Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Dept. of Information Security and Communication Technology

Ludvig Digné

Towards Quantum-Resilient Authentication: Implementing Hybrid Signatures in FIDO2 Authenticators

Master's thesis in Digital Infrastructure and Cyber Security
Supervisor: Tjerand Silde
Co-supervisor: Magnus Ringerud, Sigurhjörtur Snorrason, and Trond Peder Hagen

June 2024

NTNU
Norwegian University of Science and Technology

PONE BIOMETRICS

Ludvig Digné

Towards Quantum-Resilient Authentication: Implementing Hybrid Signatures in FIDO2 Authenticators

Master's thesis in Digital Infrastructure and Cyber Security
Supervisor: Tjerand Silde
Co-supervisor: Magnus Ringerud, Sigurhjörtur Snorrason, and Trond Peder Hagen
June 2024

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Dept. of Information Security and Communication Technology

NTNU
Norwegian University of Science and Technology

NTNU
Norwegian University of
Science and Technology

Towards Quantum-Resilient Authentication: Implementing Hybrid Signatures in FIDO2 Authenticators

Digné, Ludvig

Submission date: June 2024
Main supervisor: Silde, Tjerand, NTNU
Co-supervisors: Ringerud, Magnus, PONE Biometrics,
Snorrason, Sigurhjörtur, PONE Biometrics, and
Hagen, Trond Peder, PONE Biometrics

Norwegian University of Science and Technology
Department of Information Security and Communication Technology

Title:	Towards Quantum-Resilient Authentication: Implementing Hybrid Signatures in FIDO2 Authenticators
Student:	Digné, Ludvig

Problem description:

FIDO2 aims to replace password-based authentication with passkeys using public key cryptography. PONE Biometrics' OFFPAD, a credit card-sized device with a fingerprint reader, implements this protocol. Users authenticate to online services using a fingerprint-unlocked passkey, shifting authentication to possession-based (physical device) rather than knowledge-based (passwords). Public key cryptography systems are threatened by the advent of quantum computers which is why NIST is standardizing quantum-secure algorithms. The purpose of this project is to investigate how CRYSTALS-Dilithium, a quantum-resistant digital signature algorithm, can be implemented on resource-constrained devices in the context of FIDO2.

Approved on:	2024-02-16
Main supervisor:	Silde, Tjerand, NTNU
Co-supervisors:	Ringerud, Magnus, PONE Biometrics, Snorrason, Sigurhjörtur, PONE Biometrics, and Hagen, Trond Peder, PONE Biometrics

Abstract

Quantum computers get more powerful every year, and if a sufficiently large quantum computer is ever built, it would render all public key cryptography used today insecure. To ensure the security of future communication, the National Institute of Standards and Technology (NIST) in the United States is standardizing post-quantum cryptography (PQC). CRYSTALS-Dilithium and Falcon are two of the digital signature algorithms being standardized, and these are based on lattice problems believed to be computationally difficult to solve even for a large quantum computer. These algorithms must be integrated into the protocols that are being used every day. This work focuses on how the FIDO2 authentication standard, which aims to replace passwords with public key cryptography, can be made quantum-secure. We implement an authenticator capable of PQC that demonstrates efficient authentication against a test server using a hybrid signature construction. The results show that FIDO2 can migrate towards PQC, but there are challenges. These challenges include, but are not limited to, deciding whether to use hybrid vs. pure PQC, and determining the optimal structure of a hybrid construction for security, performance, and ease of implementation. We finally discuss trade-offs concerning algorithm choices, the impact of secure elements, and protection against side-channel attacks.

Sammanfattning

Kvantdatorer blir kraftfullare för varje år som går, och om en tillräckligt stor kvantdator någonsin byggs skulle den göra all offentlig nyckelkryptografi som används idag osäker. För att säkerställa säkerheten i framtida kommunikation standardiserar National Institute of Standards and Technology (NIST) i USA kvantesäker kryptografi (PQC). CRYSTALS-Dilithium och Falcon är två av de digitala signaturalgoritmerna som standardiseras, och dessa är baserade på gitterproblem som anses vara beräkningsmässigt svåra att lösa även för en stor kvantdator. Dessa algoritmer måste integreras i de protokoll som används varje dag. Detta arbete fokuserar på hur FIDO2-autentiseringsstandarden, som syftar till att ersätta lösenord med offentlig nyckelkryptografi, kan göras kvantesäker. Vi implementerar en autentiserare kapabel till PQC som demonstrerar effektiv autentisering mot en testserver genom användning av en hybrid signaturkonstruktion. Resultaten visar att FIDO2 kan migrera mot PQC, men det finns utmaningar. Dessa utmaningar inkluderar, men är inte begränsade till, beslut om att använda hybrid eller ren PQC, och att avgöra den optimala strukturen för en hybridkonstruktion för säkerhet, prestanda, och enkelhet i implementeringen. Slutligen diskuterar vi prestandaavvägningar gällande algoritmval, påverkan av säkra element (secure elements), och skydd mot sidokanals-attacker.

Preface

The work presented in this thesis was conducted in the department of Information Security and Communication Technology of the Norwegian University of Science and Technology (NTNU), and the Background section of this thesis builds on my specialization project [Dig23] that was conducted during the autumn of 2023.

I took my bachelor's degree from Chalmers University of Technology in Sweden in industrial engineering and management, and felt towards the end of those studies an attraction towards more technical subjects, specifically cyber security. This led to my pursuing a master's degree in Norway. During these two years, I feel I have truly learned a lot about a field I find genuinely interesting and have met great people along the way. I purposefully chose this master's project, knowing it would be challenging, because I wanted to learn more about the technologies involved, but also because it would be that much more rewarding than choosing a project with the intent to "coast by".

I want to express my sincere gratitude towards my main supervisor, Tjerand Silde, for his excellent guidance, and whose passion for cryptography is inspiring. This work was done in collaboration with PONE Biometrics, and I also want to extend my gratitude to the folks there for the opportunity and the great interest you all have shown in my work. A special thanks to Sigurhjörtur at PONE Biometrics for his technical guidance and his work on the server side of the project, without whom this thesis would not be as complete.

Lastly, I want to thank my family, and girlfriend, for their tireless support and never-ending encouraging words.

Contents

List of Figures

List of Tables

List of Listings

List of Acronyms

API Application Programming Interface.

CBOR Concise Binary Object Representation.

CDDL Concise Data Definition Language.

COSE CBOR Object Signing and Encryption.

CTAP Client to Authenticator Protocol.

ECC Elliptic Curve Cryptography.

ECDSA Elliptic Curve Digital Signature Standard.

HID Human Interface Device.

JSON JavaScript Object Notation.

MCU Microcontroller Unit.

MFA Multi-Factor Authentication.

NIST National Institute for Standards and Technology.

NSA National Security Agency.

OFFPAD Offline Personal Authentication Device.

OQS Open Quantum Safe.

OS Operating System.

PKC Public-Key Cryptography.

PoC Proof of Concept.

PQC Post-Quantum Cryptography.

RP Relying Party.

RTOS Real-Time Operating System.

SCA Side-Channel Attack.

SE Secure Element.

SIS Short Integer Solution.

SNDL Store Now, Decrypt Later.

TLS Transport Layer Security.

UART Universal Asynchronous Receiver-Transmitter.

W3C World Wide Web Consortium.

WebAuthn W3C's Web Authentication.

ZKPoK Zero-Knowledge Proof of Knowledge.

Chapter 1
Introduction

The introduction chapter aims to motivate the relevance of this work. It begins by describing the problem that justifies the need for the research, followed by a definition of the research scope. The chapter also outlines the limitations of the work and presents the research questions along with the contributions of the study. Finally, an outline of the remainder of the thesis is provided.

1.1 Motivation

Authenticating against online services is something many of us do every day of our lives, often done using knowledge-based methods such as passwords. Ideally, these passwords are unique for each service and have a high degree of randomness. However, this idealistic scenario is infrequent due to its inconvenience. This avoidance of inconvenience often results in the proliferation of low-quality and frequently reused passwords, rendering systems vulnerable to adversaries who exploit the presence of passwords stored in databases. To increase both convenience and security, there has been efforts to come up with alternative authentication methods. One of these is the FIDO2 standard developed by the FIDO Alliance, an authentication mechanism based on Public-Key Cryptography (PKC). The fundamental premise is that a user will only have to authenticate locally on a device, called an authenticator, by some means, such as biometric methods or a PIN. Afterward, the device unlocks a private key stored on it and uses it to authenticate against the service holding the corresponding public key. This increases convenience and security. However, it is all based on public key cryptography, which in turn is based on either the factorization problem or the discrete logarithm problem. Simply put, the first problem is about the hardness of factorizing large integers, and the second one is about finding x in $g^x \equiv A \pmod{p}$, given g, A, and a prime number p. These two problems are infeasible to solve on a classical computer with today's algorithms. Quantum computers, on the other hand, are a different beast. A sufficiently large quantum computer could solve these two mathematical problems that public key cryptography is built upon by virtue of

Shor's algorithm [Sho99], effectively rendering encryption and authentication based on these insecure.

The number of years until there is a cryptographically relevant quantum computer is an inherently difficult prediction to make, although many experts seem to agree that the probability it will become a reality within the nearest coming decades is non-negligable [MP21]. Although it is not here today, there still exists a threat in which individuals and larger actors can intercept data encrypted with quantum-vulnerable cryptography during its transmission over the internet. Subsequently, the intercepted data could be decrypted at a later point when the interceptor obtains access to a quantum computer. This phenomenon is called Store Now, Decrypt Later (SNDL). Although this may not be a concern for most data, there are secrets with multi-decade shelf lives that must remain confidential for an extended period of time, such as medical records, trade secrets, or documents related to national security [JMM+22]. Moreover, organizations, such as software vendors, face threats from quantum adversaries if they fail to transition to quantum-secure systems. Quantum adversaries possess the ability to compromise the classical signature schemes utilized by vendors to validate software updates, enabling them to inject malicious code like spyware or backdoors into the vendor's software and distribute it as legitimate [TSZ22]. In the same vein, authentication systems such as FIDO2 that aim to increase security by replacing passwords with cryptography-based authentication mechanisms must adapt to the looming threat of quantum computers. The authentication system is built on classical digital signatures to authenticate a user, which means that a quantum adversary would be able to break the authentication system. Considering these concerns, the urgency of migrating to quantum-secure systems becomes increasingly evident [JMM+22].

Migration of IT systems is often a complex process, which is why much of the literature on the subject urges organizations to begin their transition to quantum-safe systems as soon as possible [AWG+21; JMM+22]. Organizations in which digital signatures play an important role can make the transition from quantum-vulnerable systems to quantum-secure ones easier by utilizing hybrid digital signatures. Rather than replacing existing algorithms with novel, less studied algorithms believed to be secure against quantum computers, the classical algorithm is combined with the new one into a single construction. This makes it so that an adversary must break both algorithms and the overall system security is lower bounded by the strongest algorithm in the construction. In other words, even if the Post-Quantum Cryptography (PQC) algorithm is identified as flawed, the security by the classical algorithm is still guaranteed. This means that a systems security is only potentially increased during a migration, never decreased [JMM+22; GKP+23].

In light of the looming threat of quantum computers, National Institute for

Standards and Technology (NIST) has put much effort into standardizing algorithms deemed secure against quantum computers. The CRYSTALS-Dilithium [BDK+21] digital signature algorithm, which will be referred to simply as Dilithium hereafter, is one of the algorithms selected for standardization. In order to secure FIDO2 authentication against future quantum adversaries, such that the migration towards a world less reliant on passwords can continue, this thesis explores the integration of a hybrid signature consisting of a classical Elliptic Curve Digital Signature Standard (ECDSA) signature and the new Dilithium signature into FIDO2 authentication.

1.2 Research Scope

This work implements an authenticator capable of FIDO2 authentication using quantum-secure hybrid digital signatures. The implementation is simplified and does not reflect the complexity of a production grade authenticator. Instead, it serves as a Proof of Concept (PoC) for experimenting with PQC in a FIDO2 context.

Furthermore, while there are various methods to achieve hybridization, this implementation focuses solely on a specific approach: concatenation of classical and PQC signatures. Additionally, although multiple PQC algorithms are being standardized, this work primarily centers on Dilithium. For comparison and discussion purposes, the PQC signature algorithm Falcon [FHK+18] is also benchmarked alongside Dilithium.

This work only covers implementation on the Nucleo-L476RG development board with an Arm Cortex M4 processor, and the algorithms used in this work have not been optimized for this specific environment but have been employed from pre-existing libraries.

1.3 Limitations

Some restrictions have been deliberately placed to ensure that the work can be completed within the allocated time frame of 21 weeks. These are essential for striking a balance between depth and breadth.

ID	Limitations
L1	The code will have emphasis on functionality over performance, particularly on the client side, where the majority of the development will be done. Although efforts will be made to select appropriate data structures where there is an obvious choice to do so from a performance perspective, the overarching architectural decisions will not prioritize performance. This approach, while conducive to achieving the functional goals within the given scope, may lead to a less performance-optimized system, deviating from conventional practices often associated with embedded systems.
L2	Secure communication channels over the internet, typically done using TLS, is a fundamental and integral part of internet communications and serves to provide encryption and integrity checks of the data sent between parties. The proposed solution assumes that such a secure channel is established between the client and the service being authenticated against.
L3	The code developed serves only as a PoC of a quantum-secure FIDO2 solution. In no way, shape or form, should it be used in a production environment without having undergone rigorous scrutiny and testing in its intended environment.

1.4 Research Questions

The following research questions have been derived from the presented motivation, research scope, and limitations:

RQ1 With respect to contemporary technology and research, is the FIDO2 standard ready to migrate to quantum-secure authentication?

RQ2 What does a feasible quantum-secure solution for FIDO2 authentication entail?

1.5 Contribution

This work demonstrates the implementation of a PoC authenticator capable of performing quantum-secure authentication in FIDO2, and further explores various aspects of hybridization and general challenges related to quantum-securing FIDO2. While similar work has been performed previously, such as in [GKP+23], this work differs in that it implements a different type of hybridization (concatenation) while also going into more detail of the actual implementation with a significant focus on Client to Authenticator Protocol (CTAP), covering aspects such as serialization and CBOR Object Signing and Encryption (COSE) key structures. Furthermore, this work provides an analysis of the integration challenges and performance trade-offs

between algorithms associated with PQC FIDO2 authentication, offering further perspective on this important matter.

1.6 Related Work

A significant amount of research is being conducted in the field of PQC and its application to systems like FIDO2. NISTs work of standardizing PQC algorithms has laid the groundwork for other research that has shown the security of PQC in FIDO2, as well as the implementation of PQC algorithms like Dilithium in constrained environments for a FIDO2 context. Some of this research will now be presented further.

1.6.1 NIST Standardization Work

Although opinions on the arrival timeline of a sufficiently large quantum computer vary widely, it remains crucial to establish standardized algorithms resistant to quantum threats well in advance. NIST has been at the forefront of evaluating and standardizing various post-quantum algorithms since 2016. This initiative has involved a competition to solicit algorithm candidates. After several rounds of scrutiny of the submissions, the remaining four algorithms are Kyber, Dilithium, Falcon, and SPHINCS$^+$ [Nat23]. Kyber is a key establishment algorithm, and the latter three are digital signature algorithms. Of the three digital signatures, Dilithium and Falcon are the most efficient ones and are both based on the hardness of lattice problems. Although Dilithium has been selected as the primary one, Falcon should be used by applications that require smaller signatures than what Dilithium provides. The SPHINCS$^+$ algorithm is slightly larger and slower than the other two, but it was also selected because it is valuable as a backup, simply because it is based on a different mathematical problem than the other two [Nat22a].

1.6.2 Hybrid Signatures

To smoothen the transition into a quantum-secure digital landscape, hybrid cryptosystems have been proposed. Hybridization methods exist for, e.g., digital signatures. In [BHMS17], different methods are proposed to combine classical digital signatures and post-quantum ones to form a hybrid construction. The fundamental idea is to provide protection against future quantum adversaries while also being secure against classical attackers. In [GKP+23], the authors implemented Dilithium as part of a hybrid construction (specifically “strong nesting”) proposed in [BHMS17]. However, in contrast to the authors of [BHMS17], the authors of [GKP+23] explored the feasability of achieving this under the constraints of embedded hardware. It was performed on an ARM Cortex M4 based development board and showed how it can be applied in the context of passwordless authentication.

1.6.3 Provable PQC Security in FIDO2

In [BCZ23], the authors hold that the most recent version of the FIDO2 standard appears to be "post-quantum ready". They initially prove that FIDO2 is provably secure against classical adversaries. Thereafter, using the same model used to prove the previous, they prove that FIDO2 is also secure when instantiated with PQC-primitives. To add support for authentication using PQC-primitives, they propose to only extend the list of supported algorithms of the server (Relying Party (RP)), and explicitly allow hybrid schemes as exemplified in Subsection 1.6.2.

1.7 Outline

The remainder of this thesis consists of the following five chapters:

Chapter 2: Background introduces the necessary background knowledge for the thesis. This includes terminology and explanations of the OFFPAD, FIDO2 standard, digital signatures and the Dilithium algorithm.

Chapter 3: Methodology outlines the steps taken to implement the final solution. It provides insights into the tools utilized and offers explanations for their selection and the motivations behind their usage. Additionally, it outlines the methodology employed for benchmarking the implemented algorithms.

Chapter 4: Proposed Solution presents the solution, i.e., how the development board was made into an authenticator supporting hybrid signatures for FIDO2 authentication. The implementation of CTAP covers a large portion of this chapter. Code listings for important code sections are shown and explained.

Chapter 5: Performance and Discussion presents benchmarks of the implemented solution, showing metrics such as time for key generation and signing, as well as key and signature sizes for the different algorithms. The discussion section delves into the nuances of hybridization and examines performance trade-offs among different algorithms. Additionally, it explores the significance of Secure Elements (SEs) and Side-Channel Attacks (SCAs) as it pertains to authenticators capable of PQC. Lastly, the chapter addresses some challenges associated with quantum-securing FIDO2.

Chapter 6: Conclusion and Future Work summarizes the thesis and tries to answer the research questions. Ideas for future work are proposed.

Chapter 2
Background

This chapter aims to give the reader the necessary background knowledge to understand the research area and to follow the rest of this work. The work covers implementation of a FIDO2-authenticator capable of performing quantum-secure hybrid digital signatures, which is why the work begins with explaining the OFFPAD authenticator and the benefits of such devices. Next, the FIDO2 standard is explained in more detail, guiding the reader through its components and the processes of registration and authentication. The chapter concludes with an in-depth overview of digital signatures, with particular emphasis on the PQC digital signature algorithm, Dilithium.

2.1 The OFFPAD

The reader will now be introduced to how the possession of a physical device, i.e., an authenticator, can be used for authentication, rather than using the knowledge of a piece of data, i.e., passwords.

2.1.1 Overview of the OFFPAD

The Offline Personal Authentication Device (OFFPAD) is a device developed by the Norwegian cyber security startup PONE Biometrics. It is a credit card-sized device with a fingerprint reader that enables its users to authenticate against online services by virtue of the FIDO2 standard being implemented on the device [Pon]. The OFFPAD and FIDO2 are two separate entities and are being developed independently of each other, but combining these two is what enables biometric authentication against online services. This effectively replaces passwords, and the user does not have to enter a string of characters to authenticate, but rather press their fingertip against the device's fingerprint reader. This is achieved by the use of PKC and so called passkeys. The first time a user wants to authenticate against a service using the the OFFPAD, they must first register the device with the service. On a high level, this means that the device generates a key pair and sends the public key to the online

service which stores it, while the private key is stored locally. When the user wants to authenticate, the online service sends a challenge derived from a random nonce and the context, i.e., the user and the service. The user then authenticates locally with his fingerprint or with a PIN used as backup, which unlocks the corresponding private key used to sign the challenge and authenticate the user. This authentication mechanism is enabled by the client and server implementing their part of the FIDO2 standard [SH23].

2.1.2 Passkeys vs. Passwords

The OFFPAD functions as a replacement for passwords and Multi-Factor Authentication (MFA) by using passkeys, which are key pairs of a single private key and a single public key. The public key is stored on the online service's end, and the private key is stored locally on the device. The combination of these is used for authentication, and the process is explained in more detail in Subsection 2.2.1 on FIDO2.

The use of passkeys for authentication has several advantages over passwords, which is the authentication mechanism that people are most familiar with. The fundamental premise of passwords is to have a shared secret between the user and the service against which it wishes to authenticate. Due to this, we say that this authentication mechanism is knowledge-based. The user enters the shared secret, and the service checks whether the entered credential matches the one it has stored in its database for that particular user. Ideally, this shared secret possess a high degree of randomness such that it is difficult to brute-force a user's credentials, but it should also be unique for each service to mitigate the consequences of a potential database breach on a service. The combination of unpredictability and uniqueness of passwords imposes large expectations on a user who should remember many different complicated credentials.

Authentication using passkeys on the other hand, is possession-based [GSN+20], that is, the user has to be in possession of a physical device to authenticate. For the OFFPAD, this is a combination of the device itself, and a valid fingerprint or PIN. Such a PIN is stored locally on the device, is not connected to any external service, and is only used to gain access to passkey credentials. One important benefit of passkeys is therefore less risk exposure in case a service's database(s) are breached. In a knowledge-based authentication system, an adversary may gain access to passwords. However, in possession-based authentication systems utilizing passkeys, the adversary may gain access to public keys, which are rendered useless without the corresponding private key, which is stored locally on the device. In addition, passkeys are less exposed to phishing attacks; the most common form of cyber crime [GTJA17], in which an adversary will impersonate a reputable actor in order to get the user to

enter their credentials for what the user thinks is a trustworthy actor, e.g. by spoofing a web site. The resistance to phishing is achieved by the fact that passkeys only work on the websites they are registered with, making it so a user can not be tricked into authenticating on a deceptive site [Goo23]. Even if the user does sign a challenge coming from a fraudulent actor, this does not reveal anything of value, e.g., a secret key, to the attacker.

In recent years, there has been an effort to add an additional layer of security on top of passwords to compensate for their vulnerabilities, this is known as MFA. Smartphones are typically used to provide the second factor. However, this involves extra work for the user who must perform an extra step to authenticate, as well as having the device offering the second factor with them at the time of authenticating. Studies have shown users find this an inconvenience [SSKC14]. It has also resulted in the phenomenon known as MFA fatigue, which opens up for social engineering attacks. Due to the many MFA notifications a user may receive, the attacker hopes that you do not pay as much attention to the ones prompted by him so that you will enter the credentials necessary to authenticate against an otherwise secure environment [Tak22]. The OFFPAD, utilizing passkeys, is designed to provide local authentication making it MFA on its own, effectively eliminating the attack vector posed by MFA fatigue [KMWK23].

In addition to the security differences between passwords and passkeys, there are different degrees of security within passkeys themselves. On the OFFPAD, the passkeys are always bound to the device and never leave. In other implementations, the passkeys may be synced between a user's different devices. One example of this is Google Password Manager, in which a user's generated passkeys may be synced between their Android devices signed in to the same Google account [Goo23]. Apple is doing something similar where a user's passkeys may be synced between their Apple devices using the iCloud keychain. This type of passkey syncing can be used by an administrator to manage which passkeys are synced to which devices, and this can be beneficial in a business setting where one might want to manage the authorization levels between different roles [App23]. However, this syncing exposes a security vulnerability in which passkeys may be intercepted if proper encryption and security measures are not put in place. This particular vulnerability is absent on the OFFPAD by virtue of the passkeys always being bound to the device.

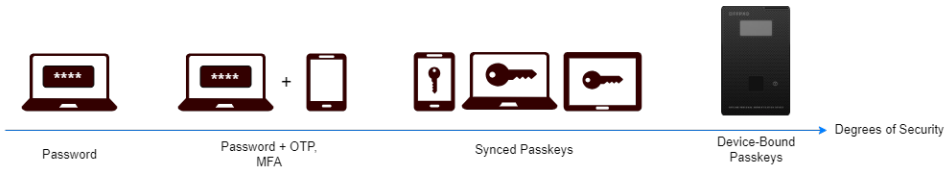


Figure 2.1: Degrees of security for different authentication methods [SH23].

2.2 FIDO2

With the reader armed with the knowledge of passkeys, it is now time to introduce the protocols involved to make these a reality and how authentication using them works in practice.

2.2.1 Overview of FIDO2

FIDO2 is an open standard based on PKC developed by the FIDO Alliance [FID23b] to pave the way for passwordless authentication using passkeys. FIDO2 is an umbrella term that encompasses specifications for W3C's Web Authentication (WebAuthn) and Client to Authenticator Protocol (CTAP), which together enable passwordless authentication [FID23a]. The two primary areas in which it offers benefits are in security and convenience; security because only the public key leaves the device which provides phishing resistance, and convenience because of its possession-based nature which lifts the burden of knowledge-based credentials of the user. FIDO2 is device independent and allows users to leverage common devices, such as the OFFPAD, to authenticate against online services. Another example of a device form using FIDO2 for authentication are USB-sticks, as being developed by Yubico [Yub23].

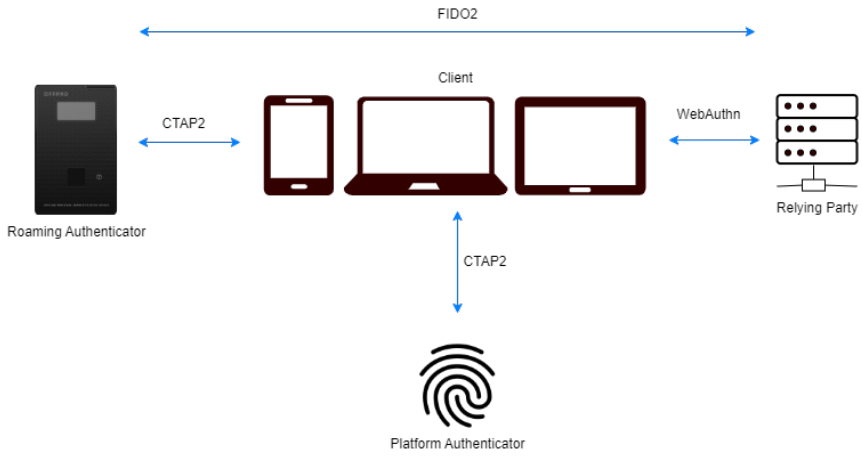


Figure 2.2: Components of the FIDO2 standard and the communication protocols involved [SH23].

2.2.2 Terminology

To better understand the workings of FIDO2, we first introduce some terminology established by the FIDO Alliance for the different components involved. FIDO2 authentication is built around the two sub-protocols WebAuthn and CTAP, and involves the different parties listed below.

- **Relying Party (RP).** The online service the user wishes to authenticate against. It registers and authenticates users by storing the public key created by the user and issuing a challenge for the user to sign [Wor22].

- **Client.** An intermediary party (e.g., a web browser) between the authenticator and the RP that is responsible for the communication between them [Wor22].

- **Authenticator.** A device capable of cryptographic operations that can register a user with a RP and later assert possession of the private key associated with a public key stored by the RP. This is done by the authenticator signing the challenge issued by the RP with the private key corresponding to that public key. We distinguish between *roaming authenticators* and *platform authenticators*. A roaming authenticator, e.g., the OFFPAD, can be used with different client devices, i.e., the hardware device on which the client runs. Platform authenticators, on the other hand, are bound to a specific client device; think of it as a "trusted device" [Wor22].

2.2.3 CTAP

CTAP allows the client to communicate with the authenticator, which enables communication between the authenticator and RP. FIDO2 uses CTAP2, a slight modification of CTAP1, but the term CTAP will hereafter be agnostic as to whether it denotes CTAP1 or CTAP2. It is an application layer protocol and is responsible for setting up a secure communication channel between the client and the authenticator. Without it, any application may try to request a challenge response from the authenticator. For security reasons, it is desired that access to the authenticator's Application Programming Interface (API) is restricted only to the intended applications [FID18; BBCW21].

2.2.4 WebAuthn

WebAuthn is an API developed by the World Wide Web Consortium (W3C) and the FIDO Alliance and is what allows RPs to register and authenticate users using PKC instead of passwords. Upon registering a device with a RP, the RP must provide data that binds the user to a passkey that will be generated. These data include identifiers for the user and the RP. This is so that each of the user's passkeys is associated with a single RP. The RP will then call the WebAuthn API to prompt the user to generate a passkey, and send the public key to the RP who will store it [Web23]. This process, as well as the authentication process, is described in more detail in Section 2.2.5 and Section 2.2.6 below.

2.2.5 Registration Procedure

The registration is the process in which the user, the user's client (with access to at least one authenticator), and the RP work together to create a passkey credential and associate it with the user's RP account. This (typically) involves a test of user verification which must be passed in order to complete the process. It is passed by authenticating locally on the authenticator [Wor22], which, in the context of the OFFPAD, involves scanning your fingerprint or entering a valid PIN. The steps below give an overview of the registration flow.

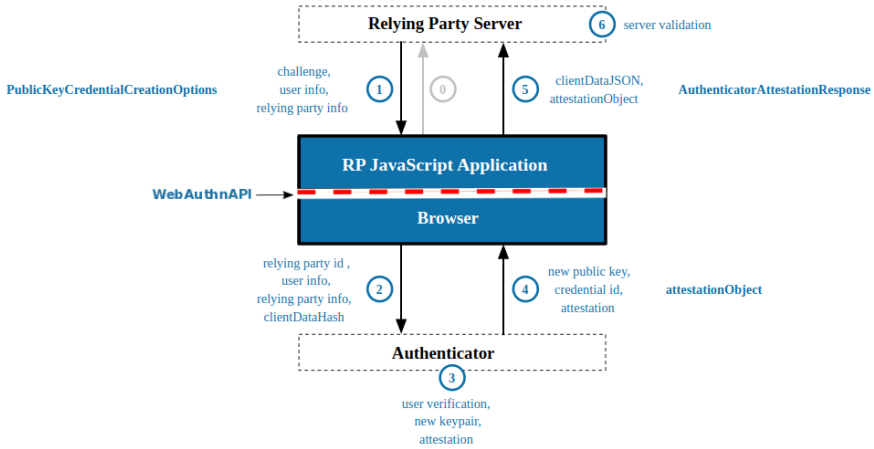


Figure 2.3: WebAuthn registration flow [Wor22].

0. A user enters a username, after which the client initiates a request to register an authenticator on behalf of the user.

1. The RP creates an instance of `PublicKeyCredentialCreationOptions` and returns it to the client. It contains information about the user, the RP and the type of credential desired.

2. The client application searches for and locates the authenticator. It then forwards the user information, RP information, and `clientDataHash` which is a hash of the serialized client data, including the challenge, to the authenticator.

3. User verification is done by having the user authenticate locally on the device. This effectively gives the authenticator permission to generate a new passkey and credential ID, as well as attestation data.

4. An `attestationObject` containing the generated public key, credential ID, and attestation data, is sent to the client application.

5. The public key and credential data are put in a `ClientDataJSON` object, and together with `attestationObject`, is sent to the RP.

6. After receiving the data, the RP performs a series of validation checks. If successful, RP securely stores the public key, associating it with the user and the authentication characteristics from the provided attestation data. This enables subsequent authentication using the private key associated with the public key sent to the RP [Wor22; Yub24b].

2.2.6 Authentication Procedure

The authentication process serves to authenticate the user against the RP and can only be performed after the user has undergone registration. The user and the client (with access to at least one authenticator) work together to prove to the RP that the user possesses the private key corresponding to a previously registered public key. The only actions necessary from the user are to provide its username and to authenticate locally.

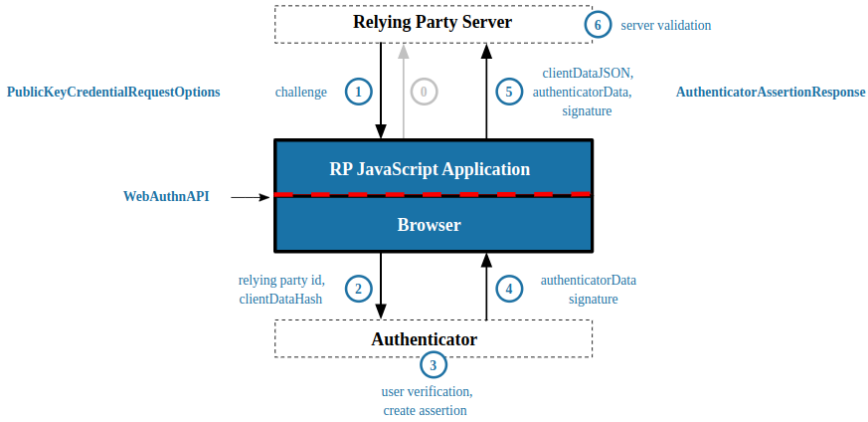


Figure 2.4: WebAuthn authentication flow [Wor22].

0. User enters username and requests to authenticate, after which the client initiates a request to authenticate on behalf of the user.

1. The RP creates an instance of `PublicKeyCredentialRequestOptions` and returns it to the client. It contains the fields `challenge` and `allowCredentials`. The first is a random value generated by the RP that will be signed, the latter contains a list of previously registered credentials that may be used to perform the authentication.

2. The client hashes the client data into `clientDataHash` and sends it to the authenticator along with the RP ID.

3. The authenticator finds the credential matching the RP ID and prompts the user to authenticate locally. The authenticator then creates an assertion by signing `authenticatorData` | `clientDataHash` with the private key generated during account during registration.

4. The authenticator returns `authenticatorData` and signature to the client.

5. The client forwards the `authenticatorData` and the signature to the RP, together with `clientDataJSON` which contains JavaScript Object Notation (JSON)-serialized data passed to the authenticator by the client to generate the credential.

6. The RP conducts a series of verification checks, including a verification of the received signature. If all verification checks are passed, the user is successfully authenticated [Wor22; Yub24a].

2.3 Digital Signatures and Dilithium

The reader will now be introduced to the concept of a digital signature, that is, how pencils and ink are swapped out for zeroes and ones. Then, the PQC digital signature algorithm Dilithium is explained in more detail.

2.3.1 Introduction to Digital Signatures

A digital signature is a cryptographic primitive, serving as an electronic equivalent to a written signature, and is used to provide the assurance that the claimed signatory signed the message. In addition, it also provides data integrity, i.e., assurance that the data have not been tampered with after signing, as well as non-repudiation, i.e., assurance that an entity cannot deny previously signed data [CMRR23; JMV01]. Digital signature schemes commonly see use in cryptographic protocols that provides various services including entity authentication, as seen in the FIDO2 standard [FID15].

Digital signature algorithms include two main processes, namely signature generation and signature verification. A signatory will generate a signature on the data, and the verifier will verify the authenticity of the signature. Digital signatures build on PKC, and each signatory has a private and public key and is the owner of that key pair. A signatory generates a signature using its private key and the verifier checks if the signature is valid with the corresponding public key. If validated, it proves that the message was signed by the claimed party and that the data has not been tampered with [CMRR23].

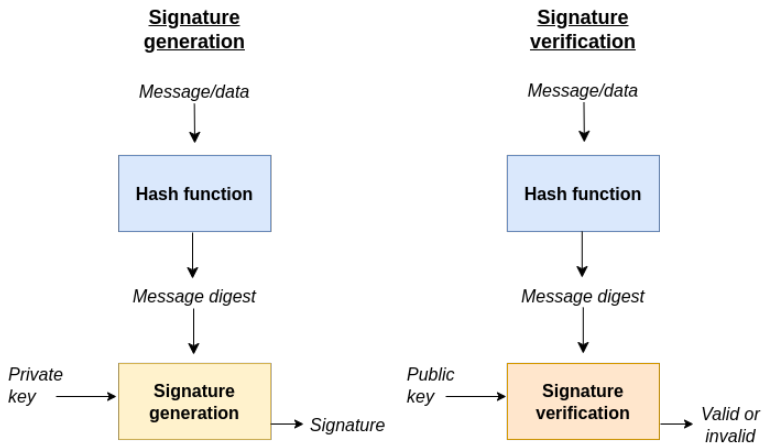


Figure 2.5: High-level overview of signature generation and verification [CMRR23].

Definition 2.1. (Digital signature scheme) A digital signature scheme Σ consists of three algorithms $(\Sigma.\texttt{KeyGen}, \Sigma.\texttt{Sign}, \Sigma.\texttt{Verify})$, with message space $m \in M$ and signature space $\sigma \in S$:

- $\Sigma.\texttt{KeyGen}()$ is a probabilistic key generation algorithm that returns a secret signing key sk and public verification key pk.
- $\Sigma.\texttt{Sign}(m, \mathsf{sk})$ is a probabilistic signature generation algorithm that takes a message $m \in M$ and a secret key sk as inputs and returns a signature $\sigma \in S$.
- $\Sigma.\texttt{Verify}(m, \sigma, \mathsf{pk})$ is a deterministic verification algorithm that takes a message $m \in M$, a signature $\sigma \in S$ and a public key pk as inputs. It returns `true` or `false`. If it returns `true`, we say that the algorithm accepts, otherwise we say that the algorithm rejects the signature σ on the message m.

It is required that $\Sigma.\texttt{Verify}(m, \mathsf{pk}, \Sigma.\texttt{Sign}(m, \mathsf{sk})) = 1$ for any honestly generated pair $\Sigma.\texttt{KeyGen}() \rightarrow (\mathsf{sk}, \mathsf{pk})$.

We say that a digital signature scheme is secure if it is existentially unforgeable against chosen-message attacks (EUF-CMA)[1] [BH23].

[1] If an attacker, given the ability to obtain signatures on as many chosen messages as they want, can then produce a new message (not seen before) and generate a valid signature for it that verifies under the public key, then the digital signature scheme is not EUF-CMA secure.

2.3.2 Lattice Cryptography and Short Integer Solution

The Dilithium digital signature algorithm is based on lattice cryptography, a branch of cryptography that involves the use of lattices. A single lattice is a set of points in n-dimensional space with a periodic structure. More formally, given n linearly independent vectors $\boldsymbol{b_1}, ..., \boldsymbol{b_n} \in \mathbb{R}^n$, the lattice generated by them is the set of all integer linear combinations of the basis vectors, as given by

$$L(\boldsymbol{b_1}, \ldots, \boldsymbol{b_n}) = \left\{ \sum_{i=1}^{n} x_i \boldsymbol{b_i} : x_i \in \mathbb{Z} \right\}$$

The vectors $\boldsymbol{b_1}, \ldots, \boldsymbol{b_n}$ make up the *basis* of the lattice. The figure below illustrates the same two-dimensional lattice generated by two different bases.

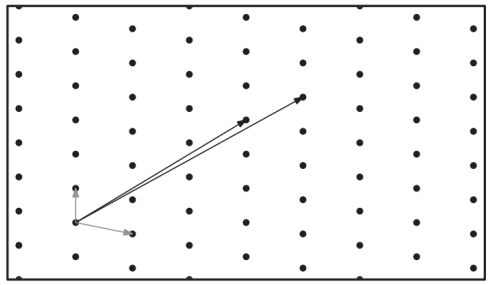


Figure 2.6: Two-dimensional lattice with two possible bases [MR09].

What makes lattices interesting for PQC is that mathematical problems can be constructed from them that are thought to be computationally difficult even for quantum computers [MR09]. One of these problems is the Short Integer Solution (SIS) problem, which is what the Dilithium algorithm is based on. Informally, SIS asks, given many uniformly random elements of a large finite additive group, find a sufficiently "short" non-trivial integer combination of these elements such that they sum to zero. The problem is parameterized by positive integers n and q that define the group $\mathbb{Z}_q^n$, a positive real $0 < \beta_{SIS} < q$, and a number m of group elements.

Definition 2.2. (Short Integer Solution ($\mathrm{SIS}_{n,q,\beta_{SIS},m}$)) Given m uniformly random vectors $a_i \in \mathbb{Z}_q^n$, forming the columns of a matrix $\mathbf{A} \in \mathbb{Z}_q^{n \times m}$, find a non-zero integer vector $\mathbf{z} \in \mathbb{Z}^m$ of norm $0 < \|\mathbf{z}\| \leq \beta_{SIS}$ such that

$$f_{\mathbf{A}}(\mathbf{z}) := \mathbf{A}\mathbf{z} = \sum_i a_i \cdot z_i = 0 \in \mathbb{Z}_q^n$$

Two constraints are worth pointing out. First, without the constraint on $\|\mathbf{z}\|$, Gaussian elimination could be used to find a solution. The problem states that it

is hard to find *short* such solutions. Secondly, it is necessary for $0 < \beta_{SIS} < q$ for SIS to be hard, otherwise $\mathbf{z} = (q, 0, \ldots, 0) \in \mathbb{Z}^m$ would be a valid, although trivial, solution.

A drawback with schemes based on traditional SIS described above is that they tend not to be efficient enough for practical application. An approach used to avoid this inefficiency is to use lattices that possess extra algebraic structure [LPR10]. This is why Dilitihum uses a variant of SIS called module-SIS with polynomials from R_q as entries, rather than integers from $\mathbb{Z}_q$. The definition is the same as above except with R_q rather than $\mathbb{Z}_q$. If the ring R_q has dimension d (a power of two), and the matrix has dimension n' over R_q, then the lattice will have dimension $n' \times d$. However, a matrix with dimension n over $\mathbb{Z}_q$ will have a lattice of dimension n. Computing over R_q is more efficient and takes time $O(n'd \log d)$ rather than $O(n^2)$, and an element in R_q can be represented by d elements but still be equivalent to a matrix of dimensions $d \times d$, effectively saving a factor d. It should be noted that $n' \times d$ should be close to, but never less than, n. Also, for Dilithium, $d = 256$ and $n' \in \{3, 4, 5\}$. The security level is only dependent on the total dimension of the matrix, regardless if it is over R_q or $\mathbb{Z}_q$. This can be exemplified by the tool *lattice-estimator* on Github [Alb24; APS15].

2.3.3 The Dilithium Algorithm

The design of the Dilithium scheme prioritized four key criteria [BDK+21], with simplicity in secure implementation concerning randomness and constant-time operations being the first, given the impracticality of expecting expert-level implementations universally. Secondly, the Dilithium scheme employed conservative parameters, considering long-term security and analyzing the applicability of lattice attacks from a perspective favorable to potential attackers. Thirdly, the total size of public key plus signature had to be minimized as many applications require transmission of both of these. Finally, the scheme had to be modular so that it was easy to vary the security.

Scheme Walkthrough

We present a high-level overview of how Dilithium works in practice. We first present the zero-knowledge proof system on which Dilithium signatures are built. The prover wants to convince the verifier that he knows short values $\mathbf{s_1}$ and $\mathbf{s_2}$ that satisfy the public relation. The available public information is a uniform matrix $\mathbf{A}$, and the value $\mathbf{t} = \mathbf{As_1} + \mathbf{s_2}$. The first step of the protocol consists of the prover sampling bounded masking values $\mathbf{y_1}$ and $\mathbf{y_2}$ and sending a commitment $\omega = \mathcal{H}(\mathbf{Ay_1} + \mathbf{y_2})$ to the verifier. The second step is for the verifier to send a challenge c to the prover. The final step consists of the prover adding the mask to the product of the challenge and the secret and sending the response if $\mathbf{z_1}$ and $\mathbf{z_2}$ are appropriate. This last step is known as rejection sampling and its purpose in this case is to output $\mathbf{z_1}$ and $\mathbf{z_2}$

values that do not leak the secrets $\mathbf{s_1}$ and $\mathbf{s_2}$. If $(\mathbf{z_1}, \mathbf{z_2}) :=$ False, the protocol will restart [Lyu20].

The prover must prove that he knows $\mathbf{s_1}$ and $\mathbf{s_2}$ that satisfy $\mathbf{As_1} + \mathbf{s_2} = \mathbf{t}$, where the coefficients of $\mathbf{s_1}$ and $\mathbf{s_2}$ fall into a particular range (ideally $[\beta]$, but $[\bar{\beta}]$ for some $\bar{\beta}$ a little larger than β is also valid) [Lyu20]. The following is the basic ZKPoK system on which Dilithium is built.

Private information: $\mathbf{s_1} \in [\beta]^m, \mathbf{s_2} \in [\beta]^n$
Public information: $\mathbf{A} \in R_q^{n \times m}, \mathbf{t} = \mathbf{As_1} + \mathbf{s_2} \in R_q^n$

Prover		Verifier
$\mathbf{y_1} \leftarrow [\gamma + \bar{\beta}]^m$		
$\mathbf{y_2} \leftarrow [\gamma + \bar{\beta}]^n$		
$\omega := \mathcal{H}(\mathbf{Ay_1} + \mathbf{y_2})$		
	$\xrightarrow{\omega}$	
		$c \leftarrow \mathcal{C}$
	$\xleftarrow{c}$	
$\mathbf{z_1} := c\mathbf{s_1} + \mathbf{y_1}$		
$\mathbf{z_2} := c\mathbf{s_2} + \mathbf{y_2}$		
if $\|\mathbf{z_1}\|_\infty \notin [\bar{\beta}]^m$ or $\|\mathbf{z_2}\|_\infty \notin [\bar{\beta}]^n$: then $(\mathbf{z_2}, \mathbf{z_2}) :=$ `False`		
	$\xrightarrow{(\mathbf{z_1}, \mathbf{z_2})}$	
		Accept iff $\|\mathbf{z_1}\|_\infty \in [\bar{\beta}]^m$ and $\|\mathbf{z_2}\|_\infty \in [\bar{\beta}]^n$ and $\mathcal{H}(\mathbf{Az_1} + \mathbf{z_2} - c\mathbf{t}) = \omega$

Figure 2.7: The basic ZKPoK system on which Dilithium is built. The value of $\bar{\beta}$ affects the probability that `False` is not sent [Lyu20].

The transformation of this ZKPoK system into a digital signature scheme is done through a Fiat-Shamir transform [FS86], where rather than the challenge being generated by the verifier, the challenge is created as a hash of the message to be signed and the first message ω of the prover, along with the public key. We will now look at a simplified version of how the three operations key generation, signature generation, and signature verification are performed.

$\underline{\texttt{Gen}}$

1: $\mathbf{A} \leftarrow\$ R_q^{n \times m}$
2: $\mathbf{s_1} \leftarrow\$ [\beta]^m$
3: $\mathbf{s_2} \leftarrow\$ [\beta]^n$
4: $\mathbf{t} := \mathbf{A}\mathbf{s_1} + \mathbf{s_2}$
5: **return** $(\mathsf{pk} = (\mathbf{A}, \mathbf{t}), \mathsf{sk} = (\mathbf{A}, \mathbf{t}, \mathbf{s_1}, \mathbf{s_2}))$

$\underline{\texttt{Sign}(\mathsf{sk}, M)}$

1: $\mathbf{z} := \texttt{False}$
2: **while** $\mathbf{z} := \texttt{False}$ **do**
3: $\quad \mathbf{y_1} \leftarrow\$ [\gamma + \bar{\beta}]^m$
4: $\quad \mathbf{y_2} \leftarrow\$ [\gamma + \bar{\beta}]^n$
5: $\quad c := \mathcal{H}(M, \mathbf{A}\mathbf{y_1} + \mathbf{y_2}, \mathsf{pk})$
6: $\quad \mathbf{z_1} := c\mathbf{s_1} + \mathbf{y_1}$
7: $\quad \mathbf{z_2} := c\mathbf{s_2} + \mathbf{y_2}$
8: $\quad$ **if** $\|\mathbf{z_1}\|_\infty \notin [\bar{\beta}]$ **or** $\|\mathbf{z_2}\|_\infty \notin [\bar{\beta}]$:
9: $\quad\quad \mathbf{z} := \texttt{False}$
10: **return** $\sigma = (\mathbf{z_1}, \mathbf{z_2}, c)$

$\underline{\texttt{Verify}(\mathsf{pk}, M, \sigma = (\mathbf{z_1}, \mathbf{z_2}, c))}$

1: **if** $\|\mathbf{z_1}\|_\infty \in [\bar{\beta}]$ **and** $\|\mathbf{z_2}\|_\infty \in [\bar{\beta}]$ **and** $c = \mathcal{H}(M, \mathbf{A}\mathbf{z} - c\mathbf{t}, \mathsf{pk})$:
2: $\quad$ **return** `True`
3: **return** `False`

Figure 2.8: Simplifed template [BDK+21] of the Dilithium signature algorithm, after performing a Fiat-Shamir transform of the ZKPoK system presented in Figure 2.7.

Key Generation. The key generation algorithm generates a $k \times \ell$ matrix $\mathbf{A}$ with polynomials in the ring $R_q = \mathbb{Z}_q[X]/(X^n + 1)$ as entries. Afterwards, secret vectors $\mathbf{s_1}$ and $\mathbf{s_2}$ are sampled. These are secret for anyone but the signer. Finally, the second part of the public key is calculated as $\mathbf{t} = \mathbf{A}\mathbf{s_1} + \mathbf{s_2}$. All algebraic operations in this scheme are assumed to be over the polynomial ring R_q.

Signature Generation. The signing algorithm begins with generating a masking vectors $\mathbf{y_1}$ and $\mathbf{y_2}$. Afterwards, the challenge c is generated, after which the masking vectors are added to the product of the challenge and the secret. A rejection sampling step is then performed to keep the secret values from leaking. If the $\mathbf{z}$-values do not meet the necessary criteria, the protocol is restarted.

Verification. The verifier accepts the signature as valid if $\|\mathbf{z_2}\|_\infty, \|\mathbf{z_2}\|_\infty \in [\bar{\beta}]$ and c is the hash of $M, \mathbf{A}\mathbf{z} - c\mathbf{t}$, and pk.

In the full scheme, computational efficiency is enhanced by utilizing either the higher or lower bits of specific elements.

Key and Signature Sizes of Dilithium

Dilithium can be instantiated for three different NIST security levels, namely levels 2, 3, and 5, where security level 5 offers the highest security. This naturally results in different key and signature sizes, as illustrated in the table below.

NIST Security Level	**2**	**3**	**5**
Output Size (Bytes)			
Public Key	1312	1952	2592
Secret Key	2528	4000	4864
Signature	2420	3293	4595

Table 2.1: Key and signature sizes for Dilithium instantiated with different NIST security levels [BDK+21; Ope].

For the remainder of this work, “Dilithium-2” will refer to the Dilithium algorithm with security level 2, while “Dilithium” will refer to the algorithm in general.

Chapter 3
Methodology

This chapter provides an overview of the implementation process in this work, detailing the steps taken to achieve a PoC PQC-capable FIDO2 authenticator. Furthermore, the chapter presents the tools and resources utilized in the project, including the rationale behind these choices. These tools and resources encompass hardware, programming language, testing environments, and libraries. Lastly, the performance metrics used to evaluate the implementation and the measurement method are presented.

3.1 Overview

This work aims to show how a quantum-secure digital signature algorithm like Dilithium can be used in FIDO2 authentication to provide quantum-secure authentication. The task then becomes implementing an authenticator with PQC-capability on a development board. It is not enough that the development board (client side) can create (and sign with) quantum-secure cryptographic primitives, but it must also be supported by the RP (server side). A FIDO2 testing server already developed by PONE Biometrics will be used as the RP, but will have to be modified to support Dilithium. This is made possible by implementing a relevant PQC-library. The majority of the implementation effort lies in implementing CTAP and PQC-capabilities.

The most important FIDO2 functionality to implement on the development board are the CTAP-commands `authenticatorGetInfo`, `authenticatorMakeCredential`, and `authenticatorGetAssertion`. The latter two will initially be implemented to support an already established cryptographic algorithm, specifically ECDSA. Only once authentication with ECDSA is working will support for Dilithium-2 be built in. With Dilithium support built in, the whole authentication flow should be quantum-secure, at which point performance testing will be done. Key performance metrics include key generation time, signing time, and signature sizes. This will be performed for ECDSA, Dilithium-2, as well as for a hybrid construction of the two. In addition

to Dilithium-2, another PQC algorithm being standardized, Falcon, will also be benchmarked to provide further context for discussion. The Falcon algorithm can be instantiated for NIST security levels 1 and 5, and this work utilizes Falcon-512, which corresponds to NIST security level 1. From here on, "Falcon-512" refers to the Falcon algorithm with NIST security level 1, while "Falcon" refers to the algorithm in general.

3.2 Implementation Steps

Below follows a more detailed review of the necessary implementation steps. Note that it will only contain what are deemed the most critical steps, and will by no means be an exhaustive list.

1. Set up a functioning development environment that contains the necessary tools to get started. Zephyr's own *Getting started guide* [Zep24a] is followed for this. To confirm that the environment is set up correctly, we flash (upload) a simple program to the board and see if it works as expected. Zephyr comes with a number of sample programs that demonstrate basic functionality.

2. Create functionality for the board to read and output data over serial. This will serve as the foundation for receiving CTAP-commands from the client (browser), and transmitting the appropriate response back to the client, for it to forward to the server.

3. Implement `getAuthenticatorInfo`. Upon receiving an `authenticatorGetInfo` request, the board should return a static response containing Concise Binary Object Representation (CBOR)-encoded data with information of the device. This information includes, among other parameters, which algorithms it supports for authentication.

4. Implement `authenticatorMakeCredential` for registration. Upon receiving an `authenticatorMakeCredential` request, the board should create a credential and create an attestation object as a response. The attestation object has three fields; `authData` (authenticator data), `fmt` (attestation statement format identifier), and `attStmt` (attestation statement).

5. Implement `authenticatorGetAssertion` for authentication. Upon receiving an `authenticatorGetAssertion` request, the board should generate a signature with the credential generated in the previous step. The response includes two fields; `authData` and `signature`.

3.3 Tools and Resources

This section outlines the tools and resources utilized in developing the authenticator. From the STM32 Nucleo-64 development board and the Zephyr Real-Time Operating System (RTOS) to software libraries like *mbedtls* and *liboqs*, each component is instrumental in realizing the project's objectives.

3.3.1 STM32 Nucleo-64 Development Board

Acting as the authenticator will be the STM32 Nucleo-64 development board with the STM32L476RG Microcontroller Unit (MCU) based on the 32-bit Arm Cortex M4 processor. The board and the OFFPAD share similar hardware, most importantly the Arm Cortex M4 processor. Since the project aims at being of relevance to the OFFPAD, it made sense to pick hardware with similar specifications.

3.3.2 Zephyr OS

As its Operating System (OS), the OFFPAD uses Zephyr OS [Zep24b]. It is an open source RTOS designed for use on embedded and resource-constrained systems and is developed by the Linux Foundation [Lin24]. Choosing the same OS for the Nucleo-L476RG as that running on the OFFPAD adds another layer of similarity between the two devices, so that the results translate better into a potential OFFPAD implementation. The OS's high configurability allows seamless integration with third-party libraries relevant to the project. Notably, Zephyr OS comes equipped with many built-in features, including cryptography modules and specific encoding schemes, further contributing to its suitability for the project. In addition to the reasons mentioned, it also comes with comprehensive documentation available on the Zephyr Project website.

3.3.3 C Programming Language

The C programming language is the de facto standard for embedded development. Many embedded systems rely on it due to its low-level nature, which offers a high degree of direct hardware control [Bar99]. Given that Zephyr OS supports both C and its closely related counterpart, C++, opting for C as the primary language for this project is a natural choice. Moreover, when seeking assistance or information online for embedded system-related queries, C is the predominant context, making it a favorable and practical choice.

3.3.4 FIDO2 Testing Server

PONE Biometrics has created their own FIDO2 testing server written in Java implementing WebAuthn. This will act as the RP. An already developed server

that can handle the FIDO2 authentication flow is sufficient for this project, as long as it can be edited to support Dilithium signatures. For this reason, I was given access to the PONE Biometrics repository with the FIDO2 testing server. There are many FIDO2 test server implementations out there, but given my direct dialogue with the people at PONE Biometrics, it made sense to go with theirs for effective collaboration.

3.3.5 Libraries

To perform CTAP encoding and decoding, as well as cryptographic operations, both classical and PQC ones, the following libraries were used.

zcbor

CTAP messages are encoded in CBOR, and so a tool was necessary to perform encoding and decoding. For this task, the open source library *zcbor* [Nor24] is used. It comes preinstalled on Zephyr and is the default CBOR-encoding tool in Zephyr, specifically tailored for use in microcontrollers. It can be used to generate C code for encoding or decoding of CBOR.

mbedtls

For classic cryptographic primitives, the open source library *mbedtls* [Mbe24] is used. Its small code footprint makes it suitable for embedded systems, and it also comes preinstalled on Zephyr. Some of its functionality used in this project is its SHA-256 and ECDSA implementations.

liboqs

For PQC functionality, the open source library *liboqs* [Ope24] is used and provides the Dilithium and Falcon algorithms. The library is part of the Open Quantum Safe (OQS) project, which aims to facilitate prototyping with quantum-secure algorithms to make migration to PQC easier. OQS is supported by the PQC Alliance as part of the Linux Foundation. The library's contributors constitute individuals, academics and researchers, as well as companies including Amazon Web Services, Microsoft Research, and IBM Research. It can be used as a Zephyr module, making its integration into this project simple.

3.4 Performance Measuring

To evaluate the performance of the algorithms implemented, several benchmarks will be conducted. These benchmarks will measure key aspects of cryptographic operations, focusing on the following metrics:

- **Key generation time.** The time (in ms) taken to generate a cryptographic key pair.
- **Signing time.** The time (in ms) taken to sign a message.
- **Public key size.** The size (in bytes) of the public key.
- **Signature size.** The size (in bytes) of the generated signature.

For the timing benchmarks, the standard deviation, a measure of the amount of variation around the mean of a set of values, will be calculated. This helps illustrate the significance of Dilithium's rejection sampling in its performance.

The performance of the following algorithms will be assessed:

- **ECDSA.** The classical ECDSA algorithm, using the P-256 curve along with SHA256. It goes under the name ES256 in the IANA COSE Algorithm Registry [Int24].
- **Dilithium-2.** The Dilithium algorithm instantiated with NIST security level 2.
- **Falcon-512.** Another PQC algorithm [FHK+18] being standardized by NIST. Instantiated with NIST security level 1.
- **Hybrid ECDSA with Dilitihum-2.** ECDSA used in combination with Dilithium-2.
- **Hybrid ECDSA with Falcon-512.** ECDSA used in combination with Falcon-512.

In this work, Zephyr's system clock, as part of its kernel services, is used to measure an operation's time. This is done by looking at the system clock once before, and once after, an operation is performed. The total time taken by the operation is the difference between these two measurements. The function used is `sys_clock_tick_get()` which returns the system clock in terms of ticks. For the Nucleo-LG476RG board, `CONFIG_SYS_CLOCK_TICKS_PER_SEC` is set to 10000. To convert from ticks to milliseconds, one can then do $\frac{ticks \times 1000}{10000}$. The testing code is run in the same place as the operation would be performed regularly, with the exception that the operation is put inside a loop such that 50 iterations of the operation can be performed, which should result in a representative mean value.

```
1 int num_runs = 50;
2 for (int i = 0; i < num_runs; i++) {
3     int64_t start_tick, end_tick;
```

```
4       start_tick = sys_clock_tick_get();
5
6       /* Operation to benchmark here */
7
8       end_tick = sys_clock_tick_get();
9       tick_diff = end_tick - start_tick;
10      printk("Iteration: %d, ticks: %lld\n", i, tick_diff);
11  }
```

Listing 3.1: Code used for benchmarking.

Chapter 4
Proposed Solution

This chapter delves into the implementation of an authenticator capable of quantum-secure FIDO2 authentication. Since the FIDO2 standard doseparately.support PQC algorithms, achieving this required building the capabilities of the authenticator from scratch. As a result, a significant portion of this section is dedicated to detailing the implementation of CTAP on the Nucleo-L476RG development board. In this context, *authenticator* refers to the development board serving as an authenticator due to the implementation of CTAP.

4.1 Requirements

There are some concrete requirements of the implementation for it to function as a PoC for quantum-secure FIDO2 authentication. These requirements define the behavior and features that the system must possess.

ID	Requirement
FR1	The authenticator should be able to register a new credential with the test server (RP), and use it to authenticate.
FR2	The authenticator (and test server) should support hybrid credentials, i.e., ECDSA combined with Dilithium-2.

4.2 Architecture

Although the majority of the implementation work during this project has been on making the development board act as an authenticator capable of the above requirements, several other components are involved in making a complete FIDO2 authentication flow. The components involved for the complete testing system are the authenticator, the client (browser), the test server (RP), and a database for the test server to store registered users.

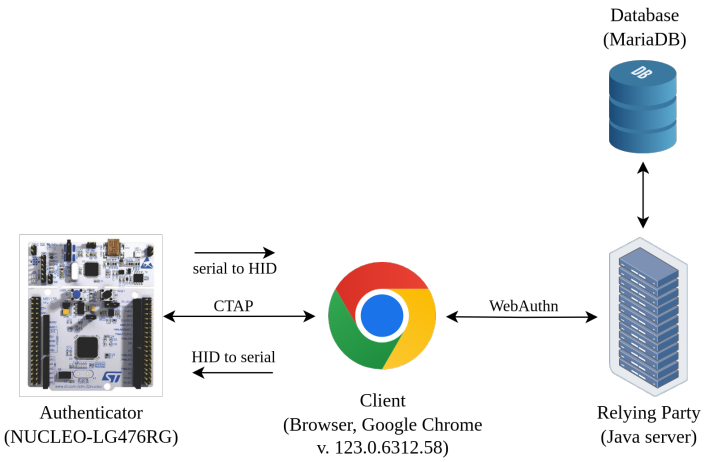


Figure 4.1: Overview of the components involved in the development environment to perform a full FIDO2 authentication flow.

The primary focus of the implementation lies in CTAP, encompassing the communication between the authenticator and the client. Supplementary efforts are directed towards the RP side, specifically to integrate support for the Dilithium-2 algorithm, a process conducted closely with PONE Biometrics. Additionally, the RP establishes connectivity to a MariaDB database for storing registered users. All components operate in a local development environment running on a Lenovo ThinkPad X1 Yoga (1st Gen), equipped with a Core i7 processor and 16 GB of RAM.

4.3 Transmission and Reception of Data

This section focuses on how data is received and transmitted between the PC and the board. It should be noted that CTAP messages are framed for USB transport using the Human Interface Device (HID) protocol [FID18]. Implementing this protocol from scratch is beyond the scope of this project, which is why an external Python script is used to translate serial to HID, and vice versa, effectively creating a virtual USB port. Implementation efforts therefore went into developing functional serial communication, and have it translated to HID.

4.3.1 UART

Implementing the functionality of the board to receive and transmit data marks the initial stage of converting the board into an authenticator. Communication between the PC and the board occurs via a serial connection. This is achieved by accessing

the Universal Asynchronous Receiver-Transmitter (UART) peripheral on the board. The code below is used to declare and initialize a constant pointer to a `device` struct for the UART device. This pointer can then be used to interact with the UART device for receiving and transmitting serial data. Note the Zephyr-specific functions such as `DT_CHOSEN` is a macro provided by the Zephyr's device tree API. The details of how Zephyr's own APIs operate are beyond the scope of this work.

```
1 define UART_DEVICE_NODE DT_CHOSEN(zephyr_shell_uart)
2 static const struct device *const uart_dev =
      DEVICE_DT_GET(UART_DEVICE_NODE);
```

Listing 4.1: UART initialization.

Zephyr provides three different UART APIs: polling, interrupt-driven, and asynchronous. Of these three, the first two are used. In this implementation, the interrupt-driven API is used for receiving, and the polling API is used for transmitting. Interrupt-driven reception allows the board to handle incoming data as soon as it is sent, rather than having to wait for another process to complete.

4.3.2 Data Reception

Zephyr's interrupt-driven UART API is used for reception. It provides the function `uart_irq_callback_user_data_set()` which takes a UART device and a callback function as parameters and is initialized in the main method. When an interrupt request is triggered, the specified function will be called. The callback function is responsible for reading the incoming data, and its most important elements are defined below.

```
1  void uart_rx_isr(const struct device *dev, void *user_data) {
2      uint8_t c;
3      ...
4      while (uart_fifo_read(uart_dev, &c, 1) == 1) {
5          if ((c == '\n' || c == '\r') && rx_buf_pos > 0) {
6              rx_buf[rx_buf_pos] = '\0';
7              k_msgq_put(&uart_msgq, &rx_buf, K_NO_WAIT);
8              ...
9          } else if {...}
10     }
11 }
```

Listing 4.2: Callback function for reading incoming data.

The Zephyr function `uart_fifo_read()` reads a single character `c` at a time and puts it in the global variable `rx_buf` until either a line feed (\n) or carriage return (\r) character is read. When all data is read, it is put in a global message queue `uart_msgq`. A loop in the main method continuously tries to read messages stored in the message queue for further processing. This is done with `k_msgq_get()` reading

messages from the queue in a first-in-first-out manner. The read message is passed to `process_ctap_request()` for further processing.

```
while (k_msgq_get(&uart_msgq, rx_buf, K_FOREVER) == 0) {
    size_t rx_buf_msg_len = find_data_length(rx_buf, sizeof(rx_buf));
    process_ctap_request(rx_buf, rx_buf_msg_len, &ctr_drbg);

    // Clear receive buffer for next message
    memset(rx_buf, 0, sizeof(rx_buf));
}
```

Listing 4.3: Processing of message queue.

4.3.3 Data Transmission

For data transmission, Zephyr's polling UART API is used, and the related function used is `uart_poll_out()`, taking a pointer to a UART device as defined in Listing 4.1, as well as a pointer to the `char` to be sent. For example, transmitting the byte sequence `0x01`, `0x02`, `0x03` would be performed as in the listing below.

```
uint8_t three_bytes[] = {0x01, 0x02, 0x03};

for (int i = 0; i < sizeof(three_bytes); i++) {
    uart_poll_out(uart_dev, three_bytes[i]);
}
```

Listing 4.4: Example of a simple transmission.

In CTAP responses, messages are prepended with a byte `0x00` to indicate success. In addition, the Python script handling HID communications, must know when to stop reading the serial message being sent, which is why each serial transmission end with the transmission of the unique string "`CLD_EOL\n`". A complete transmission of a CTAP response would therefore be performed as below. Although not shown here, in the actual codebase, the original transmission code is wrapped inside a function called `send_ctap_reply`, which takes a byte array and its length and performs the operations just mentioned.

```
char end_sequence[] = "CLD_EOL\n";
...
uint8_t ctap_response;
size_t ctap_response_len;

// Indicate success
uart_poll_out(uart_dev, 0x00)

// Payload
for (int i = 0; i < ctap_response_len; i++) {
    uart_poll_out(uart_dev, ctap_response[i]);
}
```

```
13
14 // Ending sequence to signal the end of transmission
15 for (int i = 0; i < strlen(end_msg); i++) {
16     uart_poll_out(uart_dev, end_msg[i]);
17 }
```

Listing 4.5: Example of a full transmission sequence for a CTAP message.

4.4 Concise Binary Object Representation (CBOR)

All messages exchanged in the CTAP protocol are encoded in CBOR [Bor20]. It is a binary data serialization format, and much like JSON, it allows the transmission of data objects consisting of key-value pairs, but in a much smaller format. This improves processing and transmission performance, at the cost of human readability. Often, CBOR allows some information to be encoded in several variants that take up different lengths in bytes. The encoder is generally free to choose the length that is most practical for it, but for most, it is natural to choose the encoding that results in the shortest form. This type of encoding is known as “Preferred Encoding”. There is also “Deterministic Encoding” which goes beyond Preffered Encoding, and its purpose is to always produce the same encoding for data items that are equivalent at the data model level. This is done through defining encoding rules that an encoder must then follow. CTAP uses this format, also referred to as “Canonical Encoding”. The CBOR encoding / decoding library in this work is *zcbor* which can be made to use Canonical Encoding by enabling the configuration `CONFIG_ZCBOR_CANONICAL=y`.

4.4.1 CBOR Encoding and Decoding

Encoding and decoding the CTAP commands correctly is critical. The purpose of *zcbor* is to generate C code capable of encoding and decoding CBOR data. The code generation is based on a schema written in Concise Data Definition Language (CDDL), which is a language used for expressing CBOR data structures. The CDDL-schemas are, in turn, based on the structure of the CTAP command to be received or the CTAP reply to be transmitted, e.g., an `authenticatorMakeCredential` command or `authenticatorMakeCredential` reply. To demonstrate how encoding and decoding is done in practice, the CTAP command `authenticatorMakeCredential` is used to showcase how its decoding is done, as well as the encoding of its response.

The first step is identifying the structure of the object being sent. These are well documented in the CTAP documentation. A typical CBOR structure description is listed below, this one for the `authenticatorMakeCredential` command. Here, *Key* are byte identifiers for the parameters, and *Value Type* is what data type the parameter is. Not all parameters are required, and for our intents and purposes, we will for

`authenticatorMakeCredential` only use the required ones, i.e., *clientDataHash*, *rp*, *user*, and *pubKeyCredParams*.

Command	Parameter Name	Key	Data Type
authenticatorMakeCredential	clientDataHash	0x01	byte string
	rp	0x02	map
	user	0x03	map
	pubKeyCred-Params	0x04	array of maps
	excludeList	0x05	array of maps
	extensions	0x06	map
	options	0x07	map
	pinAuth	0x08	byte string
	pinProtocol	0x09	integer

Table 4.1: Structure of an `authenticatorMakeCredential`-command in CTAP. Not all parameters are required.

The next step is to create the CDDL-file that *zcbor* will use to generate the encoding / decoding C code. More granular documentation of the data or fields that each key-value pair should contain is available in the CTAP documentation. From reading it, one can generate the CDDL-file below. In it, the name *make_credential_request* is arbitrary. It is also important to note that *zcbor* encodes / decodes data in the order of the keys in the CDDL scheme. In accordance with canonical CBOR encoding, the keys in the CDDL must therefore be in bytewise lexicographic order. We use labels for the keys (here 1, 2, 3, 4) such that we can refer to each key with the name of the field it represents. This makes CDDL files easier to read and makes *zcbor* generate more sensible variable names.

```
1  client_data_hash = 1
2  rp = 2
3  user = 3
4  pub_key_cred_params = 4
5
6  make_credential_request = {
7      client_data_hash: bstr,
8      rp: {
9          "id": tstr,
10         "name": ? tstr,
11     },
12     user: {
```

```
13          "id": bstr,
14          "name": ? tstr,
15          "displayName": ? tstr,
16      },
17      pub_key_cred_params: [+{
18          "alg": int,
19          "type": tstr,
20      }],
21  }
```

Listing 4.6: CDDL file for generating C code for decoding an `authenticatorMakeCredential command`. The symbols '?' and '+' are CDDL syntax and means 'optional' and 'one or more', respectively.

Once a fitting CDDL scheme has been created, it can be used in conjunction with *zcbor*'s code generation command line tool. As we are currently dealing with an incoming request, we will use it with `--decode` to denote the generation of decoding code.

```
1 $ zcbor code --decode -c make_credential_request.cddl -t
      make_credential_request --oc make_credential_request_decode.c --oh
      make_credential_request_decode.h --oht
      make_credential_request_types.h
```

Listing 4.7: *zcbor* command for generating source code and header files related to CBOR decoding of an `authenticatorMakeCredential` command. The `-t` flag is used to choose which type to expose in the CDDL scheme, in this case `make_credential_request`. Flags `-oc` and `-oh` refer to the path to the generated C and header file, respectively. Lastly, `-oht` is the path to the generated types header.

When the command above is run, *zcbor* will read the CDDL file and create C struct types in a new file *make_credential_request_types.h* that match the types described in the scheme. It then generates code to decode CBOR data into these structs, and/or code for encoding CBOR from the data in the structs, if being called with `--encode`.

```
1  struct rp_name {
2      struct zcbor_string rp_name;
3  };
4
5  struct user_name {
6      struct zcbor_string user_name;
7  };
8
9  struct user_displayName {
10     struct zcbor_string user_displayName;
11 };
12
13 struct pub_key_cred_params_map {
```

```
14     int32_t map_alg;
15     struct zcbor_string map_type;
16 };
17
18 struct make_credential_request {
19     struct zcbor_string make_credential_request_client_data_hash;
20     struct zcbor_string rp_id;
21     struct rp_name rp_name;
22     bool rp_name_present;
23     struct zcbor_string user_id;
24     struct user_name user_name;
25     bool user_name_present;
26     struct user_displayName user_displayName;
27     bool user_displayName_present;
28     struct pub_key_cred_params_map pub_key_cred_params_map[3];
29     size_t pub_key_cred_params_map_count;
30 };
```

Listing 4.8: Generated structs in *make_credential_request_types.h* as a result of running the previous command (Listing 4.7) with the CDDL scheme defined in Listing 4.6.

The other header file that *zcbor* generated from the above is `make_credential_request_decode.h`. This file includes a function declaration for decoding a `make_credential_request`.

```
1 int cbor_decode_make_credential_request(
2     const uint8_t *payload, size_t payload_len,
3     struct make_credential_request *result,
4     size_t *payload_len_out);
```

Listing 4.9: Function declaration in *make_credential_request_decode.h* as generated by *zcbor* for decoding a `make_credential_request`.

This implies that an incoming CBOR encoded CTAP command can be passed as an argument to the decoding function, after which the *zcbor*-generated C code will manage the decoding process and parse the decoded data into the declared `make_credential_request` struct.

```
1 struct make_credential_request make_cred_request_decoded;
2 size_t make_cred_request_decoded_len;
3
4 printk("Decoding make credential request...");
5 if (cbor_decode_make_credential_request(ctap_request_param_data,
6         ctap_request_param_data_len,
7         &make_cred_request_decoded,
8         &make_cred_request_decoded_len) != 0) {
9     printk("Error decoding make credential request.\n");
10    return;
11 }
```

```
12 printk("    ok\n");
```

Listing 4.10: Call to a *zcbor*-generated function for decoding a CBOR message.

Upon successful decoding, access to the decoded data in `make_cred_request_decoded` is available, akin to accessing elements in any other struct. That is the general process of how the CBOR decoding is done. The varying factor is the structure of the object coming in, calling for a different CDDL structure. Now, a brief overview of the encoding process will be provided, which closely mirrors the process described. To illustrate, an attestation object will be encoded, which is what the client receives following an `authenticatorMakeCredential` CTAP command.

Reply	Member Name	Key	Data Type
Attestation object	fmt	0x01	string
	authData	0x02	byte string
	attStmt	0x03	byte string

Table 4.2: Brief overview of the structure of an attestation object which gets returned upon an `authenticatorMakeCredential` command. For further details on its structure, see Figure 4.3.

Upon further studying the CTAP documentation, one would create a CDDL scheme similar to the one below.

```
1  fmt = 1
2  auth_data = 2
3  att_stmt = 3
4
5  attestation_object = {
6      fmt: tstr,
7      auth_data: bstr,
8      att_stmt: {
9          "alg": int,
10         "sig": bstr,
11     },
12 }
```

Listing 4.11: CDDL scheme used by *zcbor* to generate code for CBOR encoding an attestation object. The first three lines assigns a byte-key to each field.

Next, using *zcbor* to generate C code for encoding:

```
$ zcbor code --encode -c attestation_object.cddl -t attestation_object
    --oc attestation_object_encode.c --oh attestation_object_encode.h
    --oht attestation_object_types.h
```

Listing 4.12: *zcbor* command for generating source code and header files related to CBOR encoding an attestation object.

This results in an `attestation_object` struct in `attestation_object_types.h`:

```
struct attestation_object {
    struct zcbor_string attestation_object_fmt;
    struct zcbor_string attestation_object_auth_data;
    int32_t att_stmt_alg;
    struct zcbor_string att_stmt_sig;
};
```

Listing 4.13: Struct for an attestation object in `attestation_object_types.h` as generated by zcbor.

Also, in `attestation_object_encode.h`, a function declaration for encoding:

```
int cbor_encode_attestation_object(
    uint8_t *payload, size_t payload_len,
    const struct attestation_object *input,
    size_t *payload_len_out);
```

Listing 4.14: Function signature in `attestation_object_encode.h` for encoding an attestation object as generated by zcbor. Before the function is called, the struct `attestation_object` should have its fields initialized.

After a successful call to `cbor_encode_attestation_object`(), the pointer argument passed as the `payload` parameter will then hold the CBOR-encoded data.

4.5 CTAP Commands

This section will focus on the implementation of three CTAP commands, namely `authenticatorGetInfo`, `authenticatorMakeCredential`, and `authenticatorGetAssertion`. This includes how these commands are processed by the authenticator to return an appropriate response. These three commands are the minimum required to register a credential with a RP and perform authentication for that registered credential. With the exception of `authenticatorGetInfo` because of its static nature, the other commands were first implemented for ECDSA, then Dilithium-2, and finally the hybrid version. This section will primarily focus on the hybrid implementation, as that is the novelty of this work.

4.5.1 High-level Overview

The function responsible for handling CTAP commands and replies is `process_ctap_command`. It takes the data that was read in Listing 4.2 and initially reads the first byte of this data to identify what CTAP operation is being requested.

CTAP Command	Byte Identifier
`authenticatorGetInfo`	0x04
`authenticatorMakeCredential`	0x01
`authenticatorGetAssertion`	0x02

Table 4.3: The first byte of a CTAP message indicates what operation is being requested.

After the requested operation has been identified in the form of a byte, this value is evaluated in a switch statement and the relevant code block is triggered for further execution with the remaining data of the message. The cases for `authenticatorMakeCredential` and `authenticatorGetAssertion` are further divided into three cases based on what algorithm was negotiated between the authenticator and the RP; ES256, Dilithium-2, or hybrid. Here, ES256 is the name for ECDSA with SHA-256 as defined in [SAT24]. These two names will be used interchangeably. Additionally, in the code, `DILITHIUM` refers to Dilithium-2.

```
1  #define MAKE_CREDENTIAL 0x01
2  #define GET_ASSERTION 0x02
3  #define GET_INFO 0x04
4
5  #define ES256 -7
6  #define DILITHIUM -51
7  #define ES256_DILITHIUM -52
8
9  void process_ctap_command(const uint8_t *ctap_request, ...) {
10     switch (ctap_request[0]) {
11     case MAKE_CREDENTIAL:
12         ...
13         negotiate_algorithm();
14
15         switch (negotiated_alg) {
16         case ES256:
17             // Do makeCredential for ES256
18         case DILITHIUM:
19             // Do makeCredential for Dilithium-2
20         case ES256_DILITHIUM:
21             // Do makeCredential for hybrid
22         }
23     case GET_ASSERTION:
```

```
24          ...
25          switch (negotiated_alg) {
26          case ES256:
27              // Do getAssertion for ES256
28          case DILITHIUM:
29              // Do getAssertion for Dilithium-2
30          case ES256_DILITHIUM:
31              // Do getAssertion for hybrid
32          }
33
34      case GET_INFO:
35          // Send device info
36      }
37  }
```

Listing 4.15: High-level overview of `process_ctap_command()`, the function responsible for performing the appropriate operations based on the received CTAP command.

We now present how the three different commands were implemented in further detail.

4.5.2 authenticatorGetInfo

The `authenticatorGetInfo` (byte identifier 0x04) command is the first message sent from the client to the authenticator when communication is started. Its purpose is letting the client know the capabilities of the authenticator such that the client can adjust its behavior to fit the authenticator. The command takes no parameters, and since the authenticator's capabilities do not change, a static response is appropriate. Since this work only presents a PoC, we are only interested in the required fields of the `authenticatorGetInfo` reply; *versions* and *aaguid*, representing protocol version and an identifier for the authenticator, respectively. The *versions* text field is set to "FIDO_2_0" to indicate we are working with CTAP2/FIDO2, and the *aaguid* bytearray field is for our intents and purposes arbitrary except for the length which should be 16 bytes, and is therefore set to 16 byte values each representing the value 9, i.e., {`0x09`, ..., `0x09`}.

Since the reply of a `authenticatorGetInfo` command will always be the same, a CBOR encoding of the response is generated at the start of the program and then stored such that it can be quickly returned whenever the client asks for it. The CBOR encoding of the reply is based on the CDDL file below.

```
1 versions = 1
2 aaguid = 3
3
4 get_info_reply = {
5     versions: [+tstr],
```

```
6     aaguid: bstr,
7 }
```

Listing 4.16: CDDL scheme used by *zcbor* to generate C code for CBOR encoding an `authenticatorGetInfo` reply. The first two lines assigns a byte-key to each field.

Once the authenticator receives a CTAP command starting with 0x04, the authenticator identifies it as an `authenticatorGetInfo` command and goes into the appropriate case inside `process_ctap_request`, in which the reply is sent.

```
1 case GET_INFO:
2     send_ctap_reply(cbor_encoded_device_info,
          cbor_encoded_device_info_len);
```

Listing 4.17: Transmission of an `authenticatorGetInfo` response containing CBOR encoded information of the device.

4.5.3 authenticatorMakeCredential

One of the most fundamental commands of CTAP is `authenticatorMakeCredential`. This section presents how it was implemented in this work.

High-level Overview of Command and Reply

The `authenticatorMakeCredential` command prompts the authenticator to generate a new keypair for an agreed upon algorithm. For this work, only the required fields of the command are in the request coming from the test server (RP).

Command	Parameter Name	Key	Data Type
authenticatorMakeCredential	clientDataHash	0x01	byte string
	rp	0x02	map
	user	0x03	map
	pubKeyCred-Params	0x04	array of maps

Table 4.4: Structure of an `authenticatorMakeCredential` command used in this work.

An example of a decoded `authenticatorMakeCredential` command is shown below. The cryptographic algorithm used to generate a new public keypair during a FIDO2/Webauthn registration is determined by the authenticator and the preferences of the RP. The allowed algorithms are in the `pubKeyCredParams` field, and are listed

in the order in which they are preferred. The COSE Algorithms registry [SAT24] contains definitions of all possible algorithms that can be used, each identified with an integer value.

```
{1: h'56CDD025C70E0C3B1C768D082492ECBACFDC13974FE712A6C6F01FFA146CCC79',

 2: {"id": "localhost",
     "name": "ponebiometrics.com"},

 3: {"id": h'3082019330820138A003020102308201933082013 8A00302010230820193308201 38A00302010230820193308 2',
     "name": "digneludvig",
     "displayName": "digneludvig"},

 4: [{"alg": -7,                // ES256 (ECDSA w/ SHA-256)
      "type": "public-key"},
     {"alg": -257,              // RS256 (RSASSA-PKCS1-v1_5 w/ SHA-256)
      "type": "public-key"}]}
```

Figure 4.2: Example of a decoded `authenticatorMakeCredential` command. Key 1 is the *clientDataHash*. Key 2 is information on the RP. Key 3 is information on the user, with *id* being a RP-specific account identifier. Key 4 is *pubKeyCredParams*, consisting of algorithms supported by the RP.

The authenticator replies to `authenticatorMakeCredential` with an attestation object containing the public key along with other attestation data. An attestation object has three fields; *fmt*, *attStmt*, and *authData*. It is broken down in further detail in the figure below.

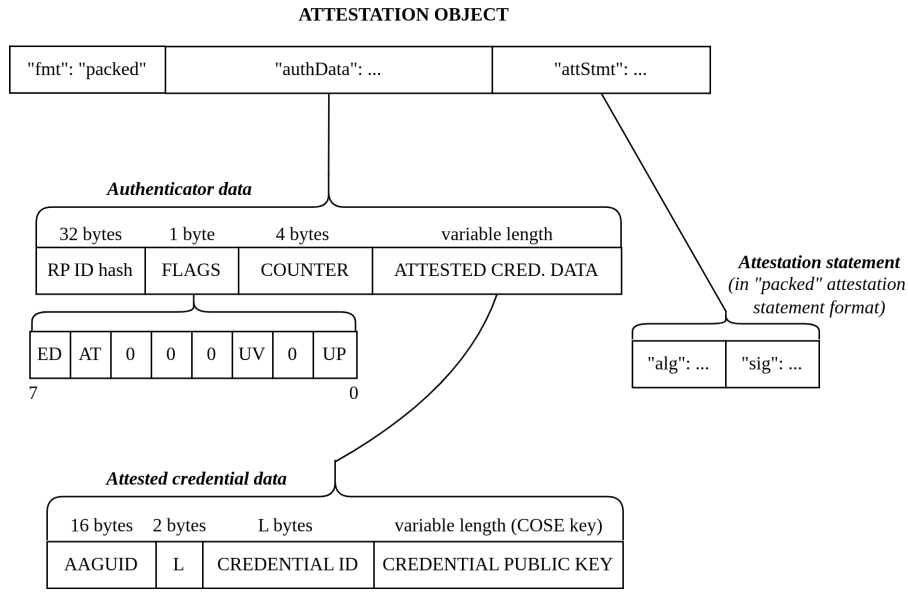


Figure 4.3: Structure of an attestation object, and is what is returned upon an `authenticatorMakeCredential` command [FID18].

CBOR Object Signing and Encryption (COSE) Keys

To proceed with the presentation of the implementation, we must first talk about COSE keys, as these play a central role in FIDO2/WebAuthn. COSE is a specification for how various security services should be represented in CBOR, such as how to represent cryptographic keys. A COSE key structure is built on a CBOR map object, and must include the field *key type (kty)*, but may also include other fields. *Kty* is used to identify the family of keys the structure belong to. All our COSE keys will also include the *algorithm (alg)* field, which is used to restrict the algorithm being used with the key. These *alg* values are the same as those in `pubKeyCredParams` in that they come from the COSE Algorithms list [SAT24].

To understand the COSE key structure for the proposed hybrid mechanism, one must first understand the COSE key structure used for ECDSA and Dilithium separately. An example of, and the one used in this work, a COSE key structure for an elliptic curve public key in EC2 format [Sch17a], on the P256 curve, to be used with the ES256 signature algorithm (ECDSA w/ SHA-256) [Sch17b] is shown below.

```
{
    1:   2,     ; kty: EC2 key type
    3:  -7,     ; alg: ES256 signature algorithm
   -1:   1,     ; crv: P-256 curve
   -2:   x,     ; x-coordinate as byte string, 32 bytes
                ; e.g., in hex: 65eda5a12577c2ba...e108de439c08551d
   -3:   y      ; y-coordinate as byte string, 32 bytes
                ; e.g., in hex: 1e52ed75701163f7...40ddf9f341b3dc9b
}
```

Figure 4.4: COSE key structure for the ES256 algorithm (ECDSA w/ SHA256).

As the Dilithium algorithm is still in its early stages in terms of reaching widespread implementation, there is currently no official COSE serialization for a Dilithium public key, let alone a hybrid one. However, there exists an Internet-Draft [PSM+24] describing COSE serialization for ML-DSA, an algorithm derived from Dilithium. The Dilithium COSE key structure used in this work is based on the structure described in that document. The chosen structure is shown below.

```
{
    1: 7,       ; kty
    3: -51,     ; alg
   -14: pk      ; public key as byte string, 1312 bytes (for Dilithium-2)
                ; e.g., in hex: 7803c0f9f1a4e7d3...3bba7abdf2da5bea
}
```

Figure 4.5: Chosen COSE key structure for the Dilithium-2 algorithm.

For the hybrid COSE key structure, which contains both the ECDSA and Dilithium-2 public keys, the union of the two individual COSE key structures is used. Since they both share *kty* and *alg* fields with unique values for these, we set a common *kty* value to 8, and a common *alg* value to one less than that of the *alg* value for ML-DSA in the draft [PSM+24], namely -52. This results in the structure below.

```
{
    1: 8,       ; kty
    3: -52,     ; alg
   -1: 1,       ; crv: P-256 curve
   -2: x,       ; x-coordinate as byte string, 32 bytes
   -3: y,       ; y-coordinate as byte string, 32 bytes
   -14: pk      ; public key as byte string, 1312 bytes (for Dilithium2)
}
```

Figure 4.6: Chosen hybrid COSE key structure for ECDSA and Dilithium-2 keys.

Generating the Keys

We now present how the keys are generated once an `authenticatorMakeCredential` command is received. Firstly, let us set some context for the code relating to keys. There exists an array containing algorithm identifiers for the algorithms the authenticator supports, and is used for negotiating an algorithm with an RP, where the algorithms the RP supports are in the `pubKeyCredParams` in 4.5.3.

```
1 const uint32_t SUPPORTED_ALGORITHMS[] = {ES256_DILITHIUM, DILITHIUM,
      ES256};
```

Listing 4.18: Global array containing algorithms supported by the authenticator. Preferability goes from most preferred (lowest index) to least preferred (highest index). The macros used here are the same as defined in Listing 4.15.

Since this is only a PoC, we chose to have only one credential active on the authenticator at a time, so that the retrieval of the correct credential is simple. Credential data is stored in a struct `Credential` that is instantiated only once as a global variable.

```
1 typedef struct {
2     char rp_id[MAX_RP_ID_LENGTH];
3     uint8_t credential_id[CREDENTIAL_ID_LENGTH];
4     mbedtls_mpi ecdsa_private_key;
5     uint8_t dilithium_private_key[OQS_SIG_dilithium_2_length_secret_key];
6     uint32_t sign_count;
7 } Credential;
```

Listing 4.19: Struct used for storing a generated credential. `mbedtls_mpi` is a library specific (*mbedtls*) data type with `mpi` meaning multi precision integer.

When the `authenticatorMakeCredential` command is received, byte identifier `0x01` triggers the `MAKE_CREDENTIAL` case as seen in Listing 4.15. The initial step involves decoding the request, followed by algorithm negotiation. The latter is done using the `negotiate_algorithm()`, which will take the most preferred algorithm of the authenticator and see if the RP supports it, otherwise move on to the second most preferred, and so on. The algorithm agreed upon is stored in the global variable `negotiated_alg`. If the two parties cannot agree on an algorithm, a `CTAP2_ERR_UNSUPPORTED_ALGORITHM` error is returned from the authenticator.

With `ES256_DILITHIUM`, as the negotiated algorithm, a hybrid credential will be generated. In practice, this means the generation of an ECDSA keypair, as well as a Dilithium-2 keypair. It should be noted that the Nucleo-LG476RG development board does not have an entropy device on the system, and that fake entropy is therefore used. This is not cryptographically secure, but is appropriate for testing and PoC purposes.

```
case ES256_DILITHIUM:
    struct credential_public_key_hybrid credential_public_key_hybrid;
    ...
    // Dilithium-2 key generation
    uint8_t
        public_key_dilithium[OQS_SIG_dilithium_2_length_public_key];

    if (OQS_SIG_dilithium_2_keypair(public_key_dilithium,
        credential.dilithium_private_key) != OQS_SUCCESS) {
        printk("Error generating Dilithium keypair.\n");
        return;
    }

    // ES256 key generation (ECDSA w/ SHA-256)
    uint8_t pk_x_coord[ECDSA_PUBLIC_KEY_COORD_SIZE]
    uint8_t pk_y_coord[ECDSA_PUBLIC_KEY_COORD_SIZE];

    if (generate_ecdsa_keypair(ctr_drbg, pk_x_coord, pk_y_coord,
        &credential.ecdsa_private_key) != 0) {
        printk("Error generating ECDSA keypair.\n");
        return;
    }
    ...
```

Listing 4.20: Generation of ECDSA and Dilithium-2 keypairs in the hybrid case. Dilithium-2 uses the API provided by the *liboqs* library, and ECDSA the one provided by *mbedtls*. Here, `generate_ecdsa_keypair()` is not a function signature exposed by *mbedtls*, but rather a wrapper function created in this work to factor out some code from the `process_ctap_command()` function. The private key for each respective algorithm is stored in its appropriate field in the `Credential` struct.

In the above listing, a struct `credential_public_key_hybrid` is declared. This struct is generated by *zcbor* using a CDDL scheme based on the hybrid COSE key structure previously defined. This structure has its fields populated via a helper function `populate_credential_public_key_hybrid()`, which takes the public keys previously generated as arguments and fills the fields *kty* and *alg* with their hardcoded values for the hybrid credential. The `credential_public_key_hybrid` object is then CBOR encoded and part of the attestation object which gets returned to the client.

4.5.4 authenticatorGetAssertion

The second fundamental CTAP command is `authenticatorGetAssertion`. This section presents its implementation in this work.

High-level Overview of Command and Reply

The `authenticatorGetAssertion` command is sent to the authenticator when the user requests to authenticate against the RP. In this work, the request will contain

three fields, two of which are required, *rpId* and *clientDataHash*, and one that is not, *allowList*.

Command	Parameter Name	Key	Data Type
authenticatorGetAssertion	rpId	0x01	string
	clientDataHash	0x02	byte string
	allowList	0x03	array of maps

Table 4.5: Structure of an `authenticatorGetAssertion` command used in this work.

Below is an example of a decoded `authenticatorGetAssertion` command.

```
{1: "localhost",
 2: h'DF70A405193F9E41F8C5C7FB0E4ED03B4CFDA6D61B4B9A82C3F52B68C0FF6E4F',
 3: [{"id": h'2DCF462904B478D868A7FF3F2BF1FCD9',
      "type": "public-key"}]}
```

Figure 4.7: Example of a decoded `authenticatorGetAssertion` command used in this work.

Once the command is received, the data in the request will be processed, and a response is returned containing the required fields, i.e., *authData* and *signature*.

Reply	Member Name	Key	Data Type
Assertion object	authData	0x02	byte string
	signature	0x03	byte string

Table 4.6: Structure of the object that gets returned upon on an `authenticatorGetAssertion` command. Contains the signature to be verified for authentication.

The Hybrid Signature Construction

We will first present the chosen structure of the hybrid signature, before showing how it is generated in practice on the authenticator. The hybrid construction chosen is a concatenation of a classical ECDSA signature and a quantum-secure Dilithium-2 signature, where each signature is generated separately with the same message as input. To verify this hybrid construction, the verifier will first split the signature into its respective components (ECDSA and Dilithium-2 signatures) and verify each

signature separately. The hybrid signature will verify if and only if both individual signatures are verified.

The verifier must know how to split the hybrid signature into its respective components, which is why each separate signature in the hybrid one is prepended with a two byte value indicating the length of its signature. This is necessary because the signature lengths may vary. ECDSA signatures (ASN.1 DER format) generated in this work vary between being 70, 71, or 72 bytes long. In addition, the Dilithium signature lengths vary based on the security level used.

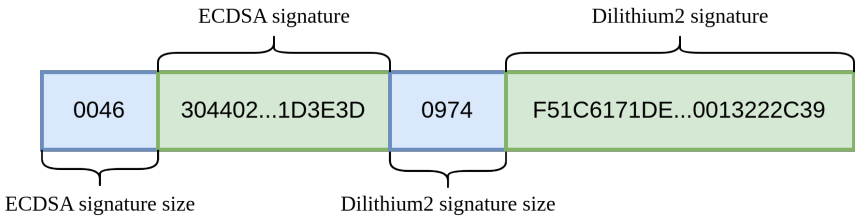


Figure 4.8: Example of a hybrid signature used in this work, showing the hybrid construction used. Each size field is a two-byte integer value indicating how the hybrid signature should be split.

Generating the Hybrid Signature

The message that gets signed is the concatenation of the *clientDataHash* field in `authenticatorGetAssertion` and the *authData* object that gets built upon authentication. This *authData* object is not the same as the one built during `authenticatorMakeCredential`, although it is similar. This one does not contain a credential public key, and the counter is increased by one (going from 0 to 1 if it is the first time signing with the credential). Each algorithm generates a signature on the message individually, after which the two generated signatures are put together to form the hybrid signature. In this work, the ECDSA signature takes a hash of the message as input because the *mbedlts* implementation of the ECDSA algorithm does not perform hashing as part of the algorithm as is typical behavior.

```
1 /* Build new authData and concatenate clientDataHash with authData as
      message */
2
3 // ECDSA
4 unsigned char signature_ecdsa[MBEDTLS_ECDSA_MAX_LEN];
5 size_t signature_ecdsa_len;
6 generate_ecdsa_signature(credential.ecdsa_private_key, ...);
7
8 // Dilithium-2
9 unsigned char signature_dil[OQS_SIG_dilithium_2_length_signature];
```

```
size_t signature_dil_len;
OQS_SIG_dilithium_2_sign(credential.dilithium_private_key, ...);

// Hybrid
unsigned char signature_hybrid[2 + MBEDTLS_ECDSA_MAX_LEN + 2 +
    OQS_SIG_dilithium_2_length_signature];

/* Convert ECDSA and Dilithium-2 signature lengths to 2 byte values */

/* memcpy() used for building the hybrid construction */

// Transmission
/* Build, encode, and transmit assertion object */
```

Listing 4.21: Generation of a hybrid signature, i.e., an ECDSA signature concatenated with a Dilithium-2 signature, with each signature prepended with a two-byte integer value indicating the length of the individual signature.

With this hybrid signature construction, it was possible to authenticate against the local test server with what is believed to be quantum-security. This demonstrates that using Dilithium in FIDO2 is plausible, but there are many nuances to explore, which will be addressed in the next chapter.

Chapter 5
Performance and Discussion

This chapter presents benchmarks conducted on the implementation developed in this work. It begins with showcasing the results for the Dilithium algorithm and its hybrid version with ECDSA. Additionally, benchmarks for another PQC algorithm, Falcon-512, are presented to provide further context for discussion.

The discussion section explores the nuances of hybridization, including various hybrid constructions. It also delves into performance trade-offs between different algorithms. Furthermore, the relevance of SEs and SCAs to this work is discussed. Finally, the chapter concludes by presenting current challenges associated with quantum-securing FIDO2.

5.1 Performance

The performance results for the implemented algorithms are presented below. The standard deviation σ is a measure of the variance in the data in relation to the mean.

Algorithm	**ECDSA**	**Dilithium-2**	**Hybrid**
		Time	
Key Generation	940 ms ($\sigma = 1.2$)	49 ms ($\sigma = 0.6$)	996 ms ($\sigma = 1.3$)
Signing	920 ms ($\sigma = 1.4$)	132 ms ($\sigma = 73.8$)	1048 ms ($\sigma = 68.7$)
		Size (Bytes)	
Signature	71 ($\sigma = 0.65$)	2420	2495 ($\sigma = 0.65$)
Public Key	64	1312	1376

Table 5.1: Key generation and signing times, along with signature sizes, for classical ECDSA, Dilithium-2, and the hybrid version. ECDSA's public key size is set to 64 as it consists of an X and Y coordinate, each being 32 bytes. The hybrid public key is then calculated as the sum of 64 and the public key size of Dilithium-2.

The most notable observation is that Dilithium-2 is significantly faster than classical ECDSA, with key generation being nearly 20 times quicker. This is expected, as Elliptic Curve Cryptography (ECC) is often slower in constrained environments due to complex operations, whereas lattice cryptography is often fast in constrained environments due to simpler arithmetic operations. The hybrid signature shows key generation and signing times that are effectively the sum of the time taken by ECDSA and Dilithium-2, respectively. This is a natural consequence of how the hybrid signature was built, where the signatures are generated one after the other in sequence and then combined into a single structure. Another observation to make is that the standard deviation σ of the signing time for Dilithium-2 is high. This variation arises due to Dilithium's rejection sampling technique, as explained in Subsection 2.3.3, where the number of iterations executed by the algorithm fluctuates, naturally resulting in fluctuations in time.

We performed the same benchmarks for another lattice-based algorithm being standardized by NIST for PQC, namely Falcon-512 [FHK+18]. Although it is also lattice-based, it is more complex than Dilithium, and has a significantly longer key generation time. However, its signature and public key sizes are smaller, which can be beneficial in certain use cases.

Algorithm	**Falcon-512**	**Hybrid (ECDSA w/ Falcon-512)**
	Time	
Key Generation	4270 ms ($\sigma = 2046$)	5337 ms ($\sigma = 1898$)
Signing	1060 ms ($\sigma = 2.5$)	2071 ms ($\sigma = 2.3$)
	Size (Bytes)	
Signature	655 ($\sigma = 2.4$)	730 ($\sigma = 2.5$)
Public Key	897	961

Table 5.2: Key generation and signing times, along with signature and public key sizes, for Falcon-512 and its hybrid version with ECDSA. The hybrid public key size here is calculated in the same way as in Table 5.1.

5.2 Discussion

This section presents a discussion covering various aspects of quantum-securing FIDO2. These topics include using hybrid vs. pure PQC, the nuance of hybrid constructions, performance trade-offs between algorithms, considerations regarding SE and SCA, as well as general challenges for the FIDO2 standard. The discussion ends with a short reflection on how this work relates to the UN Sustainable Development goals.

5.2.1 Hybrid vs. Pure PQC

The decision to use hybrid constructions to move smoothly into a quantum-secure digital landscape is not obvious. In the United States, the National Security Agency (NSA) has included Dilithium as part of their new algorithm suite, CNSA 2.0 [Nat22b]. They have explicitly mentioned that they will not require developers to implement hybrid solutions for security purposes, instead proposing to only use PQC algorithms, expressing confidence in the robustness of these. The NSA argues that hybrid solutions add unnecessary complexity to protocols and introduce interoperability concerns, as all communicating parties must adopt the same algorithms and hybridization methods. Furthermore, a hybrid approach would eventually lead to a second transition phase to entirely phase out classical algorithms once the PQC ones are deemed mature enough [Nat22c]. While these arguments are valid and highlight practical concerns, it is important to consider previous controversies surrounding the NSA, such as allegations of deliberately subverting cryptographic standards to include their own backdoor, as exemplified by the Dual EC case [BLN16].

In contrast, the national security authority (NSM) in Norway recommends the use of hybridization [Øde24], arguing that new algorithms should be combined with established classical algorithms in which there is already a high level of trust. This approach aims to ensure a lower bound of security provided by the classical algorithms while the newer algorithms undergo further scrutiny. Similarly, Germany's Federal Office for Information Security (BSI) shares this recommendation, citing that the newer algorithms have not been scrutinized to the same extent as classical ones, and thus should be used in a hybrid construction to ensure robust security [Fed21].

5.2.2 Hybrid Constructions

While it is well understood that PQC hybridization should enable security that is lower bounded by the classical algorithm, the question arises as to what such a hybrid construction should look like, and what properties it should have. In this work, we implemented a simple hybrid construction consisting of two concatenated signatures, each prepended by a two-byte value indicating the individual signature's length. However, there are many other types of constructions and means for achieving hybridization. In [BH23], the authors mention several desirable properties of hybrid signatures, the most critical one being *proof composability*. This is the fundamental premise behind hybridization, meaning that the algorithms included in the hybrid construction must be combined in such a way that it is possible to prove reduction to the security properties of each individual algorithm. Without this property, users cannot be assured that the hybrid construction builds on the standardization process and analysis performed to date on the component algorithms – the hybrid construction becomes, in effect, a completely new algorithm.

Furthermore, the authors of [BH23] discuss how various hybrid constructions of digital signatures exhibit different levels of separability. Separability refers to the extent to which a verifier can detect if one of the composite algorithms of the hybrid construction has been removed. Among the constructions with the highest degree of separability, the authors identify the concatenation of signatures (much like the method used in this work). For constructions with the lowest degree of separability, verification fails if one of the composite signatures is missing, but also if the verifier fails to verify both signatures. This exhibits a property the authors of [BH23] call *simultaneous verification*, where both composite signatures must be present, and the verifier cannot "quit" the verification process before both components are verified. Contrast this with the concatenation construction used in this work, which does not exhibit this property. In this case, a quantum-capable attacker could potentially forge the classical algorithm but not the PQC one. The attacker could then possibly perform a clock glitch attack to skip the second part of a verification statement, e.g., `verify(classic) && verify(PQC)`, and have the hybrid signature verified.

Ease of implementation is an important consideration. Simple concatenation is straightforward, but may expose certain security vulnerabilities. While other hybrid constructions with more desirable properties might offer enhanced security, they are likely more difficult to implement. With Norway and Germany recommending hybridization, it is probable that other developed countries also will, leading to widespread adoption of hybrid implementations. In such cases, ease of implementation will likely be a desirable criterion, even if it means accepting slightly lower security compared to more advanced implementations. Cryptography is already difficult to implement correctly and securely; adding additional layers of complexity for the sake of security may backfire, leading to other vulnerabilities due to faulty implementation. We may see more advanced constructions implemented in systems with higher security demands, which are likely the systems with the most urgent need to migrate to PQC.

5.2.3 Performance Trade-offs Between Algorithms

There are other PQC algorithms being standardized besides Dilithium, each with markedly different performance characteristics. For example, as illustrated in Table 5.2, the Falcon-512 algorithm has a significantly longer key generation time. However, its signature size is just over a quarter of that of Dilithium-2, and its public key size is approximately 30% smaller. When selecting a PQC algorithm for a specific use case, it is important to weigh the trade-offs between speed and size. For example, in the FIDO2 registration process, key generation with a RP is performed only once. Given the infrequency of this operation, the longer key generation time of Falcon-512 may be acceptable if the smaller signature size is prioritized, even if the signature generation itself is slightly slower. Additionally, in environments where bandwidth is a limiting factor, the smaller signature size of Falcon-512 could potentially result

in faster transmission, ultimately saving more time despite the longer signature generation time (consider Falcon-512 vs. Dilithium-2 in this work). These algorithms can also be instantiated for different security levels, which affect speed and sizes depending on the chosen level. This is also something that must be taken into consideration when choosing an algorithm, i.e., does a higher security level justify the increase in speed and size, and vice versa.

5.2.4 Secure Elements and Side-Channel Attacks

The fundamental security strength of security keys, such as the OFFPAD, lies in the use of secure microcontrollers, i.e., SE, to safely generate, store, and use secret cryptographic keys. These are crucial because without them, secret keys could be revealed through a SCA by an attacker with physical access to the device. SEs exist for many classical algorithms (e.g., ECDSA), but support for PQC is not yet available. In a scenario where an attacker successfully extracts a PQC secret key through SCA, they could forge signatures if pure PQC is used. If hybridization is used, this threat is mitigated, as the attacker would still need the classical secret key, which is assumed to be protected by a SE. This reinforces the choice to opt for hybridization over pure PQC, as discussed in Subsection 5.2.1.

In this work's authenticator implementation, no SE was used, making the secret keys susceptible to SCA by an attacker with physical access to the device. A question arises as to whether it makes sense for a security key manufacturer to implement a PQC algorithm despite it not being protected by a SE. Over the internet, the absence of a SE generally does not affect security, and the authenticator would still benefit from the PQC algorithm, even though it is not protected by a SE. With no attacker physically present, there is little to no SCA threat, and quantum security is still achieved if the developer chooses to implement PQC without SE protection. The small threat comes from possible timing attacks over the internet in which an attacker measures the time it takes to generate a signature, and from that may be able to extract the secret key. Work is being done on how to effectively prevent this, see [EEN+24] and [dPPRS23]. Ultimately, it is up to the user to decide how much risk they are willing to accept. SEs supporting PQC will likely be available in the near future, making this decision less critical for now.

5.2.5 Challenges for FIDO2

This work has shown that it is possible to use hybrid signatures to make FIDO2 authentication quantum-secure, but this is just one implementation. As previously discussed, there is a lot of nuance and many factors to consider when making a system quantum-secure. Ultimately, the FIDO Alliance decides the path forward for PQC in FIDO2 authentication. Firstly, they must decide if they should use pure PQC, hybrid,

or even both. This decision is not obvious, and different regions will likely have different degrees of preparedness for each scenario. The NSA recommending the use of pure PQC may result in local manufacturers of security keys in the United States not supporting hybrid signatures, while other regions (e.g., Norway and Germany) only supporting hybrids. It would probably make the most sense for the FIDO Alliance to make a decision and then have manufacturers tailor their new devices to fit this decision, rather than the FIDO Alliance trying to accustom many different implementations.

Secondly, assuming the FIDO Alliance opts for hybrid signatures, they must define the construction and security properties of the hybrid approach, as well as choose the algorithms to be used. This work demonstrated the feasibility of using the Dilithium algorithm in a simple concatenation construction, which was straightforward to implement and very fast. Similarly to how the FIDO2 standard supports various classical algorithms, it makes sense to offer multiple PQC algorithms, such as Dilithium and Falcon, possibly in different types of hybrid constructions. This approach would make FIDO2 flexible and supportive of various implementations of authenticators, giving users some agency over their exposure to risk (i.e., pure PQC vs. hybrid, simple hybrid construction vs. advanced hybrid construction, security level of algorithms). Additionally, it is important to note that, while not necessarily a challenge, work must be done to develop COSE representations for the public keys of the chosen PQC solutions. This work proposed a simple COSE structure for a hybrid ECDSA and Dilithium key, but unforeseen challenges may arise when mixing different types of classical algorithms with novel PQC ones. Properly defining these COSE structures is crucial to ensuring interoperability and security within the FIDO2 framework.

5.2.6 Relevance to the UN Sustainable Development Goals

The imminent threat posed by quantum computers necessitates the upgrading of digital infrastructure to ensure security. This work makes a contribution to enhancing the security of the FIDO2 standard, aligning with the 9th UN Sustainable Development Goal, i.e., “Build resilient infrastructure, promote sustainable industrialization, and foster innovation” [Uni24]. Specifically, it supports sub-goal 9.2, which emphasizes the importance of enhancing scientific research and upgrading technological capabilities across all countries.

Chapter 6
Conclusion and Future Work

In this concluding chapter, we address the research questions formulated at the outset of this work and summarize the findings of the study. The exploration of quantum-secure authentication within FIDO2 has led us to examine various aspects of hybridization, algorithm selection, user-centric security considerations, and other challenges with quantum-securing FIDO2. Let us now try to formulate some answers to our research questions and conclude, as well as propose some further research in this evolving domain.

6.1 Research Questions

RQ1: With respect to contemporary technology and research, is the FIDO2 standard ready to migrate to quantum-secure authentication?

The FIDO2 standard's readiness to migrate to a quantum-secure authentication mechanism hinges on several factors. This work presents a PoC implementation of quantum-secure FIDO2 authentication, specifically demonstrating authentication using a concatenation hybrid construction. While functional, relatively simple to implement, and seemingly providing quantum security on the surface, this specific hybridization method (and many others) lacks rigorous security analysis, leaving potential vulnerabilities unknown. Therefore, it should not automatically be assumed to be the preferred method for hybridizing FIDO2.

To facilitate the testing and public scrutiny necessary for cryptographic innovations, the FIDO Alliance could adopt multiple quantum-secure methods, both hybrid and pure, and allow users to opt in. This approach empowers users to manage their exposure to security risks while enabling real-world testing of quantum-secure FIDO2 solutions.

Although a SE is an important component of an authenticator, the lack of SE support for PQC algorithms does not mean that FIDO2 cannot begin the migration towards quantum security. PQC algorithms perform well in constrained environments, and the main hardware issue with using PQC on authenticators today is the absence of a SE, which exposes the authenticator to SCA. However, as mentioned previously, by giving users the ability to opt in or out of PQC FIDO2 authentication, they can personally decide what level of risk to accept.

With this respect for the end user in mind, and an intention to continuously monitor the effectiveness of PQC solutions, the FIDO Alliance is indeed ready to start the migration towards quantum security.

RQ2: What does a feasible quantum-secure solution for FIDO2 authentication entail?

Ultimately, a quantum-secure FIDO2 solution should offer a range of choices encompassing different PQC algorithms and hybridization methods. This variety is essential to accommodate diverse use cases with varying requirements for key and signature sizes, as well as signing speed. Insufficient flexibility in these options may result in an authenticator being compelled to adopt an algorithm or hybrid construction that proves impractical, potentially leading to its exclusion from PQC adoption and leaving it vulnerable to quantum attacks.

Moreover, all available PQC options must demonstrate the property of *proof composability*, i.e., that the algorithms included in a hybrid construction must be combined in such a way that it is possible to prove reduction to the security properties of each individual algorithm. It must also not be possible to perform a kind of "stripping" attack, wherein the verification process skips one of the composite algorithms. This could lead to only the classical algorithm being verified, but resulting in the entire hybrid signature passing as valid.

Lastly, regarding hybridization, the hybrid constructions selected by the FIDO Alliance should ideally be straightforward to implement on authenticators. This minimizes the risk of vulnerabilities arising from faulty implementation due to complexity – an occurrence not uncommon in the field of cryptography.

6.2 Conclusion

This work aimed to implement a PoC authenticator capable of performing hybrid PQC signatures for FIDO2 authentication. The implementation successfully performed this task, utilizing a hybrid construction consisting of the concatenation of a

classical signature, ECDSA, and a PQC signature, Dilithium. This particular type of hybridization proved relatively simple to implement while also performing well, and could be an option offered by a quantum-secure FIDO2 solution.

Although this work demonstrated a possibly feasible solution, further work must be done to ensure that the implemented solutions offer fundamental security properties. By initially implementing PQC solutions that are deemed secure and giving end-users the option to opt-in or out of their use, these solutions can be tested in the real world and offer PQC authentication for those willing to accept the risks of using a novel cryptographic system.

6.3 Future Work

To advance the findings presented in this study, it is recommended to build upon this work and explore the following components.

6.3.1 HID Implementation

CTAP messages are framed for USB transport using the HID protocol [FID18]. To fit this work in the given timeframe of 21 weeks, the authenticator implemented serial communication rather than HID, as it is easier. A Python script then translates the serial communication to HID. This Python intermediary may introduce noise and small delays, potentially making the benchmarks performed in this study not fully representative of a proper implementation using only HID. While the results are likely close to what a production-grade authenticator would yield, an implementation using HID would provide more scientifically accurate results.

6.3.2 Bluetooth

Some remote authenticators, such as the OFFPAD, use Bluetooth. To better understand how different algorithm choices perform based on the communication method used, it would be beneficial to implement PQC algorithms in a FIDO2 context on an authenticator using Bluetooth. Such a study could examine the relationship between key/signature sizes and key/signature generation times, and how these factors affect the overall authentication time. This relates to the discussion in Subsection 5.2.3.

6.3.3 Implementing and Attacking Hybrid Constructions

This work only implemented a concatenation hybridization. There are other methods of achieving hybridization, such as nested signatures whereby a message is first signed with a classical algorithm, after which this signature is signed with a PQC algorithm. The implementation and performance of these other hybrid constructions ought to

be tested. Furthermore, these implementations, in addition to the concatenation version in this work, should be deliberately attacked to find weaknesses such that they can be improved, or avoided completely.

References

[Alb24] M. R. Albrecht, *Malb/lattice-estimator: An attempt at a new lwe estimator*, Feb. 2024. [Online]. Available: https://github.com/malb/lattice-estimator?tab=readme-ov-file (last visited: Apr. 5, 2024).

[App23] Apple. "Passkeys". Accessed on: October 28, 2023. (2023), [Online]. Available: https://developer.apple.com/passkeys/.

[APS15] M. R. Albrecht, R. Player, and S. Scott, "On the concrete hardness of learning with errors", *Journal of Mathematical Cryptology*, vol. 9, no. 3, pp. 169–203, 2015.

[AWG+21] N. Alnahawi, A. Wiesmaier, *et al.*, "On the state of post-quantum cryptography migration", 2021.

[Bar99] M. Barr, *Programming embedded systems in C and C++*. " O'Reilly Media, Inc.", 1999.

[BBCW21] M. Barbosa, A. Boldyreva, *et al.*, "Provable security analysis of fido2", in *Advances in Cryptology – CRYPTO 2021*, T. Malkin and C. Peikert, Eds., Cham: Springer International Publishing, 2021, pp. 125–156.

[BCZ23] N. Bindel, C. Cremers, and M. Zhao, "Fido2, ctap 2.1, and webauthn 2: Provable security and post-quantum instantiation", in *2023 IEEE Symposium on Security and Privacy (SP)*, IEEE, 2023, pp. 1471–1490.

[BDK+21] S. Bai, L. Ducas, *et al.*, *CRYSTALS-Dilithium algorithm specifications and supporting documentation (version 3.1)*, Feb. 2021. [Online]. Available: https://pq-crystals.org/dilithium/data/dilithium-specification-round3-20210208.pdf#page=29&zoom=100,138,673.

[BH23] N. Bindel and B. Hale, "A note on hybrid signature schemes", *Cryptology ePrint Archive*, 2023.

[BHMS17] N. Bindel, U. Herath, *et al.*, "Transitioning to a quantum-resistant public key infrastructure", in *Post-Quantum Cryptography: 8th International Workshop, PQCrypto 2017, Utrecht, The Netherlands, June 26-28, 2017, Proceedings 8*, Springer, 2017, pp. 384–405.

[BLN16] D. J. Bernstein, T. Lange, and R. Niederhagen, "Dual ec: A standardized back door", in *The New Codebreakers: Essays Dedicated to David Kahn on the Occasion of His 85th Birthday*, Springer, 2016, pp. 256–281.

[Bor20] C. Bormann, Dec. 2020. [Online]. Available: https://www.rfc-editor.org/rfc/rfc8949.html.

[CMRR23] L. Chen, D. Moody, *et al.*, *Digital Signature Standard (DSS)*. Feb. 2023. [Online]. Available: http://dx.doi.org/10.6028/NIST.FIPS.186-5.

[Dig23] L. Digné, *Beyond bits: Securing the digital frontier with post-quantum cryptography*, Project report in TTM4502, Dec. 2023.

[dPPRS23] R. del Pino, T. Prest, *et al.*, "High-order masking of lattice signatures in quasilinear time", in *2023 IEEE Symposium on Security and Privacy (SP)*, IEEE, 2023, pp. 1168–1185.

[EEN+24] M. F. Esgin, T. Espitau, *et al.*, "Plover: Masking-friendly hash-and-sign lattice signatures", in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, Springer, 2024, pp. 316–345.

[Fed21] Federal Office for Information Security (BSI), *Migration to post-quantum cryptography*, https://www.bsi.bund.de/SharedDocs/Downloads/EN/BSI/Crypto/Migration_to_Post_Quantum_Cryptography.pdf?___blob=publicationFile&v=2, Accessed: 2024-05-19, 2021.

[FHK+18] P.-A. Fouque, J. Hoffstein, *et al.*, "Falcon: Fast-fourier lattice-based compact signatures over ntru", *Submission to the NIST's post-quantum cryptography standardization process*, vol. 36, no. 5, pp. 1–75, 2018.

[FID15] FIDO Alliance. "Fido signature format v2.0". Accessed on: February 7. (2015), [Online]. Available: https://fidoalliance.org/specs/fido-v2.0-ps-20150904/fido-signature-format-v2.0-ps-20150904.html.

[FID18] FIDO Alliance. "Fido client to authenticator protocol v2.0". Accessed on January 29, 2024. (2018), [Online]. Available: https://fidoalliance.org/specs/fido-v2.0-id-20180227/fido-client-to-authenticator-protocol-v2.0-id-20180227.html.

[FID23a] FIDO Alliance, *Fido alliance overview*, 2023. [Online]. Available: https://fidoalliance.org/overview/.

[FID23b] FIDO Alliance. "How fido works". (2023), [Online]. Available: https://fidoalliance.org/how-fido-works/.

[FS86] A. Fiat and A. Shamir, "How to prove yourself: Practical solutions to identification and signature problems", in *Conference on the theory and application of cryptographic techniques*, Springer, 1986, pp. 186–194.

[GKP+23] D. Ghinea, F. Kaczmarczyck, *et al.*, "Hybrid post-quantum signatures in hardware security keys", in *International Conference on Applied Cryptography and Network Security*, Springer, 2023, pp. 480–499.

[Goo23] Google. "Passwordless login with passkeys". Accessed on: October 28, 2023. (2023), [Online]. Available: https://developers.google.com/identity/passkeys.

[GSN+20] S. Ghorbani Lyastani, M. Schilling, *et al.*, "Is fido2 the kingslayer of user authentication? a comparative usability study of fido2 passwordless authentication", in *2020 IEEE Symposium on Security and Privacy (SP)*, 2020, pp. 268–285.

[GTJA17] B. B. Gupta, A. Tewari, *et al.*, "Fighting against phishing attacks: State of the art and future challenges", *Neural Computing and Applications*, vol. 28, pp. 3629–3654, 2017.

[Int24] Internet Assigned Numbers Authority, *IANA COSE algorithms*, Accessed: 2024-04-25, 2024. [Online]. Available: https://www.iana.org/assignments/cose/cose.xhtml#algorithms.

[JMM+22] D. Joseph, R. Misoczki, *et al.*, "Transitioning organizations to post-quantum cryptography", *Nature*, vol. 605, no. 7909, pp. 237–243, 2022.

[JMV01] D. Johnson, A. Menezes, and S. Vanstone, "The elliptic curve digital signature algorithm (ecdsa)", *International journal of information security*, vol. 1, pp. 36–63, 2001.

[KMWK23] M. Kepkowski, M. Machulak, *et al.*, "Challenges with passwordless fido2 in an enterprise setting: A usability study", 2023. [Online]. Available: https://arxiv.org/abs/2308.08096.

[Lin24] Linux Foundation, *Linux foundation*, Accessed: 2024-05-30, 2024. [Online]. Available: https://www.linuxfoundation.org/.

[LPR10] V. Lyubashevsky, C. Peikert, and O. Regev, "On ideal lattices and learning with errors over rings", in *Advances in Cryptology–EUROCRYPT 2010: 29th Annual International Conference on the Theory and Applications of Cryptographic Techniques, French Riviera, May 30–June 3, 2010. Proceedings 29*, Springer, 2010, pp. 1–23.

[Lyu20] V. Lyubashevsky, "Basic lattice cryptography: Encryption and fiat-shamir signatures", *IBM Research-Zurich, Säumerstrasse*, vol. 4, p. 8803, 2020.

[Mbe24] Mbed-TLS, *Mbed-tls/mbedtls: An open source, portable, easy to use, readable and flexible tls library*, Apr. 2024. [Online]. Available: https://github.com/Mbed-TLS/mbedtls.

[MP21] M. Mosca and M. Piani, "Quantum threat timeline report 2020", *Global Risk Insitute*, 2021.

[MR09] D. Micciancio and O. Regev, "Lattice-based cryptography", in *Post-quantum cryptography*, Springer, 2009, pp. 147–191.

[Nat22a] National Institute of Standards and Technology, "Nist announces first four quantum-resistant cryptographic algorithms", Jul. 2022. [Online]. Available: https://www.nist.gov/news-events/news/2022/07/nist-announces-first-four-quantum-resistant-cryptographic-algorithms.

[Nat22b] National Security Agency, *Announcing the commercial national security algorithm suite 2.0*, Sep. 2022. [Online]. Available: https://media.defense.gov/2022/Sep/07/2003071834/-1/-1/0/CSA_CNSA_2.0_ALGORITHMS_.PDF.

[Nat22c] National Security Agency, *The commercial national security algorithm suite 2.0 and quantum computing faq*, Sep. 2022. [Online]. Available: https://media.defense.gov/2022/Sep/07/2003071836/-1/-1/1/CSI_CNSA_2.0_FAQ_.PDF.

[Nat23] National Institute of Standards and Technology. "Nist to standardize encryption algorithms that can resist attack by quantum computers". Accessed on: November 6, 2023. (2023), [Online]. Available: https://www.nist.gov/news-events/news/2023/08/nist-standardize-encryption-algorithms-can-resist-attack-quantum-computers.

[Nor24] Nordic Semiconductor, *Nordicsemiconductor/zcbor: Low footprint c/c++ cbor library and python tool providing code generation from cddl descriptions.* Apr. 2024. [Online]. Available: https://github.com/NordicSemiconductor/zcbor.

[Ope] Open Quantum Safe, *Crystals-dilithium.* [Online]. Available: https://openquantumsafe.org/liboqs/algorithms/sig/dilithium.html.

[Ope24] Open Quantum Safe, *Open-quantum-safe/liboqs: C library for prototyping and experimenting with quantum-resistant cryptography*, Apr. 2024. [Online]. Available: https://github.com/open-quantum-safe/liboqs.

[Pon] Pone Biometrics, *Product Description*, https://offpad.ponebiometrics.com/, Accessed: October 20, 2023.

[PSM+24] M. Prorock, O. Steele, *et al.*, *Ml-dsa for jose and cose*, Jan. 2024. [Online]. Available: https://datatracker.ietf.org/doc/draft-ietf-cose-dilithium/.

[SAT24] G. Selander, D. Atkins, and S. Turner, *Cbor object signing and encryption (cose)*, Apr. 2024. [Online]. Available: https://www.iana.org/assignments/cose/cose.xhtml#algorithms.

[Sch17a] J. Schaad, *Rfc 8152: Cbor object signing and encryption (cose)*, Jul. 2017. [Online]. Available: https://datatracker.ietf.org/doc/html/rfc8152#section-13.1.

[Sch17b] J. Schaad, *Rfc 8152: Cbor object signing and encryption (cose)*, Jul. 2017. [Online]. Available: https://datatracker.ietf.org/doc/html/rfc8152#section-8.1.

[SH23] T. Silde and T. Hagen, "Secure authentication with fido, biometrics and security keys", in *Sikkerhetsfestivalen 2023*, Received on: Date (September 1, 2023), 2023.

[Sho99] P. W. Shor, "Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer", *SIAM review*, vol. 41, no. 2, pp. 303–332, 1999.

[SSKC14] M. A. Sasse, M. Steves, *et al.*, "The great authentication fatigue – and how to overcome it", in *Cross-Cultural Design*, P. L. P. Rau, Ed., Cham: Springer International Publishing, 2014, pp. 228–239.

[Tak22] D. Taku. "Beware mfa fatigue". Accessed on: October 22, 2023. (2022), [Online]. Available: https://www.rsa.com/multi-factor-authentication/beware-mfa-fatigue/.

[TSZ22] T. G. Tan, P. Szalachowski, and J. Zhou, "Challenges of post-quantum digital signing in real-world applications: A survey", *International Journal of Information Security*, vol. 21, no. 4, pp. 937–952, 2022.

[Uni24] United Nations, *Infrastructure and industrialization - united nations sustainable development*, Accessed: 2024-05-27, 2024. [Online]. Available: https://www.un.org/sustainabledevelopment/infrastructure-industrialization/.

[Web23] WebAuthn. "Introducing public key cryptography and web authentication (webauthn)". Accessed on: November 1, 2023. (2023), [Online]. Available: https://webauthn.guide/.

[Wor22] World Wide Web Consortium. "Web authentication: An api for accessing public key credentials - level 2". Accessed on: Sunday 9th June, 2024. (2022), [Online]. Available: https://www.w3.org/TR/webauthn-2/.

[Yub23] Yubico. "Solutions". Accessed on: October 28, 2023. (2023), [Online]. Available: https://www.yubico.com/solutions/.

[Yub24a] Yubico. "Webauthn client authentication". Accessed on: February 7, 2024. (2024), [Online]. Available: https://developers.yubico.com/WebAuthn/WebAuthn_Developer_Guide/WebAuthn_Client_Authentication.html.

[Yub24b] Yubico. "Webauthn client registration". Accessed on: February 7, 2024. (2024), [Online]. Available: https://developers.yubico.com/WebAuthn/WebAuthn_Developer_Guide/WebAuthn_Client_Registration.html.

[Zep24a] Zephyr Project, *Getting started guide*, May 2024. [Online]. Available: https://docs.zephyrproject.org/latest/develop/getting_started/index.html.

[Zep24b] Zephyr Project, *Zephyr project*, Accessed: 2024-02-30, 2024. [Online]. Available: https://www.zephyrproject.org/.

[Øde24] A. T. Ødegaard, *Nsms kryptografiske anbefalinger*, Mar. 2024. [Online]. Available: https://nsm.no/fagomrader/digital-sikkerhet/kryptosikkerhet/kryptografiske-anbefalinger.

Appendix A
Benchmarks

A.1 Key Generation Times

Table A.1: Key generation times (in ms) for the different algorithms. Here, Dil-2 = Dilithium-2, Fal-512 = Falcon-512.

Iteration	ES256	Dil-2	Fal-512	Hybrid (ES256/Dil-2)	Hybrid (ES256/Fal-512)
1	939.6	48.6	2931.4	996.4	7350.6
2	941.0	48.3	4148.2	996.1	3367.2
3	940.8	49.4	3980.2	995.5	4241.9
4	940.4	48.6	4199.7	996.6	5881.3
5	942.2	48.6	3050.4	997.0	6028.6
6	941.3	48.6	4436.6	996.9	4595.1
7	938.6	49.3	4319.6	995.2	5736.6
8	939.9	48.2	3100.5	995.3	3625.6
9	940.7	47.0	4034.4	995.7	5097.4
10	941.8	49.4	3558.3	994.6	7112.5
11	942.6	47.9	2695.0	994.1	5028.0
12	938.8	47.5	3337.3	995.0	3886.9
13	938.2	49.0	7682.0	997.5	5815.2
14	941.6	48.6	2576.5	998.1	3626.3
15	940.9	48.2	2577.0	994.2	7309.5
16	941.5	49.3	3507.3	995.6	4649.7
17	941.6	48.6	2931.5	997.2	3745.1
18	939.9	48.6	6973.1	994.8	3624.3
19	938.2	47.8	5484.4	997.4	5147.5
20	940.7	49.0	2576.6	996.8	3366.6
21	942.6	48.2	5468.9	996.5	12955.2
22	938.4	48.2	2695.0	996.5	4620.6
23	940.7	47.1	3286.3	996.9	5668.8
24	940.6	49.0	4910.0	995.2	7374.4

Continued on next page

Table A.1 – continued from previous page

Iteration	ES256	Dil-2	Fal-512	Hybrid (ES256/Dil-2)	Hybrid (ES256/Fal-512)
25	941.8	49.4	5247.9	998.2	3363.9
26	939.8	48.9	12133.9	994.0	3745.3
27	939.9	47.8	5247.9	994.9	3598.4
28	940.2	49.0	4266.1	996.8	7898.5
29	940.7	48.6	2813.0	993.1	6547.9
30	941.2	48.6	2695.0	996.2	3364.3
31	939.9	49.0	3623.9	993.0	5883.3
32	938.0	49.3	3454.8	996.6	4123.4
33	940.2	48.5	10050.5	997.5	3746.5
34	941.2	48.6	3152.4	995.4	8420.2
35	940.8	47.8	5703.4	994.8	4794.3
36	938.4	48.2	3152.3	995.2	4505.4
37	941.0	48.6	2695.1	996.0	9327.6
38	939.3	49.0	7261.6	999.6	6234.7
39	939.1	49.7	3963.7	996.1	7115.8
40	938.9	48.6	2577.0	995.4	3889.0
41	940.0	47.8	3675.8	997.8	3625.2
42	939.7	48.6	2577.0	996.8	6261.1
43	940.8	48.6	2577.0	995.4	4007.6
44	939.3	48.5	8952.1	996.5	8632.4
45	939.3	47.8	2576.6	996.3	4100.4
46	940.4	48.2	3218.9	995.4	4623.3
47	937.4	48.2	3964.1	997.5	4624.4
48	940.7	48.3	5028.5	997.7	5002.4
49	939.2	48.2	2576.9	997.3	4909.9
50	940.5	49.0	5873.4	995.9	4650.5

A.2 Signing Times

Table A.2: Signature generation times (in ms) for the different algorithms.

Iteration	ES256	Dil-2	Fal-512	Hybrid (ES256/Dil-2)	Hybrid (ES256/Fal-512)
1	918.2	128.6	1061.7	982.3	2072.0
2	919.1	105.3	1058.0	1114.5	2071.3
3	920.5	231.1	1063.1	1022.3	2071.6
4	920.6	262.5	1062.7	1003.0	2069.4
5	919.8	128.5	1058.0	1051.4	2071.2
6	920.9	105.4	1057.1	1051.9	2071.6
7	918.7	64.3	1057.7	1050.7	2070.2

Continued on next page

Table A.2 – continued from previous page

Iteration	ES256	Dil-2	Fal-512	Hybrid (ES256/Dil-2)	Hybrid (ES256/Fal-512)
8	920.6	282.9	1057.1	983.7	2072.0
9	918.9	149.1	1059.4	1051.8	2071.7
10	921.8	175.0	1057.8	983.7	2072.1
11	920.5	64.3	1058.9	1064.5	2074.1
12	922.7	198.2	1060.6	982.4	2066.4
13	922.2	64.4	1059.4	1264.8	2069.8
14	918.4	192.8	1060.6	981.7	2071.6
15	922.1	87.7	1065.3	1022.5	2070.1
16	921.2	321.3	1058.6	1048.4	2072.0
17	920.4	64.3	1057.2	1110.7	2068.2
18	918.6	110.8	1062.6	986.2	2069.2
19	920.2	233.7	1058.9	1092.7	2067.8
20	920.4	128.6	1061.5	984.4	2069.6
21	920.0	87.5	1062.7	1066.2	2067.7
22	920.3	105.2	1064.2	1004.9	2076.7
23	918.9	190.0	1055.6	1026.1	2073.0
24	919.6	198.2	1058.2	1048.3	2072.2
25	922.0	64.3	1056.5	1163.9	2070.9
26	919.5	84.8	1056.5	982.8	2069.9
27	919.2	64.4	1058.1	1005.3	2073.2
28	917.7	64.3	1056.4	984.0	2069.7
29	924.4	84.9	1059.6	1005.0	2072.5
30	918.0	303.6	1055.4	1091.1	2074.8
31	919.6	84.9	1058.5	984.0	2074.6
32	918.0	84.8	1059.0	1154.0	2069.3
33	922.0	64.3	1062.3	1047.7	2066.3
34	921.3	64.4	1061.6	1002.4	2071.1
35	920.7	84.8	1064.5	983.4	2072.0
36	919.4	131.3	1060.8	1094.3	2071.8
37	920.4	108.1	1058.2	1247.6	2076.0
38	919.3	110.8	1058.2	1007.6	2072.6
39	919.4	131.4	1060.4	1068.9	2073.2
40	918.6	105.3	1061.7	1007.2	2071.7
41	920.3	64.3	1058.6	1004.2	2072.0
42	921.0	131.1	1060.1	1072.8	2072.4
43	919.9	175.1	1058.9	1072.1	2073.1
44	919.3	64.3	1060.7	1090.7	2072.3
45	921.7	108.2	1060.0	982.6	2072.8
46	920.5	128.5	1059.9	1218.0	2068.4

Continued on next page

Table A.2 – continued from previous page

Iteration	ES256	Dil-2	Fal-512	Hybrid (ES256/Dil-2)	Hybrid (ES256/Fal-512)
47	918.3	64.4	1060.6	1002.3	2066.2
48	918.8	172.3	1061.8	1089.7	2071.0
49	920.6	64.4	1053.5	1066.1	2070.8
50	921.5	333.5	1059.5	1004.9	2070.8

A.3 Signature Sizes

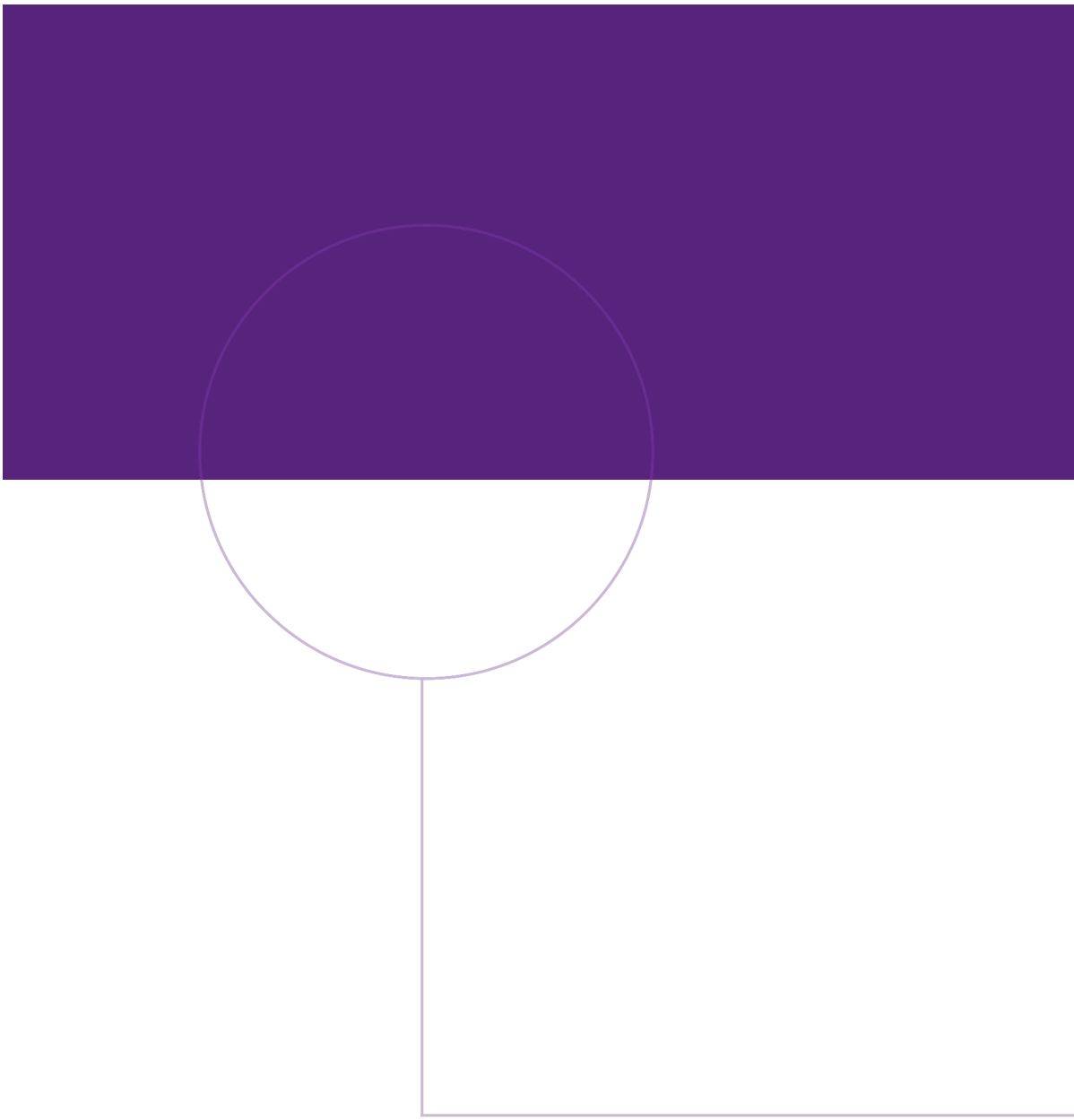
Table A.3: Signature sizes (in bytes) for ES256 and Falcon-512, the only two composite algorithms in this work with varying signature sizes.

Iteration	ES256	Falcon-512
1	71	657
2	71	655
3	71	658
4	72	660
5	72	652
6	71	658
7	71	657
8	70	660
9	71	657
10	70	658
11	72	652
12	71	653
13	71	652
14	71	653
15	72	652
16	71	657
17	71	656
18	72	654
19	71	657
20	71	657
21	71	655
22	72	657
23	71	657
24	71	652
25	72	656
26	71	655
27	72	657
28	71	659

Continued on next page

Table A.3 – continued from previous page

Iteration	ES256	Falcon-512
29	72	654
30	71	654
31	70	652
32	70	656
33	70	655
34	71	654
35	71	653
36	72	659
37	72	655
38	72	654
39	70	654
40	70	655
41	70	659
42	71	652
43	71	657
44	71	656
45	71	651
46	71	652
47	71	657
48	70	654
49	71	653
50	71	653



NTNU
Norwegian University of
Science and Technology

PONE
BIOMETRICS