNCS, vol. 12727, pp. 424–447. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-78375-4_17
34. Rivest, R.L., Shamir, A., Adleman, L.M.: A method for obtaining digital signatures and public key cryptosystems. Commun. ACM **21**(2), 120–126 (1978). https://doi.org/10.1145/359340.359342