UNIVERSITY OF BIRMINGHAM

School of Computer Science
College of Engineering and Physical Sciences

MSc. Project

Implementing FIDO2 authentication protocol with NRF52840 SoC

Submitted in conformity with the requirements
for the degree of MSc. Cyber Security
School of Computer Science
University of Birmingham

Rodrigo Esparza-Sanchez, BSc.
Student ID: 1897741
Supervisor: Dr Mark Ryan

September 2019

Declaration

The material contained within this report has not previously been submitted for a degree at the University of Birmingham or any other university. The research reported within this report has been conducted by the author unless indicated otherwise.

Signed .....

Rodrigo Esparza-Sanchez

Acknowledments

"I would like to thank and recognize all the people who where together with me in this tough, but exciting adventure of a MSc Cyber Security. To the profesors which in each lecture/assignment helped me to understand deeply all topics. To my suppervisor Mark Ryan, who always pushed me and guided me to give the best of me on this project. To my friends who where in the same path as me during this year, having tough but also memorable times. To my father who always encourages me to overcome my self. To my mother who always is at my side supporting me in anything I try. And especially I want to thank my wife and children for supporting me so much despite having to be away for so long."

"Passwords are like underwear: they need to be changed often, it is not a good idea to share them with friends, the longer they are the better they are, and its better for them to be mysterious and not left lying around."

University of Michigan

Abstract

For a long time until nowadays, users have used username and password combination as the main online authentication mechanism. However, this single factor authentication mechanism has several security drawbacks such as leakage, reuse, poor entropy and hard to remember, in addition to very common attacks which allow stealing this sensitive information, for instance, brute force, phishing and man in the middle attacks. Recently, the FIDO2 authentication protocol was published as a standard. FIDO2 uses signatures to provide proof of having a secret without revealing it. Besides, It can provide a passwordless experience which will help to improve authentication usability. Increasing security and usability will help to reduce the risks associated with the authentication process. However, FIDO2 is a very recent standard, and as every standard, it takes time to be adopted, and it requires support from many corners. Therefore, this project is trying to contribute to the FIDO2 adoption by developing an open-source FIDO2 authenticator project using a promising nRF52840 SoC. Although the result of this project is a small contribution of a basic implementation of FIDO2, it sets the path to create a robust open-source FIDO2 authenticator.

MSc. Project
Implementing FIDO2 authentication protocol with NRF52840 SoC

Rodrigo Esparza-Sanchez

Contents

Table of Abbreviations

1 Introduction **1**
1.1 Motivation . . . 1
1.2 Aim . . . 1
1.3 Contributions . . . 1

2 Background **2**
2.1 Authentication . . . 2
2.1.1 Authentication factors . . . 2
2.2 Cryptography . . . 2
2.2.1 Security goals . . . 2
2.2.2 Encryption . . . 2
2.2.3 Decryption . . . 2
2.2.4 Symmetric-key cryptography . . . 3
2.2.5 Public-key cryptography . . . 4
2.2.6 Hash functions . . . 5
2.2.7 Random number generator . . . 5
2.3 Trusted Execution Environment . . . 6
2.3.1 TrustZone . . . 6
2.4 USB HID . . . 6
2.5 nRF52840 SoC . . . 7

3 Related work **8**
3.1 U2F . . . 8
3.2 UAF . . . 8
3.3 Makerdiary nRF52-U2F . . . 8
3.4 Solo . . . 9
3.5 Yubikey . . . 9

4 Analysis and Design **10**
4.1 FIDO2 . . . 10
4.1.1 Protocol Overview . . . 10
4.1.2 Design considerations . . . 12
4.2 Threat model . . . 15
4.2.1 Threats and countermeasures . . . 15

5 Implementation **19**
5.1 Development environment setup . . . 19
5.1.1 Hardware environment requirements . . . 19
5.1.2 Software environment requirements . . . 21
5.1.3 Setup process . . . 21
5.2 Code structure . . . 24
5.3 Testing process . . . 25
5.4 CTAP2 implementation . . . 26
5.4.1 Message encoding . . . 27
5.4.2 User presence . . . 28
5.4.3 Authenticator API . . . 29
5.4.4 getInfo API . . . 29
5.4.5 makeCredential API . . . 30
5.4.6 getAssertion API . . . 36

6 Results and Evaluation **40**
6.1 Python-FIDO2 . . . 40
6.2 Webauthn.io . . . 40
6.3 Webauthn.me . . . 41
6.4 Google . . . 41
6.5 GitHub . . . 42

7 Future work **43**
7.1 CTAP2 API . . . 43
7.1.1 Get Next Assertion . . . 43
7.1.2 Reset . . . 43
7.1.3 Client Pin . . . 43
7.2 NFC and Bluetooth . . . 43
7.3 HMAC Secret . . . 43
7.4 Secure Firmware Update . . . 43

References **44**

8 Appendix **47**
8.1 Appendix A: Git Repository Location . . . 47
8.2 Appendix B: Git Project Structure . . . 47
8.3 Appendix C: Running the software . . . 47

Table of Abbreviations

- AAGUID – Authenticator Attestation Global Unique Identifier
- AES – Advanced Encryption Standard
- API – Application Programming Interface
- ARM – Advanced RISC Machine
- BLE – Bluetooth Low Energy
- CBOR – Concise Binary Object Representation
- COSE – CBOR Object Signing and Encryption
- CTAP(2) – Client to Authenticator Protocol (2)
- CTR – Counter block cipher mode
- DFU – Device Firmware Update
- (D)DOS – (Distributed) Denial of Service attack
- ECC – Elliptic Curve Cryptography
- FIDO(2) – Fast Identity Online (2)
- GCC – GNU Compiler Collection
- GNU – GNU's Not Unix
- HID – Human Interface Device
- IANA – Internet Assigned Numbers Authority
- I/O – Input and Output
- IV – Initialisation Vector
- LED – Light-Emitting Diode
- MFA – Multi-Factor Authentication
- NFC – Near-Field Communication
- PC – Personal Computer
- NIST – National Institute of Standars and Technology
- PIN – Personal Identification Number
- PRIV – Private
- PRNG – Pseudo Random Number Generator
- PUB – Public
- REE – Rich Execution Environment
- RGB – Red Green and Blue
- RISC – Reduced Instruction Set Computing
- RNG – Random Number Generator
- RP – Relying Party
- rpId – Relying Party Identifier
- SHA – Secure Hash Algorithm
- SFA – Single-Factor Authentication
- SoC – System-on-a-Chip
- TEE – Trusted Execution Environment
- TRNG – True Random Number Generator
- U2F – Universal 2^{nd} Factor
- UAF – Universal Authentication Framework
- USB – Universal Serial Bus
- W3C – World Wide Web Consortium

1 Introduction

1.1 Motivation

For a long time until nowadays, users have used username and password combination as the main online authentication mechanism. However, this single factor authentication mechanism has several security drawbacks such as leakage, reuse, poor entropy and hard to remember, in addition to very common attacks which allow stealing this sensitive information, for instance, brute force, phishing and man in the middle attacks. Therefore, implement stronger authentication mechanisms based on standards as FIDO2 protocol that increases authentication security and usability, as well as mitigates those previous attacks, with the support of hardware security anchors, will improve the security of users, systems and data.

1.2 Aim

Develop an open-source FIDO2 authenticator with the nRF52840 SoC.

1.3 Contributions

This project contributes to extend the adoption of the FIDO2 protocol, and therefore, helping to reduce the security risks of current threats that users and organizations face when they require to perform an authentication process.

2 Background

This section describes briefly the topics required to understand the details of the project and this report.

2.1 Authentication

Digital authentication is the process to verify the digital identity through one or more authenticator factors Grassi et al. (2017). For instance, an online service verifying the users identity with a username and password that only the user knows.

2.1.1 Authentication factors

A user who wants to give proof of its digital identity can use three different factors(Grassi et al. 2017, Hallsteinsen et al. 2007, Kim & Hong 2011, Ting et al. 2015):

- *Knowledge:* something that the user knows (e.g. a PIN, a password).
- *Ownership:* something that the user has (e.g. hardware token, mobile phone, cryptographic key).
- *Inherence:* something that the user is (e.g. fingerprint, face, iris, voice).

When only one factor is used to authenticate a user, this is known as a single-factor authentication (SFA) and when two or more factors are used it is known as multi-factor authentication (MFA). Two-factor authentication is a widely multi-factor authentication used. An authentication system is more robust as more authentication factors are used (Grassi et al. 2017, Hallsteinsen et al. 2007, Kim & Hong 2011).

2.2 Cryptography

Cryptography is the study area that design and analyses mathematical techniques to secure communications from malicious adversaries (Hankerson & Menezes 2011).

2.2.1 Security goals

The security goals (Hankerson & Menezes 2011) of cryptography are:

- Confidentiality: the data must remain secret except for those to whom it is intended.
- Integrity: protect the data from unauthorized modifications.
- Authentication: verifies the identity of the sender and receiver.
- Non-repudiations: prevents an entity to deny her actions.

2.2.2 Encryption

Encryption is the process that takes a plaintext and a key as input and converts the plaintext into a ciphertext which is the output (Aumasson 2017). The ciphertext contains the plaintext but only can be recovered with the decryption process and the right key.

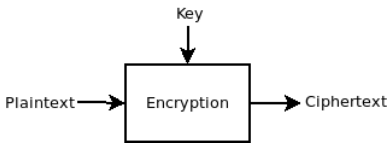

Figure 2.2.1: *Graphical view of encryption process*

2.2.3 Decryption

Decryption is the reverse process of the encryptions. It takes a ciphertext and a key as input and converts the ciphertext into a plaintext which is the output (Aumasson 2017).

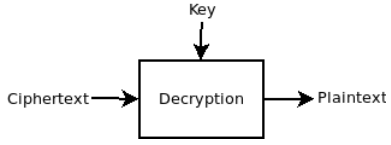

Figure 2.2.2: *Graphical view of decryption process*

2.2.4 Symmetric-key cryptography

Symmetric-key cryptography analyses and designs the encryption schemes that use the same key for encryption and decryption process (Aumasson 2017).

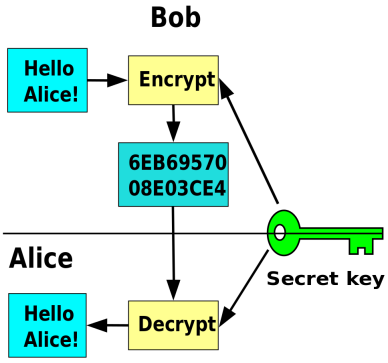

Figure 2.2.3: *Graphical view of symetric-key encryption and decryption*

Advanced Encryption Standard

AES is the symmetric-key cipher algorithm which became a NIST standard in 2001 (FIPS 2009). Currently is the most common symmetric-key cipher used in the world (Aumasson 2017).

AES can process 128-bit data blocks using a 128, 192 or 256 bits key (FIPS 2009).

Counter block cipher mode A symmetric-key cypher encrypts or decrypts a fixed and small chunk of data known as a block. For instance, AES process a 128-bit block. To perform the encryption/decryption of a variable amount of data larger than a block is necessary to combine the cypher algorithm with a block cypher mode (Aumasson 2017).

The Counter block cipher mode (CTR) is a block cipher mode that uses a nonce and a counter to perform the encryption and decryption process.

The CTR encryption process takes as input a nonce and a counter concatenated and a key. A nonce is a number used only once. The counter is incremented by one for each processed block. The nonce and the counter concatenated are processed by the encryption process from the underlying block cipher algorithm using the key, for example, AES. This produces a ciphertext of the nonce and counter concatenation, which then is XORed with the plaintext block and the result is the final ciphertext block. This process is repeated for each block incrementing the counter (Dworkin 2001).

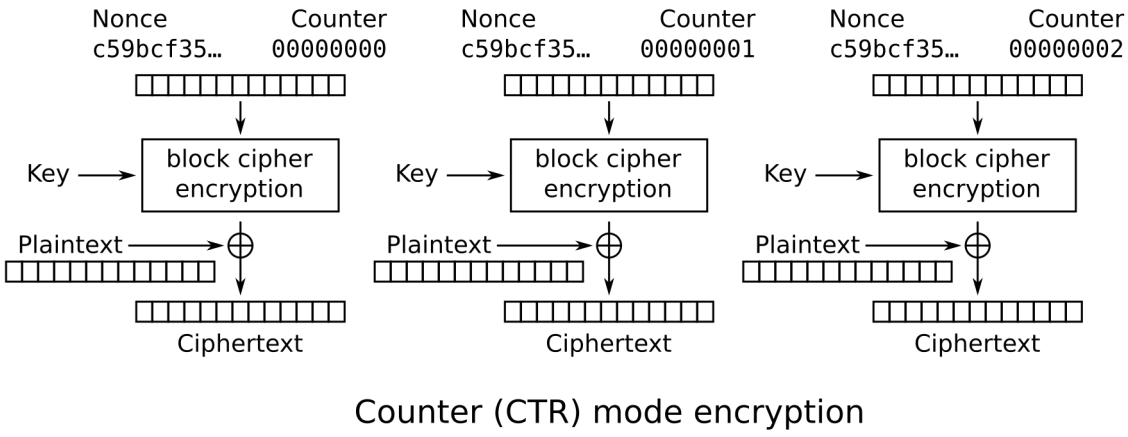

Figure 2.2.4: *Graphical view of CTR mode encryption. (Wikipedia contributors 2019a)*

The CTR decryption process takes as input the same nonce and counter used for encryption. The nonce and counter are concatenated and processed by the encryption process from the underlying block cipher using the key. This produces a ciphertext of the nonce and counter concatenation, which then is XORed with the ciphertext block and the result is the original plaintext block Dworkin (2001).

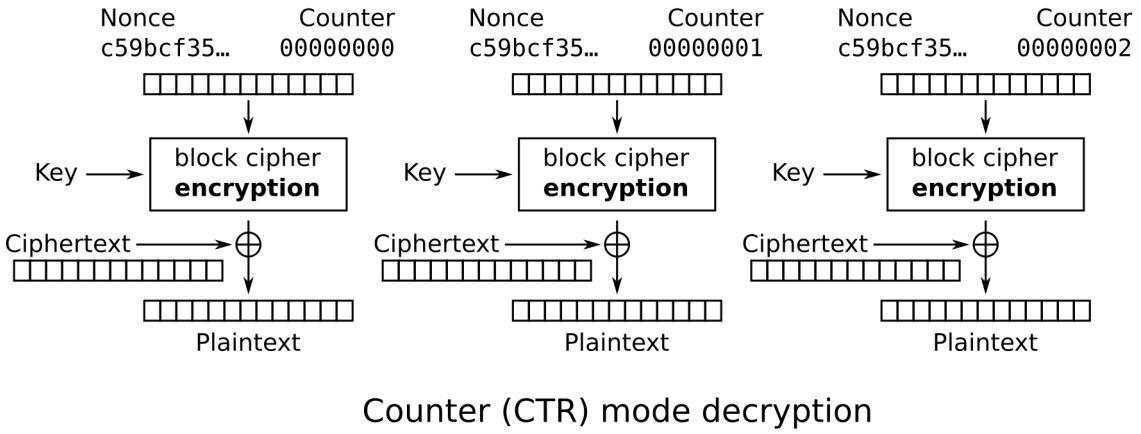

Figure 2.2.5: *Graphical view of CTR mode decryption (Wikipedia contributors 2019a)*

It is important to note that for both the encryption and decryption process, the underlying block cypher only makes use of the encryption process in both cases.

2.2.5 Public-key cryptography

Public-key cryptography analyses and designs the encryption schemes that use a key for encryption and a different key for decryption process (Aumasson 2017).

The key used to encrypt is known as the public key and it is considered that could be public available. The key used to decrypt is known as private key and must remain secret. The public key can be computed from the private key (Aumasson 2017).

Some commonly used public-key cryptographic schemes are (Hankerson & Menezes 2011):

- The RSA cryptosystem, which relies on the integer factorization problem.
- ElGamal cryptosystem, which relies on the discret logarithm problem.
- Elliptic Curve cryptosystem, which relies in the elliptic curve discret logarithm problem.

Elliptic curve cryptography ECC was introduced in 1985 and its a more complex cryptosystem, however, it provides a greater security and its more efficient than RSA. For example, a ECC with 256-bit key is stronger than RSA with 4096-bit key. ECC relies in the elliptic curve discret logarithm problem (Aumasson 2017).

Key generation Key generation is the process to create a public-key pair, consisting of a private key and its respective public key (Aumasson 2017). ECC has its key generation algorithm (Hankerson & Menezes 2011).

Signature A digital signature is a cryptographic proof of having a private key without revealing it. Therefore, if a signature is created for a message, it is possible to verify that signature and hence the authenticity of the message and the sender (Aumasson 2017).

A signature scheme consists in the following algorithms (Hankerson & Menezes 2011):

1. A key generation.
2. A signature generation, which produces a signature using the private key.
3. A signature verification, which verifies a signature using the public key.

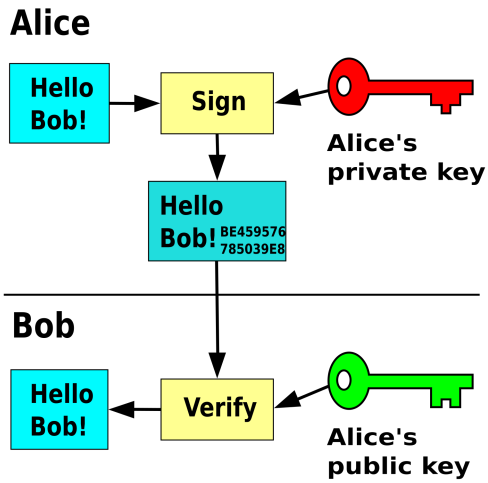

Figure 2.2.6: *Graphical view of digital signature generation and verification process (Wikipedia contributors 2019*b*)*

2.2.6 Hash functions

A hash function is a cryptographic function that takes an arbitrary input length and produces a fixed output length (Sobti & Geetha 2012). Hash functions are used for many security goals such as message integrity, message authentication and digital signatures.

A hash function has the following properties (Sobti & Geetha 2012):

- The input is of any length. Although, in practice, it has a limitation of huge size.
- The output is of fixed length.
- It is easy to compute for any message.
- Given a hash, it is computationally infeasible to find the message that produces it.
- It is computationally infeasible to find two different messages that produce the same hash.

SHA-2 SHA-2 is part of the SHA family functions. SHA family was developed by the NIST under the same principle of MD4 (except for SHA-3). SHA-2 can produce an output of 256 or 512 bits known as SHA-256 and SHA-512 respectively. Attacks on previous versions of SHA family have been reported, but no for SHA-2 or SHA-3 (Sobti & Geetha 2012).

2.2.7 Random number generator

All cryptosystems rely on randomness to be secure. This is generated from:

1. a source of entropy.
2. a pseudorandom number generator which produces high-quality random bits.

Some source of entropy may come from measurement of temperature, acoustic noise, air turbulence or electric static, however, those are not always available or produce random bits slowly (Aumasson 2017).

Pseudorandom number generator Pseudorandom number generators are algorithms that produce fast and high-quality random bits from a few random bits generated from an analogic source (Sunar et al. 2006).

True random number generator True random number generators are a piece of hardware dedicated to measuring the analogic world to produce true random bits that are used as a seed by the PRNG and then high-quality random numbers generated Sunar et al. (2006).

2.3 Trusted Execution Environment

A trusted execution environment (TEE) comes in an additional hardware module or as part of the main CPU providing a tamper-resistant isolated environment with its CPU, memory, data storage, I/O and code Sabt et al. (2015). Besides, to establish a Root of Trust, TEE has built-in cryptographic capabilities in software or hardware accelerators such as key generation, symmetric and asymmetric encryption and decryption, integrity, authentication, truly random number generation, and remote attestation Arfaoui et al. (2014), Ekberg et al. (2014).

2.3.1 TrustZone

TrustZone is the ARM implementation of the TEE into an ARM processor. ARM processors are widely implemented in smartphones and embedded devices. With the increased use of smartphones and IoT devices and the inclusion of ARM processors in servers, TrustZone is a widely deployed TEE technology nowadays.

TrustZone divides the execution environment into two worlds:

- The normal world, where the REE is executed.
- The secure world, where a TEE-kernel is executed.

The normal world cannot access the secure world. The normal world and secure world are switched using a secure monitor.

The TrustZone also divides the memory into a secure and normal memory, to provide memory protection. I/O devices can be assigned to the secure world or the normal world, to provide a secure path to those devices (Li et al. 2019).

Cryptocell Cryptocell is an ARM TrustZone family which provides a security solution to embedded devices with low power consumption (ARM Developer 2019).

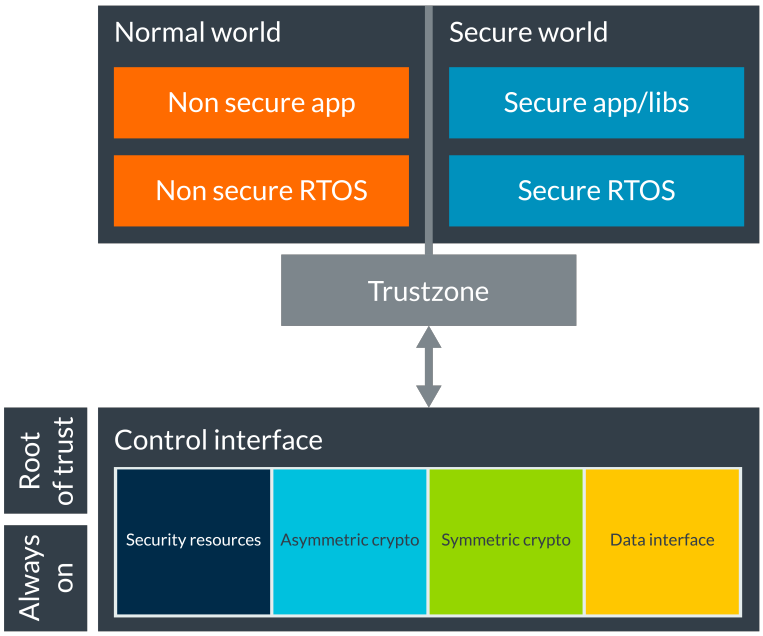

Figure 2.3.1: *ARM TrustZone Cryptocell architecture (ARM Developer 2019)*

2.4 USB HID

USB is a communication architecture which allows interconnecting a variety of devices with a wire cable. USB HID is a class driver specification which allows USB devices which have interaction with humans communicate with other devices such as a personal computer. Some common devices are keyboards, mouse, joysticks (Bus 2001).

2.5 nRF52840 SoC

The nRF52840 is a SoC ideal for IoT devices with low power consumption. Its more relevant features for this project are:

- a 64 MHz Arm Cortex-M4.
- an ARM TrustZone Cryptocell 310 security subsystem.
- 1 MB Flash.
- 256 KB RAM.
- Full-speed 12 Mbps USB.
- Bluetooth 5 multiprocol radio
- NFC-A tag

The nRF52840 SoC is manufactured by Nordic Semiconductor. It provides a complete nRF52 SDK to build software for this device (Nordic Semiconductor 2019*c*).

It also can be found in a USB dongle form factor that it is very convenient for this project.

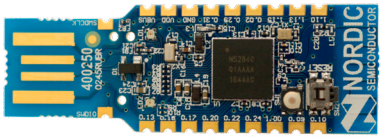

Figure 2.5.1: *nRF52840 USB Dongle (ARM Developer 2019)*

3 Related work

This section describes briefly work that has been done before and it is directly related to this project.

3.1 U2F

The U2F protocol from FIDO Alliance provides strong second-factor authentication to online services simplifying password complexity from online service. The user interacts with a U2F device simply pressing a button on a USB device or tapping an NFC device (Alliance 2017*b*).

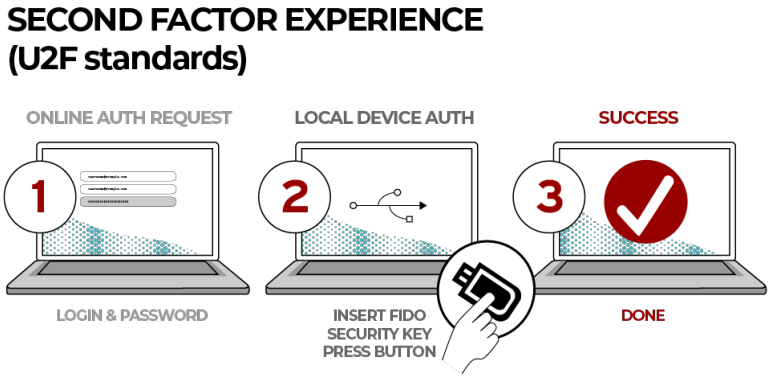

Figure 2.5.1: *FIDO U2F experience (Alliance, FIDO 2019c)*

3.2 UAF

The UAF protocol from FIDO Alliance empowers users to authenticate on online services using password-less and multi-factor authentication. The users device is registered on the online service using a local authentication method that can be any factor described above, including multi-factor options. Afterwards, the user does not require to enter his online password again to authenticate when it is done from this device (Alliance 2017*a*).

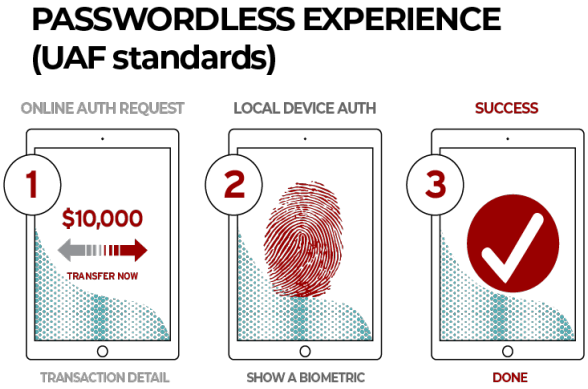

Figure 2.5.1: *FIDO UAF experience (Alliance, FIDO 2019c)*

U2F and UAF protocols are superseded by the FIDO2 protocol that is implemented within this project. FIDO2 combines both experiences provided by their predecessors U2F and UAF (Alliance, FIDO 2019*c*).

3.3 Makerdiary nRF52-U2F

The nRF52 FIDO U2F Security Key is an open-source project implementing the FIDO U2F protocol into the nRF52840 SoC using the nRF52 SDK. Its implementation leverages the nRF52840 USB and the ARM TrustZone Cryptocell-310 security subsystem capabilities (Makerdiary 2019*a*).

It is important to mention that the nRF52 FIDO U2F Security Key project was used as the code base for the FIDO2 implementation of the project described in this report.

3.4 Solo

Solo is an open-source security key developed by SoloKeys which supports FIDO2 and U2F protocols over USB and NFC (SoloKeys 2019).

Solo uses the STM32L432 microcontroller which has the following features:

- True random number generation to guarantee random keys.
- Security isolation so only simple and secure parts of code can handle keys.
- Flash protection from both external use and untrusted code segments.
- 256 KB of memory.

Advantages and disadvantages of Solo project against the nRF52840 FIDO2 from this project report:

- Advantages:
  - A compliant FIDO2 implementation over USB and NFC.
  - HMAC-Secret extension implemented.
  - ClientPIN implemented.
  - Is a mature project with more than 2 years of work and with 19 contributors.
- Disadvantages:
  - The STM32L432 used in Solo project does not support Bluetooth.
  - The STM32L432 used in Solo project does not have a cryptographic hardware accelerator.

3.5 Yubikey

YubiKey is a commercial hardware security token built by Yubico company. Different YubiKey models support different authentication protocols such as One-Time Password (OTP), PIV, U2F, Smart Card, OATH HOTP and FIDO2 (Künnemann & Steel 2012).

Yubico is one of the biggest drivers of FIDO2 protocol, and currently has a variety of security tokens devices offering a certified FIDO2 implementation (Yubico 2019*b*). However, its implementation is closed. Although, Yubico has developed a few open-source tools to increase the adoption of FIDO2 (Yubico 2019*a*).

4 Analysis and Design

This section analyzes the operation, rules and cryptographic primitives of the FIDO2 authentication protocol. At the same time, during this analysis, these characteristics are considered to establish a design that fits the objectives and limitations of this project.

4.1 FIDO2

FIDO2 is a set of specifications that together define an authentication protocol for online services and it aims to replace passwords as its main authenticator factor or to be used as a strong second-factor or multi-factor authentication (Alliance, FIDO 2017). These are composed of CTAP2 and WebAuthn specifications. FIDO2 was designed by the FIDO Alliance and submitted to the W3C on November 2015 (Alliance, FIDO 2019*b*), however, was until March 2019 that was published as a standard by the W3C World Wide Web Consortium (2019).

4.1.1 Protocol Overview

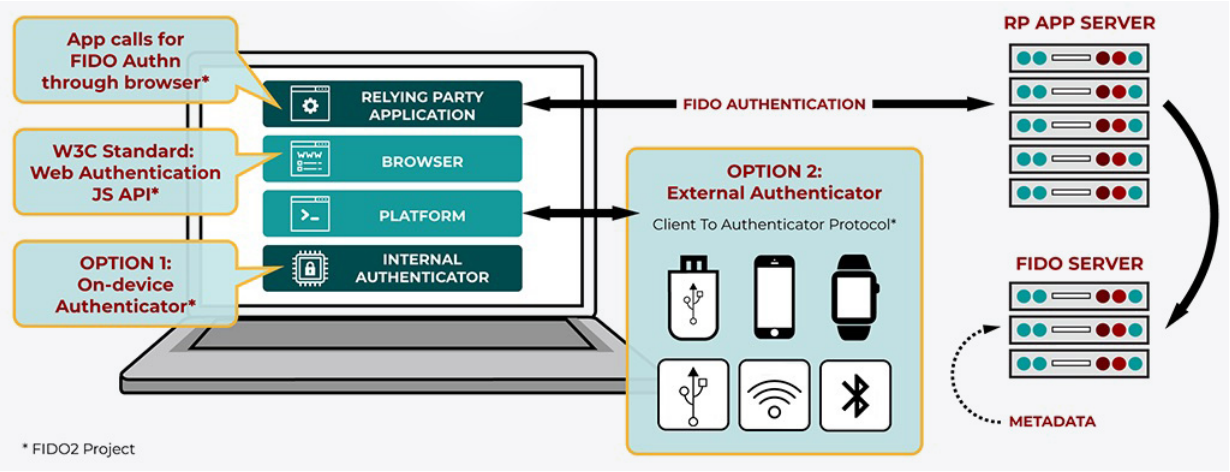

Figure 4.1.1: *FIDO2 graphical overview. Image extracted from (Alliance, FIDO 2019c)*

FIDO2 is composed of three main components (Relying Party, Client and Authenticator) (World Wide Web Consortium 2019). A relying party (RP), which is an online service where a user wants to authenticate. The RP must provide a FIDO server which will store users credentials and perform authentication on RP side. A client, which is a software on a platform used by the user to access the RP online service, for example, a web browser. The client is a trusted application which allows the RP to communicate with a users authenticator. The client must communicate with the RP through a secure channel such as TLS. And an authenticator, which is a piece of hardware and/or software owned by the user that performs the cryptographic operations to register or authenticate a credential on the RP. An authenticator could be embedded on the same device platform or be an external device that communicates to the users client through a transport protocol such as USB HID, BLE or NFC (Alliance, FIDO 2019*a*).

Registration Process

The registration process for a FIDO2 credential can occur in two scenarios:

- To be used as a single-factor authenticator and therefore as a complete replacement for a password authentication to an online service.
- Or to be used as a second or multi-factor authentication on the online service.

For both scenarios, the RP could ask some information to the user depending on the RP registration policies. Regardless of which previous scenarios, the registration process is performed as follows and depicted in Figure 4.1.2.

1. The RP sends to the client its *rpId* that it is its domain name, a *user handle* that must be a unique identifier on the RP for each user, and a fresh and random *challenge* for each request no matter it is a registration or authentication process.

2. The client validates the *rpId* against the RP origin domain. If it is the same, continues with the protocol, otherwise, it stops the protocol.

3. The client sends to the authenticator the *rpId*, a hash of the challenge and origin known as *clientData*, and the *user handle*.

4. The authenticator requests user presence or verification. If the authenticator is capable of verifying the users identity, it should do it with a PIN or other security mechanism such as fingerprint verification. In case user presence or users identity is verified, before a reasonable time, the authenticator should answer continue the protocol, otherwise, it should send an error response and stop the protocol.

5. The authenticator generates a public-key pair (k_{priv} and k_{pub}) that are associated with the *rpId* and *user handle* using a unique credential identifier on the authenticator named *id* in the figure.

6. The authenticator signs a concatenation of *rpId*s hash, *id*, k_{pub}, and *clientData* using the attestation public key A_{pub}.

7. The authenticator sends to the client the attestation object composed by the *rpId*s hash, the *id*, the public key k_{pub}, the signature generated in the previous step and the attestation public key A_{pub}.

8. The client sends to the RP the *id*, the *clientData* and the *attestation object* received from the authenticator.

9. The RP verifies the *origin* and *challenge* against the *clientData*.

10. The RP verifies that the attestation public key A_{pub} is trusted by the FIDO metadata service.

11. The RP verifies the signature using the attestation public key A_{pub}.

12. If all RP verifications are successful, the RP registers the public key k_{pub} generated by the authenticator, the credential identifier *id* and the *user handle*. Otherwise, the RP informs the client and the user that an error occurred.

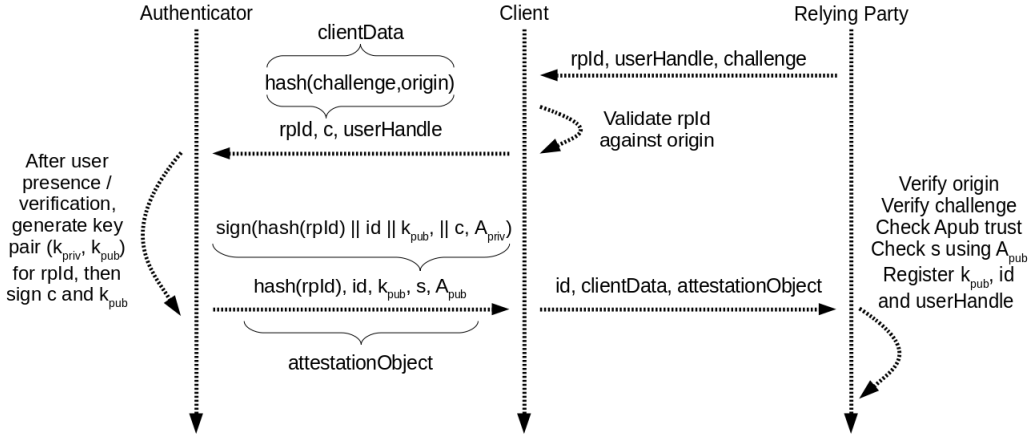

Figure 4.1.2: *FIDO2 Registration process (Make Credential)*

Authentication Process

The authentication process (also known as get assertion) for a FIDO2 credential can also be performed as a single-factor or multi-factor authentication. Regardless of this, the authentication process is performed as follows and depicted in Figure 4.1.3:

1. The RP sends to the client its *rpId*, the credential identifier as *id*, and a fresh and random *challenge*.

2. The client validates the *rpId* against the RP *origin* domain. If it is the same, continues with the protocol, otherwise, it stops the protocol.

3. The client sends to the authenticator the *rpId*, a hash of the *challenge* and *origin* as *clientData*, and the *id*.
4. The authenticator requests user presence or verification in the same way as in the registration process.
5. The authenticator retrieves the private-key k_{priv} associated with the *rpId* and *user handle* using the credential identifier *id*.
6. The authenticator signs a concatenation of *rpId*s hash and *clientData* using the private-key k_{priv} generated in the registration process and associated with the credential identifier *id*.
7. The authenticator sends to the client the *rpId*s hash, the *id*, and the *signature* generated in the previous step.
8. The client sends to the RP the *id*, *clientData*, *rpId*s hash and the *signature* created by the authenticator.
9. The RP verifies the *origin* and *challenge* against the *clientData*.
10. The RP verifies the credential identifier *id* is the same as the one originally sent at the beginning of the process.
11. The RP verifies the signature using the public key k_{pub} obtained in the registration process.
12. If all RP verifications are successful, the RP authenticates the user on the online service. Otherwise, the RP informs the client and the user that an error occurred.

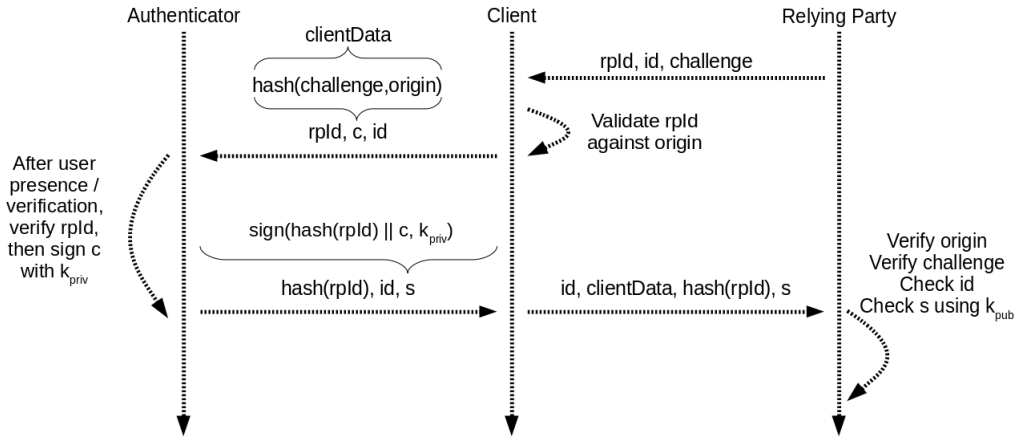

Figure 4.1.3: *FIDO2 Authentication process (Get Assertion)*

4.1.2 Design considerations

Design purpose

Design a CTAP2 protocol that fits the time constraints and hardware limitations.

It is important to mention that FIDO2 is a flexible protocol, which grants many different options when implementing the protocol according to usability, security and hardware requirements. Therefore, all design decisions made are within the specifications of the FIDO2 protocol.

Design scope

These constraints are considered to propose this design:

- The limitation of analysing, designing, implementing and documenting this project within a period of two months.
- The limitation of 1 MB of Flash storage.
- The lack of trusted storage in the ARM TrustZone Cryptocell 310.

The design and implementation only cover the CTAP2 protocol using the USB HID transport as a means of communication between the authenticator and the client, taking into account the aforementioned constraints.

Hardware selection

The nRF52840 SoC Nordic Semiconductor (2019*c*) was chosen to implement the CTAP2 protocol for this project for the following reasons:

- Has embedded an ARM TrustZone Cryptocell, which provides a TEE and a cryptographic hardware accelerator.
- Provides USB, BLE and NFC communication capabilities.
- Can be found in a small and low-cost USB dongle board.

Registration process design

The following describes the registration process designed, highlighting and explaining the modifications to the original protocol design.

1. The RP sends to the client its *rpId* that it is its domain name, a *user handle* that must be a unique identifier on the RP for each user, and a fresh and random *challenge* for each request no matter it is a registration or authentication process.
2. The client validates the *rpId* against the RP origin domain. If it is the same, continues with the protocol, otherwise, it stops the protocol.
3. The client sends to the authenticator the *rpId*, a hash of the challenge and origin known as *clientData*, and the *user handle*.
4. **MODIFICATION:** The authenticator requests user presence. **REASONS:** The nRF52840 USB dongle only has one button as user input and no display capabilities. Also, the *CTAP2 ClientPIN API* implementation is discarded due to time constraint.
5. **MODIFICATION:** The authenticator generates a public-key pair (k_{priv} and k_{pub}) using the Cryptocell, that are associated with the *rpId* and *user handle*. The authenticator constructs a credential identifier named *id* in the figure as follows:
   (a) Concatenate the private key k_{priv}, the *rpId* and *userHandle*.
   (b) Encrypt the concatenation result using AES-CTR algorithm and an AES_{key}.
   (c) The ciphertext result is the new credential identifier *id*.
   (d) **REASONS:** All credential material, including k_{priv}, *rpId*, *userHandle* are sent to the RP which stores that information encrypted by the authenticator as the credential *id*, and only the authenticator has the AES_{key} to decrypt the credential material, therefore, the information stored in the RP is secure as long as the AES_{key} remains secure. That information will be sent by the RP in each authentication request (in encrypted form), hence, the authenticator does not require to store any credential material, and only needs to store the AES_{key} to decrypt it. With this design modification, the authenticator supports an unlimited number of credentials.
6. **MODIFICATION:** The authenticator signs a concatenation of *rpId*s hash, *id*, k_{pub}, and *clientData* using private key k_{priv}. **REASONS:** FIDO2 specification allows an authenticator sign the attestation object using the respective private key k_{priv} bounded to the credential material created. The A_{pub} allows an RP to verify the trust of a manufactured authenticator against the FIDO2 metadata service, however, this project implementation is in a prototype state, therefore, it was omitted. Using an attestation key A_{pub} will be part of the future work for this project.

7. **MODIFICATION:** The authenticator sends to the client the *attestation object* composed by the *rpId*s hash, the *id*, the public key k_{pub}, and the *signature* generated in the previous step. Because there is no attestation key A_{pub}, it is omitted in the authenticator response. **REASON:** The same reason as in the previous step.

8. The client sends to the RP the *id*, the *clientData* and the *attestation object* received from the authenticator.

9. The RP verifies the *origin* and *challenge* against the *clientData*.

10. **MODIFICATION and REASON:** Due to the absence of an attestation key A_{pub} the RP does not verify an attestation public key A_{pub} against the FIDO metadata service. It is important to note that an RP may have in their authentication politics to only accept credentials that are signed by a trusted attestation key A_{pub} that can be verified by the FIDO metadata service. Therefore, this prototype version will not work in those cases.

11. **MODIFICATION:** The RP verifies the signature using the public key k_{pub}. **REASON:** Absence of an attestation key A_{pub}.

12. If all RP verifications are successful, the RP registers the public key k_{pub} generated by the authenticator, the credential identifier *id* and the *user handle*. Otherwise, the RP informs the client and the user that an error occurred.

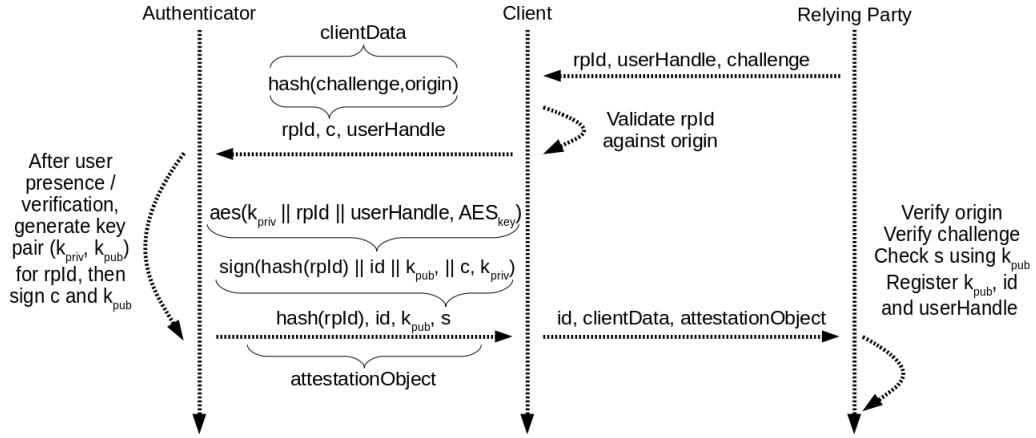

Figure 4.1.4: *Proposed design for FIDO2 Registration process (Make Credential)*

Authentication process design

The following describes the authentication process designed, highlighting and explaining the modifications to the original protocol design.

1. The RP sends to the client its *rpId*, the credential identifier as *id*, and a fresh and random *challenge*.

2. The client validates the *rpId* against the RP *origin* domain. If it is the same, continues with the protocol, otherwise, it stops the protocol.

3. The client sends to the authenticator the *rpId*, a hash of the *challenge* and *origin* as *clientData*, and the *id*.

4. **MODIFICATION:** The authenticator requests user presence. **REASONS:** The nRF52840 USB dongle only has one button as user input and no display capabilities. Also, the ClientPIN implementation is discarded due to time constraint.

5. **MODIFICATION:** The authenticator retrieves the private-key k_{priv} associated with the *rpId* and *user handle* using the credential identifier *id* as follows:

(a) The authenticator decrypts the credential identifier *id* using the AES-CTR decryption algorithm and the AES_{key} stored in the authenticator.

(b) Then retrieves the private-key k_{priv}, the *rpId* and the *userHandle*.

(c) The authenticator verifies that the *rpId* retrieved from the credential *id* is the same as the *rpId* received from the client.

(d) **REASONS:** All credential material, including k_{priv}, *rpId*, *userHandle* are sent encrypted to the RP in the registration process as the credential identifier *id*. Only the authenticator has the AES_{key} to decrypt the credential material, therefore, the information stored in the RP is secure as long as the AES_{key} remains secure. The authenticator does not require to store any credential material, and only needs to store the AES_{key}. With this design modification, the authenticator supports an unlimited number of credentials.

6. The authenticator signs a concatenation of *rpId*s hash and *clientData* using the private-key k_{priv} generated in the registration process and associated with the credential identifier *id*.

7. The authenticator sends to the client the *rpId*s hash, the *id*, and the *signature* generated in the previous step.

8. The client sends to the RP the *id*, *clientData*, *rpId*s hash and the *signature* created by the authenticator.

9. The RP verifies the *origin* and *challenge* against the *clientData*.

10. The RP verifies the credential identifier *id* is the same as the one originally sent at the beginning of the process.

11. The RP verifies the signature using the public key k_{pub} obtained in the registration process.

12. If all RP verifications are successful, the RP authenticates the user on the online service. Otherwise, the RP informs the client and the user that an error occurred.

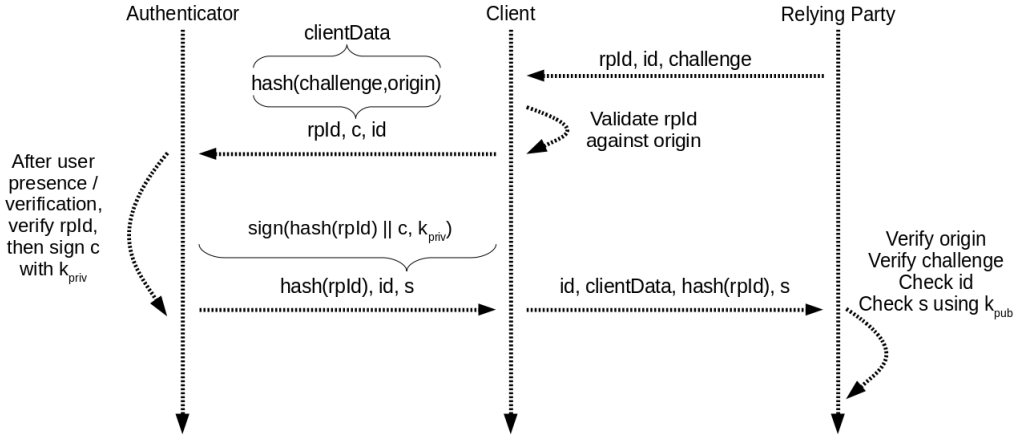

Figure 4.1.5: *Proposed design for FIDO2 Authentication process (Get Assertion)*

4.2 Threat model

This section describes possible attacks against FIDO2 authentication protocol and its countermeasures.

4.2.1 Threats and countermeasures

Stolen authenticator

An attacker who gains physical access to an authenticator only requires to guess the user handle and the relying party for which was used to impersonate a legitimate user.

Countermeasures

- Besides verify the user presence an authenticator could verify the users identity using a PIN or a fingerprint, depending on its capabilities.
- Depending on its risk tolerance, a relying party can whitelist or blacklist authenticators based on its capabilities, model, vendor, level of certification and attestation class.
- A legitimate user can revoke credentials on the relying party created with a missing authenticator.

Side-channel and fault injection attacks

An attacker who gets physical access to an authenticator can execute a side-channel attack to recover sensitive information from the authenticator or a fault injection attack to bypass some logical or physical protection.

Countermeasures

- Use of hardware security anchors, with tamper-resistant hardware and cryptographic hardware accelerator, resilient to side-channel and fault injection attacks, such as a Trusted Execution Environment or Secure Enclave.

Cryptographic algorithms attack

An attacker could leverage insecure cryptographic algorithms or flawed implementations.

Countermeasures

- Use of strong, modern and sound cryptographic algorithms.
- Use of verified cryptographic algorithms implementations.

Compromise of authentication private key

An attacker manages to get a private key for authentication and use it to impersonate a user.

Countermeasures

- An authentication private key is generated per authenticator, per relying party and per user handle. That means the impact of an authentication private key leak will have an impact only on that context and will not affect other credentials.
- Use of tamper-resistant hardware to produce and store key material.
- Use of strong, modern and sound cryptographic algorithms.
- Use of verified cryptographic algorithms implementations.

Compromise of attestation private key

An attacker manages to get an attestation private key and is able to impersonate a certified authenticator.

Countermeasures

- A relying party can verify if an attestation key is compromised using the FIDO Metadata Service from fidoalliance.org.
- Use of tamper-resistant hardware to produce and store key material.
- Use of strong, modern and sound cryptographic algorithms.
- Use of verified cryptographic algorithms implementations.

Malformed input injection

An attacker manages to inject malformed input to an authenticator to extract sensitive information such as key material, user handle or relying party id, or to bypass some logical or physical security protections.

Countermeasures

- CTAP2 specification defines input type format, length and error handling.
- FIDO certification verifies the design, implementation and manufacture of an authenticator. Moreover, a relying party can verify the level of certification of an authenticator against the FIDO Metadata Service.

Man in the middle attack

An attacker performs a MITM attack trying to read or tamper with data exchange between the authenticator and the client or the client and relying party to steal data or use it to impersonate a legitimate user on this or another relying party.

Countermeasures

- Use a secure channel such as TLS between a client and a relying party or USB HID, BLE, NFC between authenticator and client.
- If an attacker tampers with the rpId, userHandle or challenge, the relying party will notice when verifies the signature created by the authenticator.
- As long as the attacker hasnt the respective attestation private key or authentication private key, she cant forge the signature from the authenticator.

Replay attack

An attacker replays a previous authentication message to impersonate a legitimate user.

Countermeasures

- The relying party uses a fresh nonce (challenge) for every registration and authentication process.

Malicious Client

An attacker can take control of the client with malware. Therefore, can use it to trick the user to use the authenticator to authenticate to another relying party different that the user is giving consent (assuming there is an existent credential between the authenticator and the second relying party).

Countermeasures

- If the authenticator has display capability, it can show information about the relying party and user handle that will be used to authenticate before she consents the action.
- As the malicious client cant access private keys, as soon as it is removed from the device the security of credential is restored.

Compromise of FIDO Server DB

An attacker can gain access to a FIDO Server DB from a relying party and steal its stored credentials to use them to impersonate users to this or another relying party.

Countermeasures

- The FIDO Server DB only stores public key credentials and user handles.
- Using strong, modern and sound cryptographic algorithms makes infeasible to extract private keys from public keys.
- Due to a credential is per authenticator, user handle and relying party, it cant be used against other relying parties.
- The FIDO Server DB should implement security measures to prevent database compromise.

Moreover, an attacker can get writing access to a FIDO Server DB from a relying party. Therefore, inject new valid but illegitimate credentials.

Countermeasures

- The FIDO Server DB should implement security measures to prevent database compromise.

Malware on RP application

An attacker can execute malware on the relying party application and try to impersonate or execute actions on behalf of the user.

Countermeasures

- If the action such as a bank confirmation payment uses a FIDO2 transaction confirmation, and the authenticator has display capabilities, the user could spot that the relying party is trying to do a different action as requested and she could decide not to give consent of the action.

Phishing application

An attacker can create a phishing application to steal credentials or impersonate against another relying party.

Countermeasures

- Due to the credential private key doesnt leave the authenticator, the attacker cant steal the credential private key, however, she only could steal the user handle.

Recovery account attack

An attacker can try to gain access to an account through the account recovery process.

Countermeasures

- Register multiple FIDO2 authenticator devices on the relying party.
- If there is only one authenticator device registered, then, the relying party will use traditional recovery process such as identity proofing and its security will depend on the requirements of the relying party. FIDO alliance recommends the NIST 800-63A Digital Identity Guidelines for identity proofing.

5 Implementation

This section describes the implementation process. From development configuration to testing, including code structure and protocol-specific implementations for this project.

5.1 Development environment setup

5.1.1 Hardware environment requirements

To implement the FIDO2 protocol for this project, the following hardware was used:

PC Personal computer (laptop) with the following essential features for this project:

- CPU Intel(R) Core(TM) i5-4310U CPU @ 2.00GHz
- 2 GB RAM or more (8 GB RAM was used)
- 3 GB of HD space available (2+ GB used by code, tools and binaries)
- 1 USB 2.0 port.
- Network card with internet connectivity (Wireless NIC was used)

Nordic nRF52840 Dongle The nRF52840 Dongle manufactured by Nordic Semiconductor is a SoC ideal for IoT devices with low power consumption. It also has a USB dongle form factor that is very convenient for this project. Nordic Semiconductor provides an SDK in C language programming to accelerate the development process with a wide set of libraries essentials for this project (Nordic Semiconductor 2019*c*).

Its more relevant features for this project are:

- a 64 MHz Arm Cortex-M4.
- an ARM TrustZone Cryptocell 310 security subsystem.
- 1 MB Flash.
- 256 KB RAM.
- Full-speed 12 Mbps USB.
- Bluetooth 5 multiprocol radio
- NFC-A tag
- 1 programmable button
- 1 reset button
- 1 programmable RGB LEDs
- USB dongle form factor

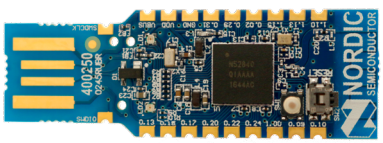

Figure 5.1.1: *Nordic nRF52840 USB Dongle (Nordic Semiconductor 2019c)*

This board was the most used for project development and testing. The final implementation is fully compatible with this board without modifications.

Nordic nRF52840 DK The nRF52840 DK manufactured by Nordic Semiconductor is a SoC ideal for software and prototyping IoT devices with low power consumption. It uses the same nRF52840 SoC but it has a different form factor, offering more features (particularly for software development) and I/O ready to use than the Dongle version. Nordic Semiconductor provides an SDK in C language programming to accelerate the development process with a wide set of libraries essentials for this project (Nordic Semiconductor 2019*c*).

Its more relevant features for this project are:

- a 64 MHz Arm Cortex-M4.
- an ARM TrustZone Cryptocell 310 security subsystem.
- 1 MB Flash.
- 256 KB RAM.
- Full-speed 12 Mbps USB.
- Bluetooth 5 multiprocol radio
- NFC-A tag
- 4 programmable buttons
- 1 reset button
- 4 programmable LEDs
- Segger J-Link OB programming/debugging
- USB dongle form factor

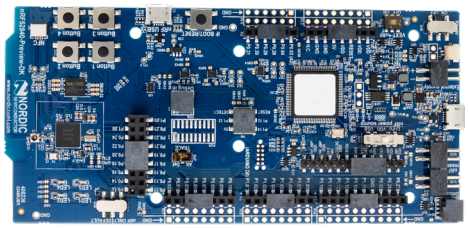

Figure 5.1.2: *Nordic nRF52840 DK (Nordic Semiconductor 2019*c*)*

This board was ideal for this project development, particularly for the debugging option that provides with the Segger IDE. However, the board was acquired in Week 10 of the project, just one week before the project demonstration. Therefore, the time was not enough to modify and test the project implementation into this board.

nRF52840 MDK USB Dongle The nRF52840 MDK USB Dongle is manufactured by Makerdiary using the nRF52840 SoC manufactured by Nordic Semiconductor (Makerdiary 2019*b*). It is also ideal for IoT devices with low power consumption and has a USB dongle form factor that is very convenient for this project.

Its more relevant features for this project are:

- a 64 MHz Arm Cortex-M4.
- an ARM TrustZone Cryptocell 310 security subsystem.
- 1 MB Flash.
- 256 KB RAM.
- Full-speed 12 Mbps USB.

- Bluetooth 5 multiprocol radio
- NFC-A tag
- 1 reset/programmable button
- 1 programmable RGB LED
- 1 power LED
- USB dongle form factor

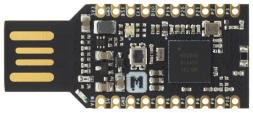

Figure 5.1.3: *Makerdiary nRF52840 MDK USB Dongle (Makerdiary 2019b)*

This was the first board acquired for the project development. Its features are very similar to the Nordic nRF52840 USB Dongle and cheaper.

Besides, the open-source project Makerdiary nRF52-U2F was developed for this card. The Makerdiary nRF52-U2F was used as the basis for the development of this FIDO2 project.

However, after start using the board, an issue with the reset/programmable button was found. The issue was that when the user button is pressed to approve the make credential or get assertion of FIDO2, the board turned to programming mode instead of performing the user consent.

5.1.2 Software environment requirements

Operating System The following operating systems are supported to set the software development environment (Nordic Semiconductor 2019*a*):

- Windows 7, Windows 8 or Windows 10
- macOS
- Linux

The Operating system used to set up the software development environment for this project was: Xubuntu 18.04 LTS Desktop.

5.1.3 Setup process

Depending on the specific environment to use, some of the following packages are optional and some are mandatory. This will be specified for the environment setup used for this project.

GNU GCC ARM To compile the FIDO2 code project it can be used the SEGGER Embedded Studio IDE on Linux, the ARM Keil MDK IDE for Windows or the GNU GCC ARM.

The GNU GCC ARM was the tool used to compile the code project. The reason to use this tool was mainly to the experience of using a GNU GCC compiler and the lack of experience of using the two IDEs mentioned above.

The GNU GCC ARM is a C programming language compiler for ARM architectures such as the nRF52840 SoC.

Xubuntu as a derivate of Ubuntu Linux provides a GNU GCC ARM compiler within its defaults repositories. However, the version provided in the repositories has problems to compile programs developed with the nRF52 SDK v15.3.0 which is the last version available of this SDK. Therefore, it is recommended to download the last version of GNU GCC ARM compatible for your operating system from the ARM Developers website: `https://developer.arm.com/tools-and-software/open-source-software/developer-tools/gnu-toolchain/gnu-rm/downloads`.

The 7.3.1 version was used to compile the code of this project.

Instalation process To install GNU GCC ARM version 7.3.1 on Xubuntu:

```
$ sudo add-apt-repository ppa:team-gcc-arm-embedded/ppa
$ sudo apt update
$ sudo apt-get update
$ sudo apt-get install gcc-arm-embedded
```

nRF Connect for Desktop The nRF Connect for Desktop is a tool to program firmware into nNRF52 family boards such as those with the nRF52840 SoC (Nordic Semiconductor 2019*b*).

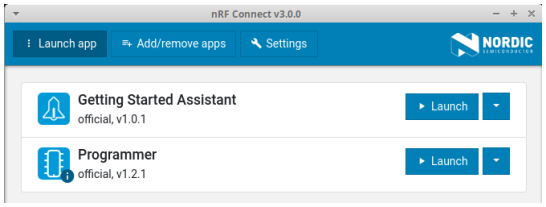

Figure 5.1.4: *nRF Connect for Desktop (Nordic Semiconductor 2019*b*)*

Instalation process The nRF Connect for Desktop for Linux comes in an AppImage format. This format is a portable distribution format for Linux which is a precompiled and ready to tun binary (AppImage 2019).

1. Download the las nRF Connect for Desktop version from `https://www.nordicsemi.com/Software-and-Tools/Development-Tools/nRF-Connect-for-desktop/Download`.
2. Give execution permissions to the binary.

   ```
   $ chmod u+x nrfconnect-3.0.0-x86_64.AppImage
   ```

3. Then you can run the nRF Connect for Desktop with root permissions due it needs to access to USB hardware:

   ```
   $ sudo ./nrfconnect-3.0.0-x86_64.AppImage
   ```

nRF52 SDK The nRF52 SDK maintained by Nordic Semiconductor is a complete SDK in C language programming to accelerate the development process with a wide set of libraries essentials for this project (Nordic Semiconductor 2019*c*).

The more important libraries used for the FIDO2 implementation in this project are:

- BLE libraries and services (for future FIDO2 Bluetooth implementation).
- Bootloader and DFU modules (for future implementation in the project).
- Board Support Package (Used to control board's I/O such as Buttons or LEDs)
- Cryptography Library (Used to perform cryptographic operations. Particularly, the nrf_cc310 library backend was used to perform cryptographic operations using the ARM TrustZone Cryptocell embedded in the nRF52840).
- Memory Manager (To allocate memory dynamically).
- NFC (for future FIDO2 NFC implementation).
- Flash Data Storage (To store persistently the AES key, and global signature counter).
- USB HID (To receive and response CTAP2 HID commands)

The nRF52 SDK version used for this project is 15.3.0. The Makerdiary nRF52 FIDO U2F Security Key project used the 15.2.0 version (Makerdiary 2019*a*). Therefore, as the FIDO2 project implementation was built upon the Makerdiary nRF52 FIDO U2F Security Key project, small changes on library names or library paths had to be made in the Makefile to compile correctly the FIDO2 project.

Instalation process

1. Download the last version of nRF52 SDK from `https://www.nordicsemi.com/Software-and-Tools/Software/nRF5-SDK`.

2. Uncompress the zip file into the project folder.

Python-FIDO2 The Python FIDO2 project developed by Yubico is a library which provides FIDO2 functionality to act as a FIDO2 client or FIDO2 server to interact and test FIDO2 authenticators. It supports FIDO2 and U2F protocols.

This library was used to test the implementation of the nRF52840 FIDO2 project during its development.

The library is installed in the Python environment as a normal Python package and the examples to perform testing can be download from the GitHub project web site in `https://github.com/Yubico/python-fido2/tree/master/examples`.

Instalation process

1. Install the Python FIDO2 package on Linux:

```
$ pip install fido2
```

2. Download the examples from `https://github.com/Yubico/python-fido2/tree/master/examples`.

3. Run a simple .

```
$ python get_info.py
```

UDEV rules UDEV is the device manager in Linux which detects when a new device is plugged or unplugged. When a new device is plugged, it locates a UDEV rule that applies that devices and execute the instructions on that rule to make the device work as expected.

When a new device is manufactured, if there is no UDEV rule that applies to it, then it needs to add it. Those rules are added with system updates. However, a rule can be added manually to use the device.

Linux by default does not provide UDEV rules for U2F and FIDO devices. Therefore, a UDEV rule has to be added in Linux to detect and configure correctly the FIDO2 authenticator.

Yubico provides a file with a variety of U2F and FIDO2 rules. The recommendation is to use this list and add a new rule to it for the nRF52840 FIDO2 authenticator implemented in this project.

The Yubico UDEV rules can be downloaded from `https://github.com/Yubico/libu2f-host/blob/master/70-u2f.rules`.

In most Linux distributions, the UDEV rules are in the directory /etc/udev/rules.d/. Therefore, the file 70-u2f.rules should be stored in this directory.

Then, at the end of the file 70-u2f.rules, the following line must be added in order to Linux detect and configure correctly the nRF52840 FIDO2 authenticator implemented in this project.

```
KERNEL=="hidraw*", SUBSYSTEM=="hidraw", ATTRS{idVendor}=="1915",
ATTRS{idProduct}=="520f", TAG+="uaccess", GROUP="plugdev", MODE="0660"
```

This line indicates that the device uses the generic USB HID protocol. Its vendor ID is 1915 that is part of a configuration of the USB HID library in the code project. And its product ID is 520f, that also is part of a configuration of the USB HID library in the code project. These configurations can be found and modified if needed in the *sdk_config.h* file in the source code of this project. A USB HID device manufacturer should request to the USB Implementers Forum to be assigned by a vendor id, and then for a product id. However, as the project is in a prototype state, this is not required.

The plugdev group and permission mode 0660 are applied to a normal user without root permissions can use the device.

After adding the rule, the UDEV daemon service should be restarted as follows.

```
$ sudo udevadm control --reload-rules
```

Now, the device should be detected and this can be verified as follows.

```
$ dmesg
...
[77981.321587] usb 2-2: new full-speed USB device number 8 using xhci_hcd
[77981.471215] usb 2-2: New USB device found, idVendor=1915, idProduct=520f
[77981.471217] usb 2-2: New USB device strings: Mfr=1, Product=2, SerialNumber=3
[77981.471218] usb 2-2: Product: nRF52 CTAP Security Key
[77981.471219] usb 2-2: Manufacturer: Nordic Semiconductor
[77981.471219] usb 2-2: SerialNumber: 000000000000
[77981.472558] hid-generic 0003:1915:520F.0003: hiddev0,hidraw0: USB HID v1.11
Device [Nordic Semiconductor nRF52 CTAP Security Key] on usb-0000:00:14.0-2/input0
```

Gedit text editor The Gedit is a simple text editor which supports code highlighting. This was the code editor choice due to familiarity with this editor and its lightness.

Instalation process Gedit is normally installed in most Linux distributions, although it can be installed using as follows.

```
$ sudo apt gedit
```

SEGGER J-Link Installation: Optional.

SEGGER J-Link is a tool which allows debugging the execution of programs in J-Link enabled boards such as the Nordic nRF52840 DK.

Its installation is not mandatory to the project development unless a compatible J-Link board is used.

The SEGGER J-Link was installed for this project to try to debug the implementation using the Nordic nRF52840 DK board. However, the J-Link was not tested because the Nordic nRF52840 DK was obtained in the last week of the development process and for time constraint its testing was dismissed.

Instalation process

1. Download the J-Link installer from its official web page `https://www.segger.com/downloads/jlink/`.

2. On Linux, install it with the following command:

```
$ sudo dpkg -i JLink_Linux_V646h_x86_64.deb
```

5.2 Code structure

As shown in the Appendix B, the main source code of the project is the CTAP2 folder inside the nRF52840_NORDIC_USB_Dongle.

The CTAP2 code structure is as follows:

```
/
├── cbor -- TinyCBOR Intel implementation used in this project
├── certs -- Keys and Certificates used for Makerdiary U2F implementation
├── pca10059 -- Configurations specific for the Nordic nRF52840 Dongle board
│   └── mbr
│       ├── armgcc -- Compiling and linking configurations
│       │   ├── _build -- Build directory where hex files are compiled
│       │   ├── ctap_gcc_nrf52.ld -- Linking configurations
│       │   └── Makefile -- Compiling makefile configurations
│       └── config -- SDK configurations
├── tools -- Tools to generate Keys and Certificates for Makerdiary U2F
├── cose_key.h -- COSE Key values
├── ctap.h -- CTAP Data structures and definitions
├── ctap_errors.h -- CTAP error definitions
├── ctap_hid.c -- CTAP USB HID protocol implementation
├── ctap_hid.h -- CTAP USB HID data structures and definitions
├── ctap_hid_if.c -- USB HID protocol implementation
├── ctap_hid_if.h -- USB HID data structures and definitions
├── ctap_impl.c -- CTAP2 protocol implementation
├── main.c -- Main function that initialize the execution
├── timer.c -- Timer functions used by Makerdiary U2F
├── timer_interface.h -- Timer interface used by Makerdiary U2F
├── timer_platform.h -- Timer structures used by Makerdiary U2F
├── util.c -- Utilities to blink led for debugging pourpuses
└── util.h -- Data definitios for debugging utilities
```

Within the code's comments, it is established which functions where obtained completely from another source, which are based from other source. Otherwise, the code and functions where developed from scratch.

5.3 Testing process

Two methods were used to test the FIDO2 implementation.

Python FIDO2 was used to test CTAP2 requests and responses, using the examples provided in its GitHub repository.

The ctap2.py from Python FIDO2 library was modified to print and debug the requests and responses of CTAP commands as follows.

```
599 print('send_bor')
600 request = struct.pack('>B', cmd)
601 if data is not None:
602 request += cbor.encode(data)
603 print('request size: %s' %sys.getsizeof(request))
604 print('request: %s' %request)
605 with Timeout(timeout) as event:
606 response = self.device.call(CTAPHID.CBOR, request, event,
607 on_keepalive)
608 print('response: %s' %response)
```

The example get_info.py was used to test the getInfo implementation with the authenticator as follows.

```
$ python3 get_info.py
CONNECT: CtapHidDevice(/dev/hidraw0)
CTAPHID protocol version: 2
send_bor
```

```
request size: 34
request: b'\x04'
response: b'\x00\xa4\x01\x81hFIDO_2_0\x03P\xe6w\xcej\x86>^\xff(\x0eu\xcc\xbf\x1f\x
↪  99s\x04\xa3brk\xf5bup\xf5dplat\xf4\x05\x19\x04\xb0'
DEVICE INFO: Info(versions: ['FIDO_2_0'], aaguid:
↪  h'e677ce6a863e5eff280e75ccbf1f9973', options: {'rk': True, 'up': True, 'plat':
↪  False}, max_message_size: 1200)
WINK sent!
```

The example credential.py was used to test the makeCredential and getAssertion implementations as follows. Due to the output of debugging requests and responses of those commands is very long, most of the output is omitted in the following example.

```
$ python3 credential.py
############ OUTPUT OMITTED INTENTIONALLY ##############
Touch your authenticator device now...
############ OUTPUT OMITTED INTENTIONALLY ##############
New credential created!
############ OUTPUT OMITTED INTENTIONALLY ##############
Attestation signature verified!
Touch your authenticator device now...
############ OUTPUT OMITTED INTENTIONALLY ##############
Assertion signature verified!
```

LED Feedback Due to the absence of a debugging option in the Nordic nRF52840 USB Dongle, an archaic LED debugging process was used to trace the code execution and errors.

The debugging functions blink LEDs in different colours, combining the RGB colours to get up to seven different colours plus the power led.

Besides, three different blinking speeds were used to have more feedback options.

Also, function to blink a specific colour a specific number of times was used to have another option of feedback.

This debugging functions using LEDs can be found in the utils.c and utils.h files in the source code.

5.4 CTAP2 implementation

The CTAP2 is the FIDO2 specification which establishes how an external authenticator should operate and communicate with a FIDO2 client using the WebAuthn standard (Alliance, FIDO 2017).

CTAP2 is divided into three parts (Alliance, FIDO 2019*a*):

- **Authenticator APIs:** these APIs is a conceptual representation of the operations that a FIDO2 authenticator should perform, and the inputs and outputs for each of those operations, including error handling specification.
- **Message Encoding:** here the specification defines how the inputs and outputs messages for requests and responses should be constructed and encoded.
- **Transport-specific Binding:** It defines how requests and responses should be processed according to the transport protocol used by the authenticator.

In the following sections, the specification used for the implementation is described and related to the source code implemented.

5.4.1 Message encoding

CTAP2 supports USB HID, Bluetooth Smart, BLE and NFC (Alliance, FIDO 2019*a*). However, those protocols are bandwidth restricted. Therefore, CTAP2 uses CBOR encoding (Alliance, FIDO 2019*a*) based on the JSON data model (Bormann & Hoffman 2013). CBOR offers a small code size, small message size and extensibility (Bormann & Hoffman 2013). This reduces the complexity of resources to parse and construct messages (Alliance, FIDO 2019*a*).

The requests sent to an authenticator using CBOR encoding are named commands (Alliance, FIDO 2019*a*).

The Intel TinyCBOR library which is embedded into the source code, allows to parse and encode data into the CBOR specification (Intel 2019).

Commands All CTAP2 requests start with a command value of 1-byte length. Some commands have parameters encoded in CBOR (Alliance, FIDO 2019*a*).

CTAP2 CBOR commands are processed with the function `static void ctap_hid_cbor_response(ctap_channel_t *p_ch);` in the ctap_hid.c file source.

The commands and its parameters with their respective CBOR key value can be observed in the next table.

Command (Key)	Parameter (Key)
authenticatorMakeCredential (0x01)	clientDataHash (0x01) rp (0x02) user (0x03) pubKeyCredParams (0x04) excludeList (0x05) extensions (0x06) options (0x07) pinAuth (0x08) pinProtocol (0x09)
authenticatorGetAssertion (0x02)	rpId (0x01) clientDataHash (0x02) allowList (0x03) extensions (0x04) options (0x05) pinAuth (0x06) pinProtocol (0x07)
authenticatorGetInfo (0x04)	

Table 1: CTAP2 command and its parameters (Alliance, FIDO 2019*a*)

The authenticatorMakeCredential request parameters are processed with the function `uint8_t ctap_parse_make_credential(CTAP_makeCredential * MC, CborEncoder * encoder, ctap_channel_t *p_ch);` in the ctap_impl.c file source.

The authenticatorGetAssertion request parameters are processed with the function `uint8_t ctap_parse_get_assertion(CTAP_getAssertion * GA, uint8_t * request, int length);` in the ctap_impl.c file source.

Responses All CTAP2 responses start with a status value of 1-byte length. The status 0x00 means success, otherwise means a specific error defined by CTAP2 standard (Alliance, FIDO 2019*a*).

The responses and its data with their respective CBOR key value can be observed in the next table.

Response	Parameter (Key)	Description
Make Credential	fmt (0x01)	The attestation statement format.
	authData (0x02)	The authenticator data.
	attStmt (0x03)	The attestation statement.
Get Assertion	credential (0x01)	The public key credential descriptor used create the assertion signature.
	authData (0x02)	The authenticator data without an attested credential data.
	signature (0x03)	The assertion signature.
	user (0x04)	The user entity bounded to the credential used for the assertion.
	numberOfCredentials (0x05)	The total number of valid credentials bounded to the authenticator and the rpId.
Get Info	versions (0x01)	List of supported protocols by the authenticator.
	extensions (0x02)	List of supported extensions by the authenticator.
	aaguid (0x03)	The AAGUID for the authenticator.
	options (0x04)	List of supported options.
	maxMsgSize (0x05)	Maximum message size supported by the authenticator.
	pinProtocols (0x06)	List of supported PIN protocols.

Table 2: CTAP2 Responses and its parameters (Alliance, FIDO 2019*a*)

The authenticatorMakeCredential response parameters are constructed inside the function `uint8_t ctap_make_credential(ctap_channel_t *p_ch);` in the ctap_impl.c file source.

The authenticatorGetAssertion response parameters are constructed inside the function `uint8_t ctap_get_assertion(ctap_channel_t *p_ch);` in the ctap_impl.c file source.

The authenticatorGetInfo response parameters are constructed inside the function `uint8_t ctap_get_assertion(uint8_t ctap_get_info(ctap_channel_t *p_ch);` in the ctap_impl.c file source.

Status codes As mentioned above, status codes represent the status of a response. A 0x00 status means a successful response, otherwise, it means a particular error during the operation of the API. For a detailed list of status codes, see the Status Code section from CTAP2 specification (Alliance, FIDO 2019*a*).

The source code file *ctap_errors.h* contains the definition of all CTAP1 and CTAP2 status codes. This source file was obtained from Solo implementation (SoloKeys 2019).

5.4.2 User presence

The FIDO2 requires user consent through user presence or user verification before create or access any credential. In this project user presence is verified as follows:

1. An event to handle button actions is added at board initialization in main.c.

```
143         static void init_bsp(void)
144         {
145                 ret_code_t ret;
146                 ret = bsp_init(BSP_INIT_BUTTONS, bsp_event_callback);
```

2. If the button is pressed, the event is captured in main.c.

```
120         static void bsp_event_callback(bsp_event_t ev)
121         {
```

...

```
133                 m_user_button_pressed = true;
```

3. At make credential request, the user presence is verified in ctap_impl.c.

```
1841        uint8_t ctap_make_credential(ctap_channel_t *p_ch)
1842        {
```

...

```
1897        while( !is_user_button_pressed() );     // USER PRESENCE REQUIRED HERE
```

4. At get assertion request, the user presence is verified in ctap_impl.c.

```
1956        uint8_t ctap_get_assertion(ctap_channel_t *p_ch)
1957        {
```

...

```
1995        while( !is_user_button_pressed() );     // USER PRESENCE REQUIRED HERE
```

5.4.3 Authenticator API

The CTAP2 API requests can be executed independently without any order in particular. Each request must provide a successful or failure response to the client. It is important to note that this API is conceptual and the implementation could vary for each authenticators manufacturer (Alliance, FIDO 2019*a*).

The CTAP2 API is composed of the following requests (Alliance, FIDO 2019*a*):

- authenticatorMakeCredential: used for the registration process.
- authenticatorGetAssertion: used to authenticate a credential previous registered.
- authenticatorGetNextAssertion: used when there is more than one credential bounded to the RP and the authenticator.
- authenticatorGetInfo: used by the RP to know the capabilities of the authenticator.
- authenticatorClientPIN: used by the client to set, update and verify a PIN stored in the authenticator.

The authenticatorMakeCredential, authenticatorGetAssertion and authenticatorGetInfo methods are implemented and design in detail for this project. Other methods were discarded due to time constraints and proposed as part of future work for this project.

5.4.4 getInfo API

The getInfo API is requested by a client to know the capabilities of an authenticator. A getInfo request is encoded as 0x04 command and it does not receive parameters.

In ctap_impl.c source file:

```
2076 uint8_t ctap_get_info(ctap_channel_t *p_ch)
```

On a successful request, the authenticator response must provide the following information (Alliance, FIDO 2019*a*):

Member name	Data type	Required?	Description
versions	Sequence of strings	Required	List of supported protocols by the authenticator, such as FIDO_2_0 for CTAP2 and U2F_V2 for CTAP1/U2F.
extensions	Sequence of strings	Optional **and omitted in the implementation**	List of supported extensions by the authenticator.

aaguid	Byte string	Required	The AAGUID for the authenticator. It is a 128-bit identifier that indicates the manufacturer and model of the authenticator.
options	Map	Optional	List of supported options.
maxMsgSize	Unsigned Integer	Optional	Maximum message size supported by the authenticator.
pinProtocols	Array of unsigned integers	Optional **and omitted in the implementation**	List of supported PIN protocols.

Table 3: Get Info response (Alliance, FIDO 2019*a*)

The option lists are the following key-value pair and when an option is not present the default value is used.

Option ID	Description	Default
plat	Platform device: informs that the authenticator is embedded on the same platform as the client.	false
rk	Resident key: informs that the authenticator can store credential by itself.	false
clientPin	Client PIN: informs if the authenticator is capable of accepting a PIN from the client and if it has been set or not.	Not supported
up	User presence: informs if the authenticator is capable to verify user presence.	true
uv	User verification: informs if the authenticator is capable of verifying user identity by itself, and if this capability is already configured or not into the authenticator.	Not supported

Table 4: List of options for Get Info response (Alliance, FIDO 2019*a*)

Implementation note: *The extensions and pinProtocols are omitted from the design and implementation of authenticatorGetInfo for this project due to time constraints..* Those are proposed for future work.

5.4.5 makeCredential API

The makeCredential API is requested by a client to generate a new credential in the authenticator. The newly created credential is used in the registration process of the users credential in the RP online service (Alliance, FIDO 2019*a*, World Wide Web Consortium 2019).

In ctap_impl.c source file:

```
1840 uint8_t ctap_make_credential(ctap_channel_t *p_ch)
```

MakeCredential receives the following parameters:

Parameter name	Data type	Required?	Definition
clientDataHash	Byte Array	Required	A hash of the client data composed by a type, the challenge, the origin, and a token binding (World Wide Web Consortium 2019). The type indicates if the operation will make a new credential or if it will get an assertion of an existing credential. The challenge is provided by the RP. The origin contains the FQDN of the RP. The token binding contains information of the token binding protocol (Popov n.d.) used between the client and the RP.
rp	Public-Key Credential RP Entity	Required	A data structure that describes the RP with which the new credential will be associated. Contains the RP identifier, a human-friendly RP name and an RP icon URL. The RP name and RP icon URL are optional and are used only if the authenticator has display capabilities.
user	Public-Key Credential User Entity	Required	A data structure which represents the user account to which the new public key credential will be associated at the RP. It contains an account identifier, a user name, a user display name, a user icon URL. The user name, user display name and user icon URL are optional and are used only if the authenticator has display capabilities.
pubKeyCredParams	CBOR Array	Required	A sequence of public-key cryptographic algorithms supported. The sequence is encoded in a CBOR Map and must be COSE algorithms registered by the IANA (Schaad 2017).
excludeList	Sequence of Public-Key Credential Descriptors	Optional **and ignored in this implementation**	A list of public-key credential descriptors, each of them is a structure that contains a previous credential created on the RP for the users account.
extensions	CBOR map	Optional **and ignored in this implementation**	A CBOR map of extensions which modify the operation of the authenticator, such as FIDO AppID or Simple Transaction Authorization (World Wide Web Consortium 2019).

options	Map of authenticator options	Optional	List of options which modify the operation of the authenticator. The options may be rk (resident key) and uv (user verification). Resident key requests to the authenticator store the credentials by itself. User verification requests to the authenticator verify the user with a gesture such as PIN or fingerprint. Both options are set to false by default.
pinAuth	Byte Array	Optional **and ignored in this implementation**	First 16 bytes of an HMAC-SHA-256 of clientDataHash using a pinToken obtained from the authenticator using the CTAP2 clientPIN API from a previous PIN created on the authenticator also using the same clientPIN API.
pinProtocol	Unsigned Integer	Optional **and ignored in this implementation**	PIN protocol chosen by the client.

Table 5: Make Credential input parameters (Alliance, FIDO 2019*a*, World Wide Web Consortium 2019)

When an authenticator receives a make credential request, it performs the following procedure (Alliance, FIDO 2019*a*, World Wide Web Consortium 2019):

1. If the excludeList is present, and it contains a credential created on the authenticator for the rpId, wait for user presence or verification and sends the error CTAP2_ERR_CREDENTIAL_EXCLUDED. Otherwise, continue operation (Alliance, FIDO 2019*a*).

    - ***Implementation note:*** *the exludeList is ignored in the project implementation and is proposed for future work. The implication is that the prototype implementation does not detect if a credential was created by the authenticator and bounded to the rpId and userHandle.*

2. If the pubKeyCredParams does not contain a supported COSE algorithm by the authenticator, then return the error CTAP2_ERR_UNSUPPORTED_ALGORITHM and stop the process, otherwise, continue with the operation (Alliance, FIDO 2019*a*).

    - ***Implementation note:*** *the COSE_ALG_ES256 is the only algorithm supported in the project implementation.*

```
941 static int pub_key_cred_param_supported(uint8_t cred, int32_t alg)
942 {
943         if (cred == PUB_KEY_CRED_PUB_KEY)
944         {
945                 if (alg == COSE_ALG_ES256)
946                 {
947                         return  CREDENTIAL_IS_SUPPORTED;
948                 }
949         }
950
951         return  CREDENTIAL_NOT_SUPPORTED;
952 }
```

3. Process all options received. If there is an unsupported option, then return error CTAP2_ERR_UNSUPPO and stop the process. If an option is not valid, return error CTAP2_ERR_INVALID_OPTION and stop the process. Otherwise, continue with the operation (Alliance, FIDO 2019*a*).

   - ***Implementation note:*** *the options rk and uv are not supported in the project implementation.*

4. Process all extensions if the parameter is present. If an extension is present and not supported return the error CTAP2_ERR_EXTENSION_NOT_SUPPORTED.

   - ***Implementation note:*** *no extension is supported in the project implementation. Therefore, extensions parameter is ignored and is part of future work for this project.*

5. Obtain user consent with a gesture (Alliance, FIDO 2019*a*). This could be done testing user presence or performing user verification through a PIN or fingerprint. If user consent is received, continue with the operation. Otherwise, return error CTAP2_ERR_OPERATION_DENIED. The options for obtaining user consent are:

   - Detect user presence using the authenticator capabilities, such as pressing a button.
   - Verify user identity with the authenticator capabilities, such as a PIN, or a fingerprint.
     - If pinAuth is present and is not supported return error CTAP2_ERR_PIN_AUTH_INVALID, but if it is supported, verify it before continuing. If the verification fails, return error CTAP2_ERR_PIN_AUTH_INVALID. If it is not present and is set in the authenticator, return error CTAP2_ERR_PIN_REQUIRED.
   - If the authenticator has display capabilities, display user and RP information and requests user gesture. If the user rejects action, return error CTAP2_ERR_OPERATION_DENIED.
   - ***Implementation note:*** *this project only implements user presence with pressing a button. User verification through a PIN is proposed for future work.*

```
1895 bsp_board_leds_off();
1896 bsp_board_led_on(GREEN_LED);
1897 while( !is_user_button_pressed() );     // USER PRESENCE REQUIRED HERE
1898 bsp_board_leds_off();
```

6. Generate a new credential object (World Wide Web Consortium 2019).

   (a) Create a public-key pair (private and public keys).

```
1610 err_code = nrf_crypto_ecc_key_pair_generate(
1611         NULL,                                          // Keygen context (NULL)
1612         &g_nrf_crypto_ecc_secp256r1_curve_info,        // Info structure
1613         private_key,                                   // Output private key
1614         &public_key);                                  // Output public key
```

   (b) Create a public-key credential source structure that contains:

   - type: the string public-key.
   - private-key: the private-key in binary format.
   - rpId: the rpId.
   - userHanle: the user entity id also known as user handle.

```
1638 memmove(credentialSource->rpId, rp->id, DOMAIN_NAME_MAX_SIZE + 1);
1639 memmove(credentialSource->userHandle, credInfo->user.id, USER_ID_MAX_SIZE);
```

   (c) If rk option is present in the request:

   - Create a new and unique credential id (shown as id in figure 4.1.2) as a 128-bit randomly.

   (d) Otherwise:

- Construct the credential id serializing and encrypting the public-key credential source so that only the authenticator be able to decrypt it.

```
err_code = nrf_crypto_aes_crypt(
        &aes_ctr_128_encr_ctx,                  // Context
        &g_nrf_crypto_aes_ctr_128_info,         // AES mode
        NRF_CRYPTO_ENCRYPT,                     // Ecrypt
        ↪ operation
        aes_key,                                // AES key
        iv,                                     // Initialization
        ↪ Vector (IV) Nonce and Counter
        credentialSourceWithOutIV,              // Data to encrypt
        CTAP_CREDENTIAL_SOURCE_SIZE,            // Data size
        credential_source_cipher,               // Ciphertext
        &data_out_size);                        // Ciphertext size
```

- ***Implementation note:*** *this project uses this second method to store and retrieve the private key, the rpId and the userHandle from the credential id.*

7. Create the signature counter (World Wide Web Consortium 2019).

- If the authenticator uses a global signature counter, use its actual value.

```
authData->head.signCount = m_auth_counter;
```

- If the authenticator supports a per credential signature counter, create the new counter initialized with cero and associate it with the new credential.
- If the authenticator does not support a signature counter, set the signature counter with cero constant.
- ***Implementation note:*** *due to no credential material is stored in the authenticator, a single global signature counter is used and stored in the authenticator for this project.*

8. Construct the attested credential data containing (World Wide Web Consortium 2019):

- The AAUID of the authenticator.

```
memmove(authData->attest.aaguid, CTAP_AAGUID, 16);
```

- The credential length in a 2-bytes big-endian.

```
authData->attest.credLenL =  sizeof(CTAP_credentialSource) & 0x00FF;
authData->attest.credLenH = (sizeof(CTAP_credentialSource) & 0xFF00) >> 8;
```

- The credential id.

```
memmove(&authData->attest.credentialId, credentialId,
↪ sizeof(CTAP_credentialSource));
```

- The public-key generated in previous steps using the COSE_key format.

```
err_code = ctap_add_cose_key(&cose_key, x, y,
↪ credInfo->publicKeyCredentialType, credInfo->COSEAlgorithmIdentifier);
```

9. Construct the authenticator data containing (World Wide Web Consortium 2019):

- rpIdHash: a SHA-256 hash of the rpId.

```
err_code = nrf_crypto_hash_calculate(
        &hash_context,                      // Context
        &g_nrf_crypto_hash_sha256_info,     // Info structure
        rp->id,                             // Input buffer
        rp->size,                           // Input size
        authData->head.rpIdHash,            // Result buffer
        &digest_size);                      // Result size
```

- flags: a byte indicating: if there was user presence and user verification, if the attested credential data and extensions data are included in the response.

```
1581 authData->head.flags = (1 << 0);     // User presence flag
1582
1583 if (credInfo != NULL)
1584 {
1585         authData->head.flags |= (1 << 6);   //include attestation data
```

- sigCount: the signature counter.

```
1577 authData->head.signCount = m_auth_counter;
```

- attested credential data.
- extensions data if any was processed in previous steps.

10. Construct the attestation statement format (World Wide Web Consortium 2019). There are a variety of attestation statement formats supported by FIDO2 such as:

- Packed.

```
1891 ret = cbor_encode_text_stringz(&map, "packed");
```

- TPM.
- Android Key.
- Android SafetyNet.
- FIDO U2F.
- None.

11. Construct the attestation statement (World Wide Web Consortium 2019).

(a) Sign the authenticator data with the private key generated in previous steps.

```
1915 ret = ctap_calculate_signature(auth_data_buf, auth_data_sz,
     ↪  MC.clientDataHash, signature, &signature_size, &private_key);
```

(b) Then construct the attestation statement according to the attestation statement format used.

```
1934 ret = ctap_add_attest_statement(&map, signature_der, signature_der_size);
```

12. Return to the client the attestation object which contains (Alliance, FIDO 2019*a*):

- The authenticator data.
- The attestation statement format.
- The attestation statement.

```
1942 ret = ctap_hid_if_send(p_ch->cid, p_ch->cmd, ctap_resp.data,
     ↪  ctap_resp.length);
```

In the figure below you can see graphically the response structure of a make credential request.

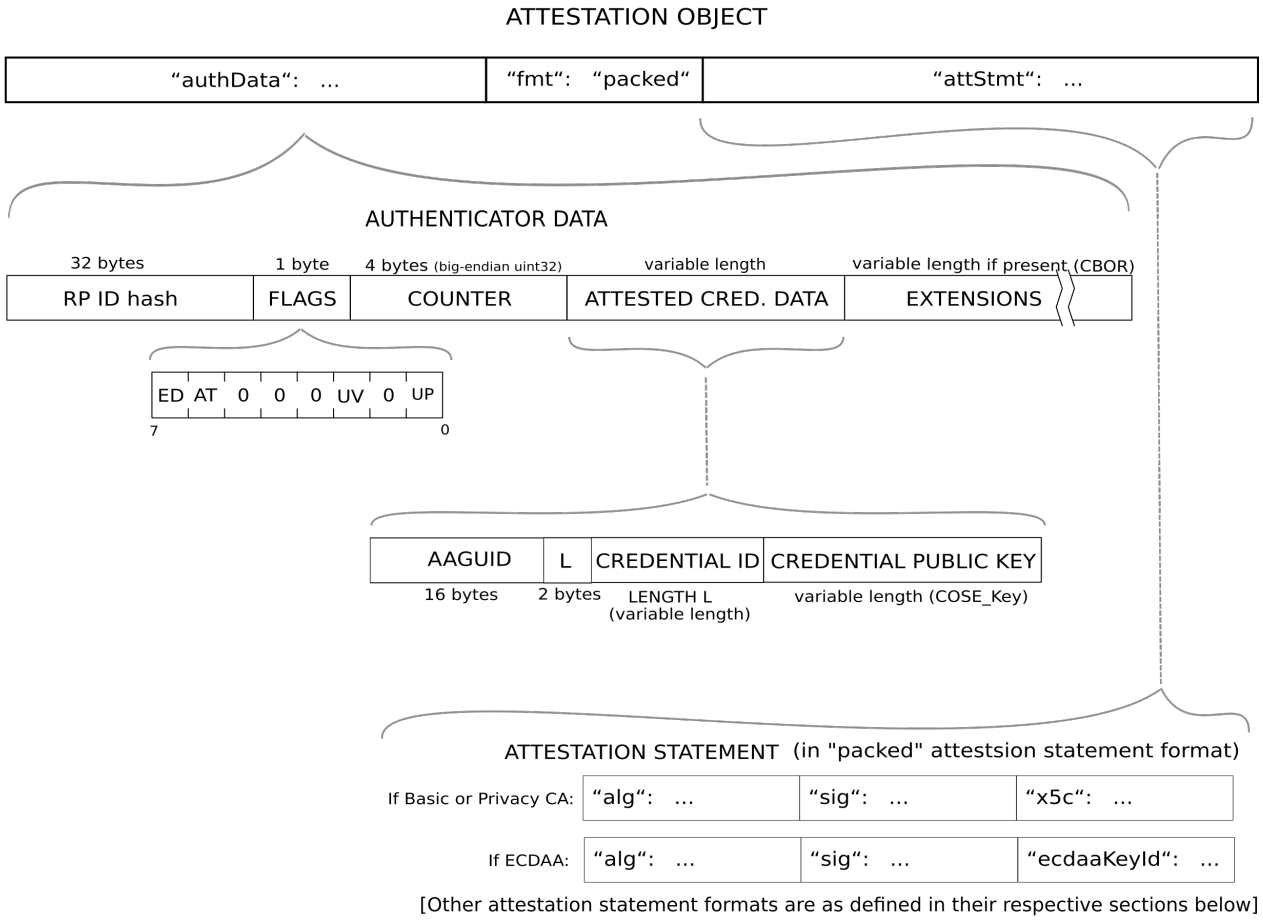

Figure 4.1.4: *Attestation object layout response for a make credential request. Obtained from (World Wide Web Consortium 2019)*

5.4.6 getAssertion API

The getAssertion API is requested by a client to give an assertion (cryptographic proof) from a previously generated credential with the authenticator for an RP (Alliance, FIDO 2019*a*, World Wide Web Consortium 2019).

In ctap_impl.c source file:

```
1956 uint8_t ctap_get_assertion(ctap_channel_t *p_ch)
```

GetAssertion receives the following parameters:

Parameter name	Data type	Required?	Definition
clientDataHash	Byte Array	Required	A hash of the client data composed by a type, the challenge, the origin, and a token binding (World Wide Web Consortium 2019). The type indicates if the operation will make a new credential or if it will get an assertion of an existing credential. The challenge is provided by the RP. The origin contains the FQDN of the RP. The token binding contains information of the token binding protocol (Popov n.d.) used between the client and the RP.

allowList	Sequence of Public-Key Credential Descriptors	Optional	A list of public-key credential descriptors, each of them is a structure that contains a previous credential created on the RP for the users account. The authenticator must generate an assertion for only one of the credentials present in this parameter (Alliance, FIDO 2019*a*).
extensions	CBOR map	Optional **and ignored**	A CBOR map of extensions which modify the operation of the authenticator, such as FIDO AppID or Simple Transaction Authorization (World Wide Web Consortium 2019).
options	Map of authenticator options	Optional	List of options which modify the operation of the authenticator. The options may be up (user presence) and uv (user verification). User presence requests to the authenticator to perform user consent. User verification requests to the authenticator verify the user with a gesture such as PIN or fingerprint. By default, up is set to true and uv set to false (Alliance, FIDO 2019*a*).
pinAuth	Byte Array	Optional **and ignored**	First 16 bytes of an HMAC-SHA-256 of clientDataHash using a pinToken obtained from the authenticator using the CTAP2 clientPIN API from a previous PIN created on the authenticator also using the same clientPIN API.
pinProtocol	Unsigned Integer	Optional **and ignored**	PIN protocol chosen by the client.

Table 6: Make Credential input parameters (Alliance, FIDO 2019*a*, World Wide Web Consortium 2019)

When an authenticator receives a get assertion request, it performs the following procedure (Alliance, FIDO 2019*a*, World Wide Web Consortium 2019):

1. If allowList is present, process all credential descriptors received. For each credential descriptor do:
   (a) Locate the respective credential stored in the authenticator. If the credential is not stored in the authenticator discard the credential received.
   (b) Verify that the credential is bounded to the rpId. If it is not the same rpId, discard the credential received.

```
1458 ret = parse_allow_list(GA, &map);
```

2. If allowList is not present, locate all credentials stored in the authenticator that belongs to the rpId.
3. Process all options received (Alliance, FIDO 2019*a*). If there is an unsupported option, then return error CTAP2_ERR_UNSUPPORTED_OPTION and stop the process. If an option is not valid, return error CTAP2_ERR_INVALID_OPTION and stop the process. Otherwise, continue with the operation.
   - ***Implementation note:*** *the option uv is not supported in the project implementation.*

4. Process all extensions if the parameter is present (Alliance, FIDO 2019*a*). If an extension is present and not supported return the error CTAP2_ERR_EXTENSION_NOT_SUPPORTED.

   - ***Implementation note:*** *no extension is supported in the project implementation. Therefore, extensions parameter is ignored and is part of future work for this project.*

5. Use the list of valid credentials from firsts steps. If the list of credentials is empty, return the error CTAP2_ERR_NO_CREDENTIALS and stop the operation.

6. If there is more than one credential in the list of valid credentials, only one must be selected.

   - If the authenticator has display capabilities, display each credential and let the user select the appropriate one.
   - Otherwise, order credentials by creation time and use the most recent.
     - ***Implementation note:*** *only the first credential is used in this implementation. Therefore, reorder or select a particular credential is part of future work for this project.*

```
2047 ret = cbor_encode_byte_string(&map, (uint8_t *)GA.creds[0],
     ↪  sizeof(CTAP_credentialDescriptor));
```

7. Obtain user consent with a gesture (Alliance, FIDO 2019*a*). This could be done testing user presence or performing user verification through a PIN or fingerprint. If user consent is received, continue with the operation. Otherwise, return error CTAP2_ERR_OPERATION_DENIED. The options for obtaining user consent are:

   - Detect user presence using the authenticator capabilities, such as pressing a button.
   - Verify user identity with the authenticator capabilities, such as a PIN, or a fingerprint.
     - If pinAuth is present and is not supported return error CTAP2_ERR_PIN_AUTH_INVALID, but if it is supported, verify it before continuing. If the verification fails, return error CTAP2_ERR_PIN_AUTH_INVALID. If it is not present and is set in the authenticator, return error CTAP2_ERR_PIN_REQUIRED.
   - If the authenticator has display capabilities, display user and RP information and requests user gesture. If the user rejects action, return error CTAP2_ERR_OPERATION_DENIED.
   - ***Implementation note:*** *this project only implements user presence with pressing a button. User verification through a PIN is proposed for future work.*

```
1992 bsp_board_leds_off();
1993 bsp_board_led_on(GREEN_LED);
1994 bsp_board_led_on(RED_LED);
1995 while( !is_user_button_pressed() );      // USER PRESENCE REQUIRED HERE
1996 bsp_board_leds_off();
```

8. Update the signature counter (World Wide Web Consortium 2019).

   - If the authenticator uses a global signature counter, increment it by some positive value.

```
1998 m_auth_counter++;
1999 /* Write the updated record to flash. */
2000 ret = fds_record_update(&m_counter_record_desc, &m_counter_record);
```

   - If the authenticator supports a per credential signature counter, increment it by some positive value.
   - If the authenticator does not support a signature counter, set the signature counter with cero constant.

9. Construct the authenticator data (without an attested credential data) containing (Alliance, FIDO 2019*a*, World Wide Web Consortium 2019):

```
2011 ret = ctap_make_auth_data(p_ch, &GA.rp, &map, auth_data_buf, &auth_data_sz,
     ↪  NULL, NULL);
```

- rpIdHash: a SHA-256 hash of the rpId.

```
1561 err_code = nrf_crypto_hash_calculate(
1562 &hash_context,                          // Context
1563 &g_nrf_crypto_hash_sha256_info,        // Info structure
1564 rp->id,                                // Input buffer
1565 rp->size,                              // Input size
1566 authData->head.rpIdHash,               // Result buffer
1567 &digest_size);                         // Result size
```

- flags: a byte indicating: if there was user presence and user verification, if the attested credential data and extensions data are included in the response.

```
1581 authData->head.flags = (1 << 0);     // User presence flag
1582
1583 if (credInfo != NULL)
1584 {
1585 authData->head.flags |= (1 << 6);    //include attestation data
```

- sigCount: the signature counter.

```
1577 authData->head.signCount = m_auth_counter;
```

- extensions data if any was processed in previous steps.

10. Create the assertion signature (World Wide Web Consortium 2019).
    (a) Concatenate the authenticator data and the clientDataHash.
    (b) Sign the concatenation with the respective credential private key.

```
2028 ret = ctap_calculate_signature(auth_data_buf, auth_data_sz,
     ↪  GA.clientDataHash, signature, &signature_size, &private_key);
```

11. Return to the client (Alliance, FIDO 2019*a*):

- credential: the public key credential descriptor bounded to the credential used to create the assertion signature. It can be omitted if there is only one credential in allowList.
- authData: the authenticator data constructed in previous steps.
- signature: the assertion signature obtained.
- user: the user entity bounded to the credential used for the assertion. The user entity is present only if user verification was performed.
- numberOfCredentials: the total number of valid credentials bounded to the authenticator and the rpId. It is omitted during getNextAssertion requests.

```
2062 ret = ctap_hid_if_send(p_ch->cid, p_ch->cmd, ctap_resp.data,
     ↪  ctap_resp.length);
```

In the figure below you can see graphically the assertion signature of a get assertion request.

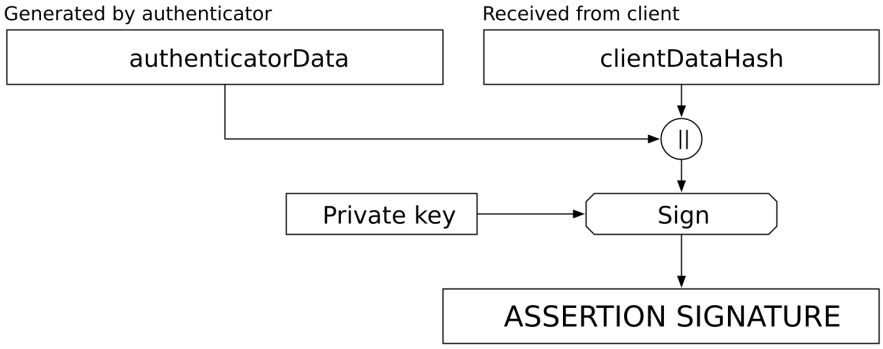

Figure 4.1.5: *Assertion signature for a get assertion request. Obtained from (World Wide Web Consortium 2019)*

6 Results and Evaluation

Despite some difficulties, the result of the project after ten weeks of work was successful. The CTAP2 protocol works as expected in its basics functions to perform credential registration and credential authentication. Therefore, the main result of this project is a new open-source FIDO2 project using the nRF52840 SoC. This project will contribute to extending the adoption of the FIDO2 protocol, and therefore, incrementing the security of users and organizations, and reducing the risks of current threads to the authentication process.

6.1 Python-FIDO2

The evaluation of the nRF52840 FIDO2 authenticator developed in this project with Python FIDO2 was sucesfull.

The example get_info.py was used to test the getInfo implementation with the authenticator as follows.

```
$ python3 get_info.py
CONNECT: CtapHidDevice(/dev/hidraw0)
CTAPHID protocol version: 2
request: b'\x04'
response: b'\x00\xa4\x01\x81hFIDO_2_0\x03P\xe6w\xcej\x86>^\xff(\x0eu\xcc\xbf\x1f\x
↪  99s\x04\xa3brk\xf4bup\xf5dplat\xf4\x05\x19\x04\xb0'
DEVICE INFO: Info(versions: ['FIDO_2_0'], aaguid:
↪  h'e677ce6a863e5eff280e75ccbf1f9973', options: {'rk': False, 'up': True,
↪  'plat': False}, max_message_size: 1200)
WINK sent!
```

And as it can be observed, the get_info.py request and response was **succesfull**.

The example credential.py was used to test the makeCredential and getAssertion implementations as follows. Due to the output of debugging requests and responses of those commands is very long, most of the output is omitted in the following example.

```
$ python3 credential.py
############ OUTPUT OMITTED INTENTIONALLY ##############
Touch your authenticator device now...
############ OUTPUT OMITTED INTENTIONALLY ##############
New credential created!
############ OUTPUT OMITTED INTENTIONALLY ##############
Attestation signature verified!
Touch your authenticator device now...
############ OUTPUT OMITTED INTENTIONALLY ##############
Credential authenticated!

ASSERTION DATA: AssertionResponse(credential: None, auth_data:
↪  AuthenticatorData(rp_id_hash:
↪  h'a379a6f6eeafb9a55e378c118034e2751e682fab9f2d30ab13d2125586ce1947', flags:
↪  0x01, counter: 1577058304), signature:
↪  h'30460221009cebd559fcbc8ba63275d0b2f7abdb5cd64cf75e817b6038fafd90de3cfccdeb02
↪  2100bfb4552eccf27ae58e6a5cc8ddffa892332263b41888ac9d7f45f422ee00f8bb')
Assertion signature verified!
```

6.2 Webauthn.io

Webauthn.io (DUO 2019) is a website designed to test the WebAuthn standard which is part of FIDO2. The website supports FIDO2 authenticators to sign-in and log-in.

The nRF52840 FIDO2 authenticator developed in this project was tested **successfully** in Webauthn.io.

Figure 6.2.1: *WebAuthn.io registration process*

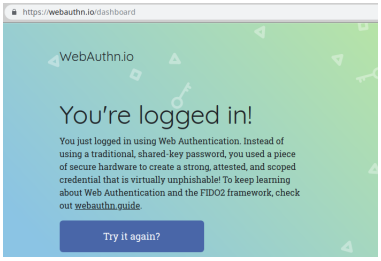

Figure 6.2.2: *WebAuthn.io authentication process*

6.3 Webauthn.me

Webauthn.me (Auth0 2019) is other website designed to test the WebAuthn standard which is part of FIDO2. The website supports FIDO2 authenticators to sign-in and log-in.

The nRF52840 FIDO2 authenticator developed in this project was tested **successfully** in Webauthn.me.

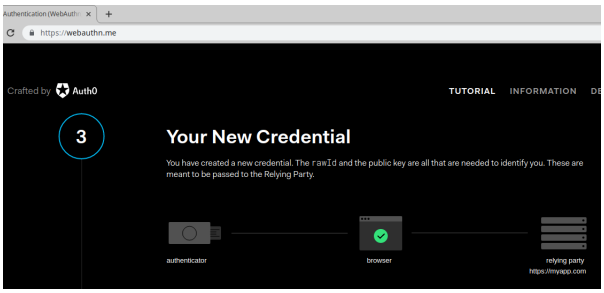

Figure 6.3.1: *WebAuthn.me registration process*

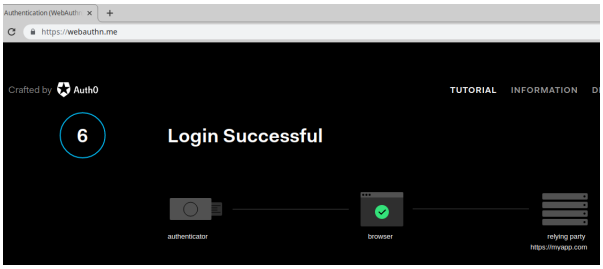

Figure 6.3.2: *WebAuthn.me authentication process*

6.4 Google

Recently, Google started to support FIDO2 authenticators to authenticate into a Google account (Dongjing He 2019).

The nRF52840 FIDO2 authenticator developed in this project was tested **unsuccessfully** in a Google account.

The security key registration process was succesfull, however, the authentication process produce an unexpected error in the authenticator which was not identified by time constraint.

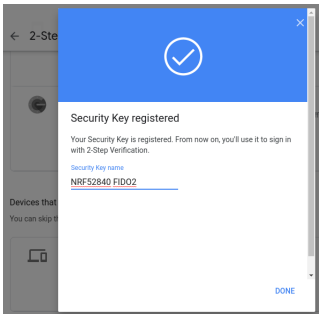

Figure 6.4.1: *Google account security key registration process*

6.5 GitHub

GitHub is the most popular hosting for version control system. It is used for many open-source projects.

Github started to support FIDO2 authenticators to authenticate into a Google account just a month ago (Tim Anderson 2019) .

The nRF52840 FIDO2 authenticator developed in this project was tested **successfully** in a GitHub account.

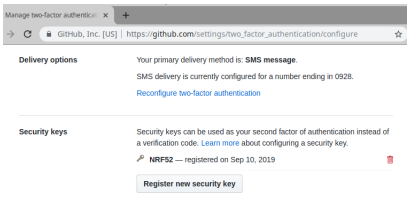

Figure 6.5.1: *GitHub account security key registration process*

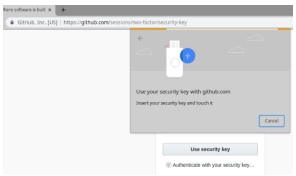

Figure 6.5.2: *GitHub account security key authentication process*

7 Future work

In this section, some interesting proposals are presented for future work. Most of these features were beyond the scope of the project due to the time constraint.

7.1 CTAP2 API

7.1.1 Get Next Assertion

A caracteristic which distinguish FIDO2 from U2F is that FIDO2 supports more than one credential per RP per authenticator. An use case would be, if you whish to have two Gmail accounts: one for personal use, and the other for your job.

The CTAP2 API Get Next Assertion allows an authenticator to select a different credential from a list of credentials bounded between the RP and the authenticator.

Therefore, it is important to develop this functionality.

7.1.2 Reset

Although the Reset function is a feature likely to be rarely used, it is very important to have for security reason. For example, when an authenticator has to be discarded or has to change ownership. If the credentials remain stored in the authenticator, it can be used to impersonate the previous owner.

7.1.3 Client Pin

The CTAP2 ClientPIN API is a great functionality to reduce more the security risks of authentication. It enables authenticators without a keyboard to set, update and validate a PIN. Therefore, even if a malicious user gets an authenticator from another user, she also needs to find the PIN which unlocks the credentials stored in the authenticator.

7.2 NFC and Bluetooth

During this project, the USB HID transport-binding was developed. However, the Bluetooth and NFC transports will allow this authenticator to be used in other use-cases such as a second-factor authentication in an online service with a smartphone with NFC or Bluetooth capabilities.

7.3 HMAC Secret

HMAC Secret enables an authenticator and a Client to authenticate securely using the FIDO2 protocol without the RP presence. This scenario is useful when the client is not always connected to the RP, such a personal computer. Therefore, this scenario will be very useful as more devices which are not always connected to an RP support FIDO2 protocol.

7.4 Secure Firmware Update

An additional security mechanism that will help to reduce the threads of this implementation is a Secure Firmware Update. This will make that the device only accepts firmware updates from a trusted sender. Therefor, an attacker cannot overwrite the firmware and take control of the authenticator.

References

Alliance, F. (2017*a*), 'Fido uaf architectural overview', *Proposed Standard, Feb* .

Alliance, F. (2017*b*), 'Universal 2nd factor (u2f) overview', *Proposed Standard, April* .

Alliance, FIDO (2017), 'FIDO 2.0: Overview', Available: `https://fidoalliance.org/specs/fido-v2.0-rd-20170927/fido-overview-v2.0-rd-20170927.pdf`. Online; accessed 28 August 2019.

Alliance, FIDO (2019*a*), 'Client to Authenticator Protocol (CTAP)', Available: `https://fidoalliance.org/specs/fido-v2.0-ps-20190130/fido-client-to-authenticator-protocol-v2.0-ps-20190130.html`. Online; accessed 28 August 2019.

Alliance, FIDO (2019*b*), 'Download Specifications', Available: `https://fidoalliance.org/specifications/download/`. Online; accessed 28 August 2019.

Alliance, FIDO (2019*c*), 'Specifications Overview', Available: `https://fidoalliance.org/specifications/`. Online; accessed 28 August 2019.

AppImage (2019), 'AppImage - Linux apps that run anywhere', Available: `https://appimage.org/`. Online; accessed 9 August 2019.

Arfaoui, G., Gharout, S. & Traoré, J. (2014), Trusted execution environments: A look under the hood, *in* '2014 2nd IEEE International Conference on Mobile Cloud Computing, Services, and Engineering', IEEE, pp. 259–266.

ARM Developer (2019), 'CryptoCell-300 family', Available: `https://developer.arm.com/ip-products/security-ip/cryptocell-300-family`. Online; accessed 20 June 2019.

Aumasson, J.-P. (2017), *Serious Cryptography: A Practical Introduction to Modern Encryption*, No Starch Press.

Auth0 (2019), 'Web Authentication (WebAuthn) Credential and Login Demo', Available: `https://webauthn.me/`. Online; accessed 29 July 2019.

Bormann, C. & Hoffman, P. (2013), 'Concise binary object representation (cbor)'.

Bus, U. S. (2001), 'Device class definition for human interface devices (hid)', *Version* **1**, 1996–2001.

Dongjing He (2019), 'Making authentication even easier with FIDO2-based local user verification for Google Accounts ', Available: `https://security.googleblog.com/2019/08/making-authentication-even-easier-with_12.html`. Online; accessed 10 Sep 2019.

DUO (2019), 'WebAuthn.io', Available: `https://webauthn.io/`. Online; accessed 29 July 2019.

Dworkin, M. (2001), 'Nist special publication 800-38a, recommendation for block cipher modes of operation-methods and techniques', *National Institute of Standards and Technology/US Department of Commerce* .

Ekberg, J.-E., Kostiainen, K. & Asokan, N. (2014), 'The untapped potential of trusted execution environments on mobile devices', *IEEE Security & Privacy* **12**(4), 29–37.

FIPS, P. (2009), '197, advanced encryption standard (aes), national institute of standards and technology, us department of commerce, november 2001', *Link in: http://csrc. nist. gov/publications/fips/fips197/fips-197. pdf* .

Grassi, P. A., Garcia, M. E. & Fenton, J. L. (2017), 'Digital identity guidelines', *NIST special publication* **800**, 63–3.

Hallsteinsen, S., Jorstad, I. et al. (2007), Using the mobile phone as a security token for unified authentication, *in* '2007 Second International Conference on Systems and Networks Communications (ICSNC 2007)', IEEE, pp. 68–68.

Hankerson, D. & Menezes, A. (2011), *Elliptic curve cryptography*, Springer.

Intel (2019), 'TinyCBOR: Concise Binary Object Representation (CBOR) Library', Available: `https://github.com/intel/tinycbor`. Online; accessed 9 August 2019.

Kim, J.-J. & Hong, S.-P. (2011), 'A method of risk assessment for multi-factor authentication', *Journal of Information Processing Systems* **7**(1), 187–198.

Künnemann, R. & Steel, G. (2012), Yubisecure? formal security analysis results for the yubikey and yubihsm, *in* 'International Workshop on Security and Trust Management', Springer, pp. 257–272.

Li, W., Chen, H. & Chen, H. (2019), 'Research on arm trustzone', *GetMobile: Mobile Computing and Communications* **22**(3), 17–22.

Makerdiary (2019*a*), 'nRF52 FIDO U2F Security Key An Open-Source FIDO U2F implementation on nRF52 SoC', Available: `https://wiki.makerdiary.com/nrf52-u2f/`. Online; accessed 29 July 2019.

Makerdiary (2019*b*), 'nRF52840 MDK USB Dongle', Available: `https://wiki.makerdiary.com/nrf52840-mdk-usb-dongle/`. Online; accessed 29 July 2019.

Nordic Semiconductor (2019*a*), 'Nordic Semiconductor Infocenter', Available: `https://infocenter.nordicsemi.com/`. Online; accessed 9 August 2019.

Nordic Semiconductor (2019*b*), 'nRF Connect for Desktop', Available: `https://www.nordicsemi.com/Software-and-Tools/Development-Tools/nRF-Connect-for-desktop`. Online; accessed 9 August 2019.

Nordic Semiconductor (2019*c*), 'nRF52840 System on Chip', Available: `http://www.nordicsemi.com/Products/Low-power-short-range-wireless/nRF52840`. Online; accessed 22 July 2019.

Popov, A. (n.d.), Nystroem, m., balfanz, d., and j. hodges," the token binding protocol version 1.0, Technical report, RFC 8471, DOI 10.17487/RFC8471, October 2018,¡ https://www. rfc-editor. org .

Sabt, M., Achemlal, M. & Bouabdallah, A. (2015), Trusted execution environment: what it is, and what it is not, *in* '2015 IEEE Trustcom/BigDataSE/ISPA', Vol. 1, IEEE, pp. 57–64.

Schaad, J. (2017), 'Cbor object signing and encryption (cose)'.

Sobti, R. & Geetha, G. (2012), 'Cryptographic hash functions: a review', *International Journal of Computer Science Issues (IJCSI)* **9**(2), 461.

SoloKeys (2019), 'Solo: open security key supporting FIDO2 and U2F over USB and NFC', Available: `https://github.com/solokeys/solo`. Online; accessed 29 July 2019.

Sunar, B., Martin, W. J. & Stinson, D. R. (2006), 'A provably secure true random number generator with built-in tolerance to active attacks', *IEEE Transactions on computers* **56**(1), 109–119.

Tim Anderson (2019), 'GitHub upgrades two-factor authentication with WebAuthn support', Available: `https://www.theregister.co.uk/2019/08/23/github_upgrades_its_twofactor_authentication_with_webauthn_support/`. Online; accessed 10 Sep 2019.

Ting, D. M., Hussain, O. & LaRoche, G. (2015), 'Systems and methods for multi-factor authentication'. US Patent 9,118,656.

Wikipedia contributors (2019*a*), 'Block cipher mode of operation — Wikipedia, the free encyclopedia', https://en.wikipedia.org/w/index.php?title=Block_cipher_mode_of_operation&oldid=914616923. [Online; accessed 8-September-2019].

Wikipedia contributors (2019*b*), 'Digital signature — Wikipedia, the free encyclopedia', https://en.wikipedia.org/w/index.php?title=Digital_signature&oldid=912810031. [Online; accessed 8-September-2019].

World Wide Web Consortium (2019), 'Web Authentication: An API for accessing Public Key Credentials', Available: https://www.w3.org/TR/webauthn/. Online; accessed 28 August 2019.

Yubico (2019*a*), 'Yubico Developers', Available: https://developers.yubico.com/. Online; accessed 9 August 2019.

Yubico (2019*b*), 'YubiKey 5 Series Technical Manual', Available: https://support.yubico.com/support/solutions/articles/15000014219-yubikey-5-series-technical-manual. Online; accessed 29 July 2019.

8 Appendix

8.1 Appendix A: Git Repository Location

The location of the git repository of this project can be found in `https://git-teaching.cs.bham.ac.uk/mod-msc-proj-2018/rxe841`.

8.2 Appendix B: Git Project Structure

During the process development, a few small projects were developed to test SDK and board functionality.

The main source code of the project is the CTAP2 folder inside the nRF52840_NORDIC_USB_Dongle.

```
/
├── docs
│   ├── report -- contains a digital copy of this report, and the LaTeX source
│   └── proposal -- contains a copy of the Project Proposal, and the LaTeX source
├── nRF5_SDK -- Directory to store the SDKs used for all projects
│   ├── nRF5_SDK_15.2.0_9412b96 -- SDK used to compile and test the Makerdiary nRF52
│   │   FIDO U2F Security Key implementation
│   └── nRF5_SDK_15.3.0_59ac345 -- SDK used to test the FIDO2 project implementation
│       and other small testing projects
├── nRF52480_MKD_USB_Dongle -- Testing projects implemented for this board
└── nRF52480_NORDIC_USB_Dongle -- Testing projects implemented for this board
    ├── ctap2 -- CTAP2 source code implementation for FIDO2 authenticator
    ├── tests -- Small projects to test SDK libraries
    └── u2f -- U2F port from Makerdiary project to Nordic nRF52840 Dongle
```

8.3 Appendix C: Running the software

To run the perform the following steps:

1. Install the nRF Connect for Desktop setup process described in section 5.
2. Perform the configuration described in UDEV rules in section 5.
3. Launch the nRF Connect.
4. Connect the Nordic nRF52840 USB Dongle in the USB port and press its reset button.
5. Using the nRF Connect, select the board from the panel and load the .hex file from the _build compilation directory.
6. Write the hex file into the board with the nRF Connect.
7. Test the board into WebAuthn.me, WebAuthn.io or other FIDO2 enabled service.