Humboldt-Universität zu Berlin
Mathematisch-Naturwissenschaftliche Fakultät
Institut für Informatik

An implementation of the FIDO2 TLS 1.3 extension in Rustls

Masterarbeit

zur Erlangung des akademischen Grades
Master of Science (M. Sc.)

eingereicht von: Emily Ehlert
geboren am:
geboren in:

Gutachter*innen: Prof. Dr. Jens-Peter Redlich
Dr. Ernst Günter Giessmann

eingereicht am: ....

Abstract

This thesis implements a FIDO2-based client authentication extension for TLS 1.3 in the Rust TLS library RusTLS, enabling secure, passwordless authentication for protocols beyond HTTP. By integrating FIDO2 into TLS, the solution provides an application protocol-agnostic method for client authentication. The implementation uses a double-handshake mechanism for registration and a single-handshake approach for authentication, ensuring compatibility with the existing FIDO2 TLS extension prototype for OpenSSL. As a compromise between security and speed, FIDO2 support is restricted to resident keys. The modified Rustls library is integrated into the MQTT server and client Rumqtt, demonstrating real-world usability. Performance analysis shows a server-side overhead of 426 microseconds per handshake. This work demonstrates the feasibility of FIDO2 integration into TLS, offering a foundation for broader adoption in secure communication systems.

Contents

1. Introduction **3**

2. Background **4**
- 2.1. TLS 4
- 2.2. FIDO2 6
- 2.3. Rust 8
  - 2.3.1. General 8
  - 2.3.2. Memory Safety 9
  - 2.3.3. Ownership 9
  - 2.3.4. Usage 10
- 2.4. Rustls 10

3. Previous Work **11**

4. Methodology **14**
- 4.1. Considerations 15
- 4.2. Overview 16

5. Implementation **20**
- 5.1. rustls 21
- 5.2. rustls-fido 22
- 5.3. Challenges 25
- 5.4. Cross-compatibility 25
- 5.5. Integration with rumqtt 26
- 5.6. Performance 27

6. Discussion **28**

7. Conclusion **30**

A. Connection Logs for Authentication **32**

B. Performance Data **33**

C. Repositories **33**

1. Introduction

Secure communication over digital networks is a cornerstone of modern computing, underlying everything from financial transactions to personal data exchange. The Transport Layer Security (TLS) protocol remains the de facto standard for securing data streams on the World Wide Web [1]. Positioned between the application and transport layers of the TCP/IP [2] model, TLS provides cryptographic guarantees such as confidentiality, integrity, and authentication [3]. Its most ubiquitous application is in HTTPS, where it secures web traffic over the Transmission Control Protocol (TCP) and Internet Protocol (IP).

While TLS mandates server authentication in its newest version, client authentication is optional [4] and rarely deployed in consumer-facing applications. Traditional certificate-based client authentication in TLS is cumbersome, requiring users to manage and transmit certificate files, which are more often than not stored unencrypted. To improve security, limited lifetimes and renewals are advised, further complicating deployment. This complexity has historically shifted the burden of user authentication to the application layer, where passwords and, more recently, FIDO2/WebAuthn, dominate. However, protocols beyond HTTP (e.g., IMAP, FTP, MQTT) lack robust authentication mechanisms, leaving them vulnerable to credential-based attacks.

The Fast IDentity Online 2.0 (FIDO2) standard, finalized in 2015, addresses these shortcomings by enabling passwordless authentication through hardware-based cryptographic tokens (e.g., YubiKey, Trusted Platform Module (TPM)) [5]. By leveraging public-key cryptography and challenge-response protocols, FIDO2 eliminates phishing risks and weak secrets. Yet, its adoption remains largely confined to web applications, leaving other protocols without modern authentication solutions.

The core challenge of this thesis is to extend the TLS 1.3 handshake. The goal is to introduce a new FIDO2-based client authentication mechanism, enabling protocol-agnostic security. This would reduce the reliance on passwords in protocols without needing to implement login FIDO2 logic themselves.

This problem has already been researched in previous papers [6] [7] [8] [9]. Challenges included sustaining security properties provided by TLS while integrating the challenge-response mechanism of FIDO2. Especially registering new security keys involved multiple round-trips, making the integration into a single TLS handshake difficult. Various prototypes in different programming languages and TLS libraries have been proposed.

This thesis aims to advance the research topic by integrating the FIDO2 TLS extension as designed by Panizza into the Rust TLS library RusTLS. The implementation is compatible with Panizza's FidoSSL. An extensive overview of the extension details is given. Furthermore, the modified RusTLS library is integrated into the MQTT application protocol using rumqtt, showcasing real-world usability. Finally, the performance impact is analyzed, discussing practicability.

2. Background

2.1. TLS

General

TLS is a protocol that provides a cryptographically secure communication channel over a digital network [10]. First specified in 1999, it succeeds the Secure Sockets Layer (SSL) protocol. Within the TCP/IP model, TLS is situated between the application and transport layers. It requires a reliable transport underneath [4], directly wrapping application data and providing a bi-directional bit stream. It is most prominently used in the HyperText Transfer Protocol (HTTP), where it secures connections over both TCP and QUIC.

TLS facilitates the authentication of the server but also the client through the use of X.509 certificates along with a cryptographic proof asserting ownership of the private key associated with the certificate [10]. While server authentication is required in TLS 1.3 [4], client verification is not. Since most application-layer protocols include their own mechanism, TLS's client authentication is very much underutilized. This thesis will focus on the most recent TLS specification, version 1.3. Standardized in 2018, this version introduces numerous essential security enhancements [11]. It modernizes the supported cryptographic primitives, reduces handshake time, and allows early application data [12]. Furthermore, TLS 1.3 removes support for static RSA (Rivest–Shamir–Adleman) and Diffie-Hellman (DH). In previous versions, a long-term, static RSA server key pair could be used to encrypt the client-generated pre-master secret [13]. By using only an ephemeral state during session key derivation, perfect forward secrecy is guaranteed. This means that past communication is secure, even if all long-term secrets are exposed.

TLS Handshake

A TLS connection is always initiated by the client. If no resumption information exists, a 3-way handshake is required. Otherwise, the client can utilize a pre-shared key from a previous TLS session to send encrypted application data (a so-called '0-RTT handshake') with the handshake. Client authentication would have then been completed during a previous session. This section will provide a brief overview of the handshake procedures. For a complete explanation, please refer to Rescorla [4].

Every handshake starts with a *ClientHello* message sent from the client. It includes, among other information, the protocol version and a set of speculative key shares for the DH key exchange, utilizing various cipher suites. This first message is only encrypted if the connection is resumed or the Encrypted Client Hello (ECH) extension is configured. When the message is received, the server proceeds to generate a key share itself and packages it with other information, such as the chosen cipher suite in the *ServerHello* message. Both sides use the key shares to derive the session key. Further messages are encrypted. *EncryptedExtensions* contains responses to extensions initiated through the *ClientHello*. If client authentication is used, a *CertificateRequest* message is sent. If no pre-shared key authentication is used, the server must send a *Certificate* and

CertificateVerify message to provide its public certificate and cryptographically prove the ownership of the associated private key. The server side is completed with a *Finished* message, containing a Message Authentication Code (MAC) over the handshake to prove integrity. Back on the client side, if client authentication is enabled, a *Certificate* and *CertificateVerify* message are necessary. Otherwise, only the *Finished* message to prove integrity is sent back to the server. After the handshake is completed, the client can start sending application data. Should the MACs not match on the server side, any data is discarded.

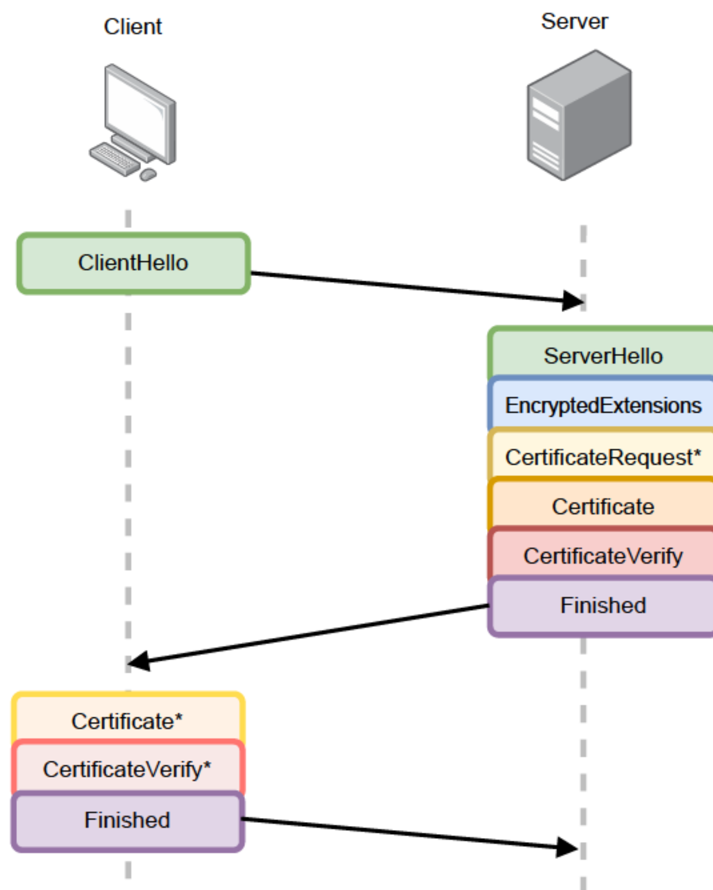

Figure 1: Flow of a full TLS 1.3 handshake with client authentication using certificates as specified by [4]. Messages marked with a star are optional. While theoretically optional, server certificate authentication is assumed.

Many of TLS's messages, such as `ClientHello`, can be extended using extensions. They

can not modify the message flow, but provide additional data. One such example is the Application-Layer Protocol Negotiation (ALPN) extensions, which allow the client and server to negotiate which version of the protocol used on top of TLS should be used [14]. These extensions are standardized by the TLS Working Group (TLSWG). However, the specifications allow for custom extensions to supplement the 'official' ones.

2.2. FIDO2

General

FIDO2 is a suite of protocols designed to authenticate clients using a cryptographic authenticator such as a security or platform key [15]. Submitted to the Web 3 Consortium (W3C) in 2015 by the FIDO2 Alliance [5], it aims to address some of the weaknesses of traditional authentication by eliminating the need for a password. Thus, solving issues of low entropy, password phishing, and the reuse of secrets [16].

FIDO2 operates primarily at the application layer in the TCP/IP model, where it is most often used in the context of HTTP. It builds on the earlier Universal 2nd Factor (U2F) specification and consists of two main components: the Web Authentication API (WebAuthn) standardized by the W3C for JavaScript and the Client to Authenticator Protocol (CTAP), maintained by the FIDO2 Alliance.

Authentication is provided through a challenge-response flow between a client and the cryptographically bound Relying Party (RP; server), ensuring the user's presence and consent. Additionally, user verification (such as PIN or biometrics) is supported directly by the authenticator. During registration with an RP, a key is generated. This application-specific key can either be stored on-device (resident key) or on-server (non-resident key). Non-resident keys offer high scalability and minimal storage requirements on the authenticator itself, making it the preferred option. On the other hand, resident keys do not need a username, making them ideal for scenarios where interactivity is discouraged.

The FIDO2 framework is designed with privacy and security in mind. Authenticators do not share any personal information between services, and cryptographic assertions are tied to specific RPs through guarantees made through TLS. Most authenticators can verify the user through another factor, such as biometrics, making even classical two factors, such as timed codes, redundant.

Protocol Flow

All steps are also outlined in the table 1. Initially, the client signals its support for FIDO2 to the relying party (RP). If this authentication method is selected, the process commences. This process involves interactions between the RP, the client browser, the platform, and the authenticator, utilizing the WebAuthn and Client to Authenticator Protocol (CTAP). It encompasses either the registration of a new credential or the verification of an existing one. While a concise overview of the process is outlined below, comprehensive technical specifications are available in W3C Web Authentication Working Group [17] and FIDO Alliance [18].

For both registration and authentication, the RP transmits a request to the client, which includes a cryptographic challenge and additional parameters such as the RP's identity (ID), supported cryptographic algorithms, and authenticator preferences. The client subsequently interacts with the authenticator using CTAP, either to generate a new credential or to utilize an existing one.

Phase	**Participants**	**Action / Message**
Registration		
1. Initiate Registration	User → Client	Client indicates support for FIDO2 to RP and requests registration
2. Send Registration Request	RP → Client	RP generates a challenge and credential creation options and sends them
3. Call to Authenticator	Client → Authenticator	Client uses CTAP to send registration request with options from RP
4. Generate Key Pair	Authenticator → Client	Authenticator creates a new key pair and returns the attestation object
5. Send Registration Response	Client → RP	Client sends the attestation response back to RP
6. Verify and Store	RP	RP verifies the attestation, challenge, and stores the public key with the user account
Authentication		
1. Initiate Login	Client → RP	Client indicates support for FIDO2 to RP and requests authentication
2. Send Authentication Request	RP → Client	RP generates a challenge and optionally allowed credential IDs, then sends an assertion request
3. Call to Authenticator	Client → Authenticator	Client uses CTAP to send an authentication request with options from RP
4. Sign Challenge	Authenticator → Client	Authenticator signs the challenge using the stored private key and returns the assertion
5. Send Authentication Response	Client → RP	Client sends the assertion (signature, client data) to RP
6. Verify	RP	RP verifies the assertion signature and client data to authenticate the user

Table 1: FIDO2 Registration and Authentication Flow

During registration, the authenticator generates a new public–private key pair and securely stores it together with related metadata, such as the relying party (RP) ID. It then creates an attestation object for the new credential, which includes the public key and a signature over the authenticator data and the hash of the client data (containing the challenge). This signature is produced using the manufacturer's private attestation key, allowing the authenticator to prove its legitimacy to the RP.

During authentication, the authenticator signs a combination of the authenticator data and the hash of the client data using the private key previously associated with the RP ID and origin. The resulting signature, which is bound to the TLS-verified RP ID, along with the user handle and credential ID, is sent to the RP, which verifies the signature and authenticator data against the information stored from registration.

Similar to TLS, FIDO2 does support extensions both during registration and authentication. This can, for example, be used to get additional device information or signal enterprise-level features.

Overall, FIDO2 provides a modern and cryptographically strong authentication framework. It reduces the reliance on passwords and is resistant to common threats such as phishing or low-entropy secrets. Over the years, it has seen a steady increase in support from such companies as Google, Amazon, or Apple [19]. However, since FIDO2 is a standard envisioned for the web, support for other protocols, such as IMAP or FTP, is limited to nonexistent.

2.3. Rust

2.3.1. General

Rust is a modern programming language developed initially as a side project by Graydon Hoare before being picked up by Mozilla in 2009 [20]. It tries to bridge the gap between the performance and control of low-level programming languages such as C while retaining memory safety usually attributed to high-level languages like Java [21]. It accomplishes this by smartly applying theoretical research on programming languages to practical features such as ownership [20]. This allowed Rust to remove its garbage collection early on, eliminating periodic, random interrupts. On benchmarks such as calculating the first 10000 digits of pi, Rust can match the performance of the fastest C/C++ implementation [22].

Rust code is usually 'safe'; however, sometimes it is imperative to disregard the memory safety checker to directly handle pointers. This can be done by wrapping code in an 'unsafe' code block. If a program contains a single unsafe block, the memory safety can no longer be guaranteed by the compiler [23]. The problem now is that most libraries and even the Rust standard library contain unsafe code [24]. The developers claim that the usage is manually secured, but there have been soundness bugs like access to freed memory [25]. While this may seem concerning, Jung *et al.* [24] formally proved that the logic on a reasonable subset of the Rust programming language is sound. The proof is extendable to libraries, laying out what needs to be checked to verify soundness.

While Rust is an imperative programming language, it also supports declarative

programming [23]. Furthermore, object orientation using `structs` is limited but possible. This is all packaged with a comprehensive system and package manager, Cargo, allowing programmers to easily start and extend their projects with a wealth of libraries.

All these properties make Rust the most admired programming language in 2024, according to the Stack Overflow developer survey [26]. However, in real-world uses, only 12.6 % of developers said that they had extensively used Rust in the past year. While this fact may make Rust seem insignificant, it is important to know that the first stable release was only published in 2015 [20]. According to TIOBE, which measures search engine queries for programming languages, Rust has seen a stable rise in popularity [27].

2.3.2. Memory Safety

One may wonder, what are the problems traditional, low-level languages like C have that Rust aims to fix?

Around 70 % of security bugs are memory-safety related [28]. Common issues are using memory after it has been freed, data races on shared variables, and unexpected changes and side effects. Ensuring that a program, if it compiles, is memory safe would improve the vulnerability landscape significantly. This idea was the main driving force behind the development of Rust. To accomplish this, one of the main techniques the language employs is ownership [23].

2.3.3. Ownership

Ownership is a third way of thinking about management besides garbage collection and manual management. It does not incur any runtime penalty since ownership rules are checked at compile time, but it also prevents bugs when memory is self-managed by the developer. The basic rules are rather simple. When data is stored on the heap, there must be exactly one pointer to it. This variable holding the pointer is called the owner. There must only be one owner at the time. If the owner goes out of context, the data is freed. If the variable is copied, the previous variable (pointer) becomes invalid. This is called a move, and it transfers ownership. To perform a deep copy, the `.clone()` function is used.

Listing 1: Rust Ownership Example

```
1  fn main() {
2      // Let's create a String on the heap
3      let a = String::from("hello");
4      // Now transfer ownership
5      let b = a;
6      // a is invalid
7      println!("{}", a);
8  }
9
10 // when trying to compile, will get the following error:
   error[E0382]: borrow of moved value: `a`
```

To have multiple "pointers" to a variable, Rust employs a reference system. A reference points to the owner of a variable. A reference must not outlive the owner. This is managed through lifetimes. There can be either multiple immutable references or at most one mutable reference. This is mostly to prevent data races. One would not expect the data to change if one holds an immutable reference to it.

2.3.4. Usage

Since Rust compiles down to machine code but also brings memory safety at no additional cost, it has become popular for implementing system-level services and drivers. For example, since Linux version 6.1, Rust code can be found in the kernel, and progress has been steady to make it equal to C [29]. Rust was also introduced in the Android Open Source Project (AOSP) to provide, among other things, safer concurrency [30]. Furthermore, Windows started using Rust in 2023 for some data types, but also parts of the graphics library Win32 GDI and system calls [31].

2.4. Rustls

Rustls [32], first released in 2016 [33], is a TLS library aiming to provide a Rust-native, platform-independent TLS implementation [34]. As of writing, the current version is 0.23.28, released on the 16th of June 2025 [35]. It implements both the client and server for TLS 1.2 and 1.3. The cryptographic provider can be swapped. The default is aws-lc-rs [36].

When designing Rustls, much care was taken to ensure past mistakes in implementation, such as OpenSSL or GnuTLS, are not repeated [37]. The protocol implementation is written in safe Rust to prevent memory bugs like Heartbleed in OpenSSL [38]. Rustls enforces the TLS state machine by making message processing consume the current state and return a new state [37]. This is different from the classical, reactionary approach of many TLS libraries. These are often susceptible to State Machine AttaCKs (SMACK) as outlined by Beurdouche *et al.* [39].

Over time, TLS has had many security vulnerabilities [40]. This led the Rustls developer to only implement a reduced subset of features specified in TLS 1.2 – and to some extent 1.3 – to mitigate these design errors. Rustls does not support any export, 64-bit block, or CBC ciphers. Furthermore, RSA key exchanges, renegotiations, and compression are also not implemented [41].

Rustls only supports future-proof cryptographic primitives such as X25519MLKEM768 for a post-quantum key exchange [42]. The library is opinionated, with a focus on providing a secure and concise TLS implementation. The developers chose not to support all standardized extensions, as well as the custom extension mechanism found in, for example, OpenSSL. The reason stated is that the extension could modify the protocol flow in unforeseen ways, reducing security.

Thanks to its modern code base, besides the security gains, Rustls is significantly faster than OpenSSL [43]. It can perform more handshakes and has a higher application data transmission bandwidth, as seen in figure 2.

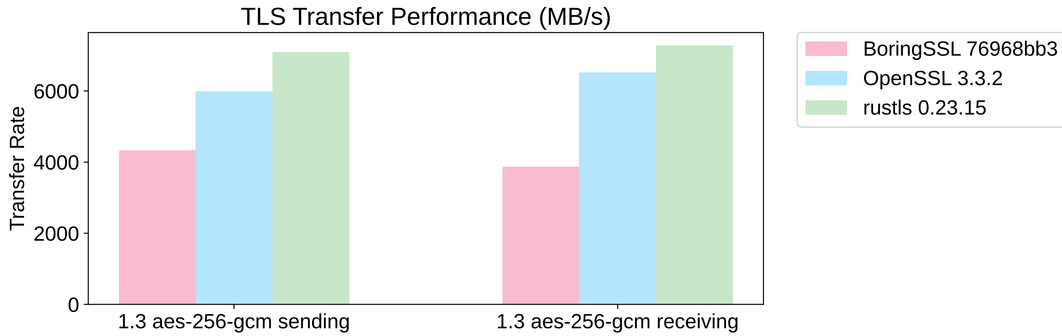

Figure 2: Comparison of TLS library application data transfer performance between BoringSSL, OpenSSL, and rustls. Tested on an Intel Xeon E-2386G with AVX-512 and AES-NI. Data from [43].

All these properties make Rustls seem like a possible contender to overtake OpenSSL in mainstream usage. Since 2024, the library can be used as a cryptographic provider for nginx using a compatibility layer [44]. The Rust crate reqwest, which is the most popular HTTP library in Rust, is in the process of making rustls the default TLS provider [45], switching off the platform native implementation (such as OpenSSL on Linux).

3. Previous Work

Initial considerations

Initial work for the TLS extension has been laid out by Tom-Lukas Johann Breitkopf [6]. In their bachelor's thesis titled "FIDO2 als TLS-1.3-Erweiterung" (FIDO2 as TLS 1.3 extension), they explored the idea of implementing client authentication using FIDO2 security tokens as a replacement for user certificates. Potential advantages pointed out are the secure authentication provided to the application developers, potentially even removing the need for password-based authentication completely. The extension could bring strong authentication to protocols with traditionally weak authentication, like the Internet Message Access Protocol (IMAP), or enhance VPN protocols such as OpenVPN, which utilizes a TLS tunnel for key exchange [46]. Negative consequences could be the increased complexity of the TLS protocol, increasing the attack surface in one of the most used security protocols. Furthermore, FIDO2 authentication would slightly increase latency for users. Questionable is also the adoption willingness for non-technical users, as highlighted also in other research [47]. The goal was to implement registration and authentication using FIDO2 resident keys. The criteria for the addition to the library were:

- Must not increase implementation complexity significantly
- Must retain security guarantees such as perfect forward secrecy made by TLS

- Must keep latency added through additional round trips to a minimum

The author proceeded to evaluate possible handshake mechanisms. The biggest roadblock was the transmission of the user name when registering, since it is required for the server-based challenge generation. Using a static key for encryption in the *ClientHello* would reduce forward secrecy. Other options would increase complexity too much or change the TLS messaging order. In the end, they decided on the double handshake mechanism. A first handshake is used to transmit a user name and ephemeral identifier, and the second one for the actual challenge-response mechanism. Authentication is contained in a single handshake, since it does not require transmission of a user identifier for resident keys. The proposed extension would add new messages to the core TLS protocol to facilitate the FIDO2 request and response protocol, going above what a normal TLS extension can do. I believe that this approach would lead to slow adoption due to requiring significant modifications in TLS libraries. Furthermore, this would complicate the state machine with more possible state transitions. The author also created a proof-of-concept implementation in Python utilizing the `tlslite` library.

Implementation in GnuTLS

In Mario Freund's bachelor's thesis, "FIDO2-Erweiterung von TLS 1.3 in einer gängigen Kryptobibliothek" (FIDO2-Extension of TLS 1.3 in a common cryptography library) [7], they investigated how well the proposed extension can be implemented in a mainstream TLS library. In their work, the authors use the TLS protocol modification as proposed by Breitkopf. After reviewing the proposal, they continue with selecting a library to implement the changes in. The candidates are OpenSSL, WolfSSL, and GnuTLS. WolfSSL was rejected due to its advanced code structure, making extension efforts difficult. While OpenSSL offers an extensible state machine, due to its high functionality, the code size is very large. Finally, the author decided to go forward with GnuTLS due to its compact size and state machine implementation.

The implementation only implements the authentication of FIDO2 tokens, but not the registration. However, in turn, both resident and non-resident keys may be used. For non-resident keys, a second handshake is required. For the FIDO2 authentication, the server needs to maintain an SQLite3 database and connect to a WebAuthn server for the actual challenge generation and verification. The database is for the storage of ephemeral registration information (id, user name, and valid_until). The WebAuthn server manages actual registered user information. If non-resident keys are used, it also stores the encrypted key material. An architecture overview can be found at figure 3.

While the approach seems sensible, the usage of an external server for authentication seems disadvantageous. This adds complexity to the TLS system and increases latency by 30 ms for the single handshake approach.

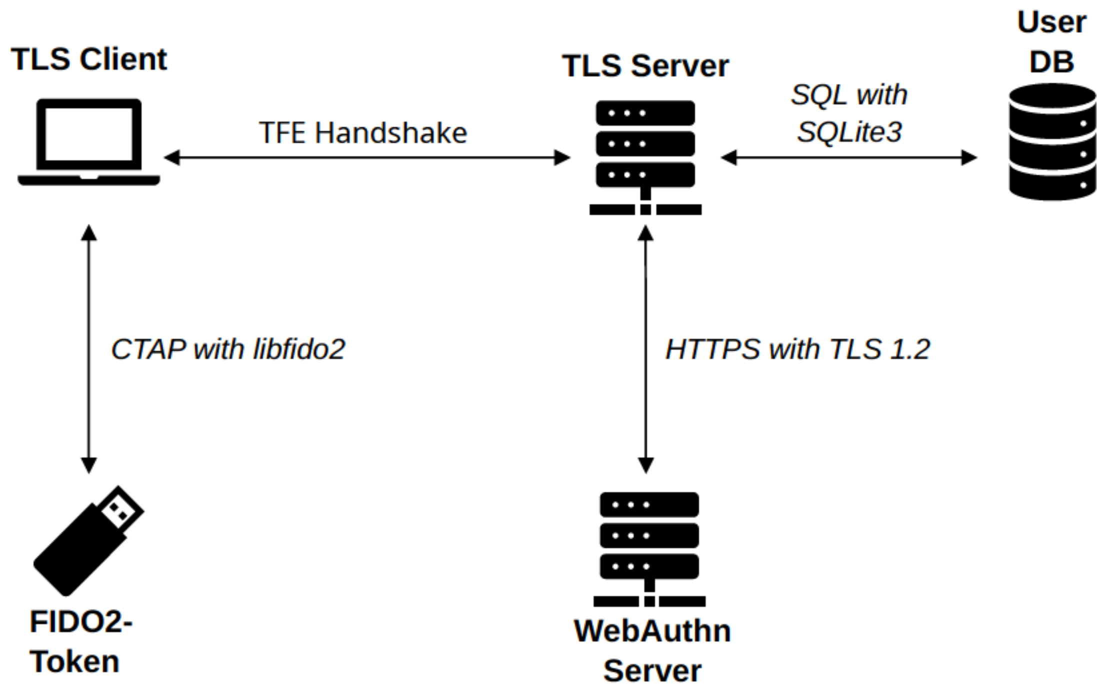

Figure 3: Conceptual overview of Freund's GnuTLS TLS 1.3 with FIDO2 implementation. Translated from the original [7].

Implementation in OpenSSL

The master's thesis "FIDO2 TLS 1.3 Extension: Strong EAP-TLS Authentication for 802.1X Networks" by Jonas Panizza [8] concerns itself with the implementation of the FIDO2 extension in OpenSSL using the custom extension mechanism as well as testing FIDO2 authentication in the EAP-TLS protocol.

The author modified the existing FIDO2 extension flow to better work with the TLS extension mechanism. OpenSSL implements the custom extension mechanism, allowing Pannizza to use an unmodified OpenSSL version. Benefits of this approach include a reduced potential for vulnerabilities compared to direct code modification and leaving a trustworthy base. To achieve this, the FIDO2 messages were packaged in a TLS extension, needing a handshake message to be attached to.

The FIDO2 request message is sent with the TLS *CertificateRequest* message, and the FIDO2 response from the client is an extension of the *Certificate* message. In Panizza's implementation, a mock client certificate is required to fulfill the OpenSSL state machine requirements. The approach reduces implementation complexity by allowing the TLS state machine transitions to be mostly unchanged. Registration still requires two handshakes, while authentication can be achieved in a single handshake. In contrast to Freund's work, only resident keys are supported. The failure of the FIDO2 process is communicated using TLS alerts, repurposing existing ones as defined by the TLS 1.3 RFC [4].

A further improvement made by the author is the addition of symmetric encryption in the registration phase to prevent a Monsters-In-The-Middle (MITM) replay attack. During the first handshake, the server provides a symmetric key for later use. Then, in the second handshake, the server encrypts some parts of the FIDO2 registration request. While an attacker could replay the second *ClientHello* with the unencrypted ephemeral identifier, it would not be able to decrypt the server's response due to the lack of the encryption key. A further challenge during registration is the permission to register a token. While during WebAuthn the user is usually already in an authenticated context, this is not the case for TLS. The author solved this problem by employing a ticket-based approach with a registration secret provided out-of-band by the server and consumed when a token was successfully added.

In the latter part of their thesis, Panizza integrates the modified OpenSSL into the Extensible Authentication Protocol (EAP), specifically EAP-TLS. This variant uses TLS to authenticate a client with a certificate against a TLS server and subsequently against the application employing EAP-TLS, such as a Wireless Local Area Network (WLAN). A FIDO2 implementation could simplify both the registration and authentication procedures by having the credentials stored on a token and not in a file. Registration could easily be handled by adding the desired token to the network using a web interface.

Overall, the work advanced the research on FIDO2 with TLS significantly by streamlining the modifications into custom extensions directly supported by the protocol. The author made further security improvements and even demonstrated the feasibility of the modification with EAP-TLS. My main critique is the many incongruences between the thesis and the actual source code provided. This circumstance made analyzing the implementation significantly more challenging.

Implementation in JSSE

During the writing of this thesis, Gloria Geise published her master's thesis on the extension of the Java Secure Socket Extension (JSSE) [9]. In it, they use a double-handshake procedure to register and authenticate a token with a resident key in TLS 1.3. The work does not contain many new advances and is adjacent to the works of Panizza and Freund.

4. Methodology

The goal of this thesis is to provide an extension to the TLS 1.3 protocol to extend client authentication. Traditionally, certificates were the main choice when authenticating clients on the TLS layer. However, that has always come with a plethora of pain points, such as distribution and renewal. By making the authentication in TLS more attractive through the use of FIDO2, it could gain broader appeal and support. By employing FIDO2 tokens, client registration, security, and roaming capabilities would be significantly improved. Some research has already been completed on this topic. Rust was chosen as a target programming language since it provides bare-metal performance while providing memory safety that is checked at compile time.

4.1. Considerations

The goals formulated here are adjacent to the design principles given by Panizza [8].

- **Minimal Invasion** – Since Rustls does not allow custom extensions, the source code of the library must be modified. To ensure higher acceptance, the changes should be small, with most application logic code being outsourced to an optional library.
- **Interoperability** – The Rustls extension should work with the extension to OpenSSL done by Panizza.
- **Security** – No security guarantees, such as confidentiality, authenticity, and forward secrecy, should be impacted. Personal information used for FIDO2 also needs to adhere to the same standards.
- **Performance** – The impact on the handshake time should be low. No second handshake should be required in case of authentication.
- **Authentication** – The implementation should perform a valid FIDO2 authentication, ensuring proper identity verification.
- **Clean and safe code** – The code should be well-written and easy to understand, with little need for additional comments. It should also be straightforward to review.
- **Thread safety** – Since Rustls supports multiple threads, the extension must also be thread-safe. There must not be any race condition.

During registration, the client needs to transmit personal information, such as a user name, to the server. Only with that information can a challenge be created. Since the *ClientHello* message is not encrypted, another way must be found to transmit this data. Two primary methods were considered: extending the TLS handshake with new messages or employing a double handshake approach, where the first handshake establishes a secure channel for transmitting personal data, and the second handshake completes the authentication process.

Extending the handshake requires modifying the possible handshake states, which would involve significant modifications to the Rustls library and ultimately the TLS specifications. However, it streamlines the registration and authentication process, making it transparent to the TLS library user. In addition, no state must be kept between handshakes, since the registration can be completed as a whole. The question would remain, how many libraries would implement such extensive changes to their implementations just to support a single extension?

By using a double handshake, no major modifications to the TLS state machine are required. The extension data would be restricted to existing messages. Using the *CertificateRequest* and *Certificate* messages seems to be ideal for the challenge-response-based process. These messages are usually only sent when a client certificate-based authentication is desired. Thus, even when using FIDO, a client certificate would

still be required to conform to the TLS 1.3 standard. If the extension were to gain enough acceptance, a future TLS version could reframe the certificate messages into *ClientChallengeRequest* and *ClientChallengeResponse* messages. Another challenge with this approach is that it would require the application using Rustls to initiate a second handshake by itself while also keeping the inter-handshake state. If any application would implement this is open to discussion.

The double handshake approach proposed by Panizza was chosen to maintain compatibility with the existing FidoSSL implementation. Modifying the handshake message would introduce incompatibilities and prohibit the FIDO2 authentication from being a TLS extension.

A possible improvement would involve performing the pre-registration out of band, resulting in a personalized setup code that contains both a one-time identifier and an encryption key. No additional personal information would need to be exchanged, and registration could be performed in a single TLS handshake when using resident keys.

FIDO2 supports two key storage modes: resident keys and non-resident keys. With the first, all key material is stored on the token itself. With the second, the encrypted key is stored at each server where the token has an account. Non-resident keys involve the client transmitting a user identifier to the server to return the correct key to the authenticator. This requires an encrypted communication channel to provide confidentiality. In a TLS handshake, the first message the client sends is not encrypted. Thus, a second handshake would be required, impacting performance. Otherwise, the compromise would be to either transmit the user name in plaintext or use the Encrypted Client Hello (ECH) extension. ECH in TLS 1.3 allows the first message to be encrypted by using a long-term public key stored with the DNS record of the server, requiring additional configuration [48]. If the private key were compromised, all transmitted and stored user names could be deciphered. This means using the ECH approach would break forward secrecy. Since supporting non-resident keys would either require reducing security guarantees or an additional handshake during authentication, a decision was made not to support them.

While previous works designated a separate FIDO2 server to handle user authentication, this thesis implements the complete FIDO2 server logic into the rustls-fido library itself. By embedding the procedure, performance will be improved.

To be compatible with OpenSSL, great care must be taken to ensure the API is identical. The FidoSSL source code has to be inspected carefully so the extension data can be parsed by both sides.

4.2. Overview

The project is logically split into two distinct parts: registration and authentication. While the latter is contained in a single handshake, registration in-band requires two sequential connections to transmit user data encrypted. A helper application exists that can register a token with a server out of band, simplifying the integration with existing application layers.

Registration

The registration process is separated into two TLS handshakes. The first handshake is used for the client to transmit its personal details and for the server to establish an ephemeral user identification and symmetric encryption key for the second handshake. In that, the actual FIDO2 registration procedure is performed.

Pre-Registration: A client must decide beforehand if it wants to register and authenticate. In the *ClientHello* TLS message, the FIDO2 Indication extension message is added. The CBOR-encoded data contains a message type `0x01` marking it as a pre-registration indication. No additional data is contained.

The server-side *CertificateRequest* message incorporates the FIDO2 Request extension message. To facilitate this, a 256-bit-long identifier and a 256-bit key used for AES-256-GCM are randomly generated. The AES key is essential to mitigate replay attacks during registration. The server temporarily stores the identifier and key in an in-memory hash map for the second phase of the registration and includes them in the extension.

Upon receiving the request, the client uses its FIDO2 token to register a new credential. The authenticator's response is then sent back in the FIDO2 Response extension structure, accompanying the *Certificate* TLS message. This response includes the user name, display name, and a ticket. Registration validation relies on a ticket-based mechanism: the ticket is a secret shared out-of-band between the server and client that authorizes the creation of a new token. Currently, the ticket is static and is not bound to any specific account information.

When the server receives the *Certificate* message, the FIDO2 registration procedure is started, already saving the registration request for later. Another variant would involve only communicating the ephemeral identifier and encryption key during the first handshake. In the second FIDO2 indication, all the user context, including the ticket, could be transmitted. This would reduce the amount of state needed to be kept between handshakes, reducing the effectiveness of possible Denial-Of-Service (DOS) attacks through the reduction in memory consumption. However, changing the FIDO2 extension flow would require significantly changing the FidoSSL implementation, which is not the scope of this thesis. Thus, the user context is transmitted during the FIDO2 response of the first handshake.

Registration: After terminating the first TLS session, the client connects again. The FIDO2 indication in *ClientHello* contains the message type, i.e., registration, and the ephemeral user identifier. If no pre-registration state exists, the FIDO2 registration is disregarded, and no *CertificateRequest* is sent out. Depending on whether client authentication is required, the TLS handshake can continue or be aborted.

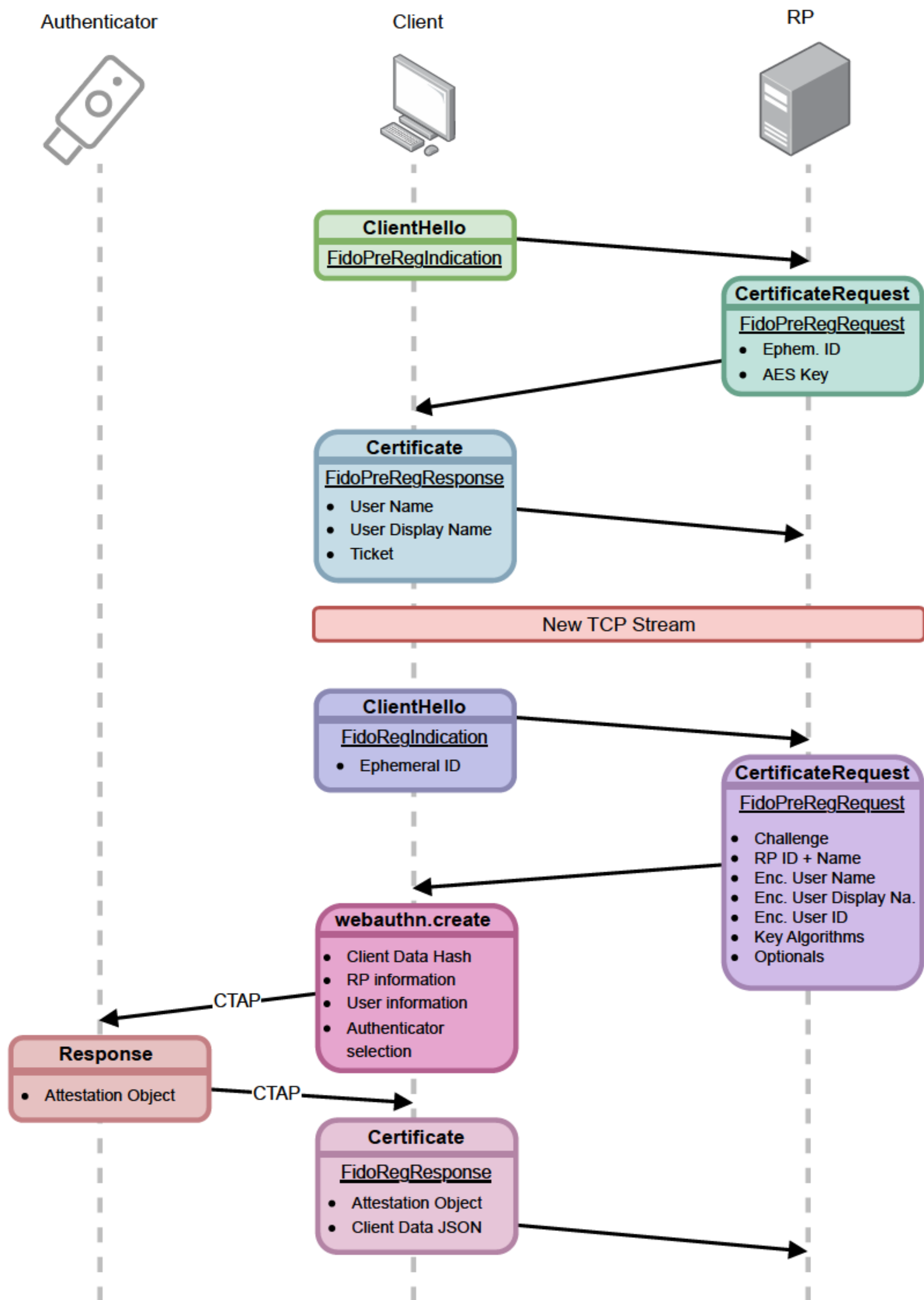

Figure 4: Overview of TLS-FIDO2 client authentication for FIDO2 token registration

The prepared FIDO2 registration request is taken from an in-memory map, identified through the ephemeral user identifier. The request contains all the information that the client will need to perform the registration with their FIDO2 passkey. If an attacker were to replay the second *ClientHello*, it would not gain any insights, since all fields containing personal details are encrypted. Furthermore, the request contains the RP identifiers and the registration challenge. Additionally, more optionals, such as timeout, excluded credentials, or authenticator selection, can be specified.

After the client receives the message, it prepares and performs the registration with the passkey. A PIN must be configured through client configuration. Currently, only USB-based tokens are supported. Should user confirmation be required, a significant delay is added to the connection process. If the key creation is successful, the token generates a CTAP-encoded attestation object, which is sent back alongside some client data to the server in the FIDO2 response extension.

The server then validates the response and stores the passkey data, including the public key, counter, attestation, and some other information in a database associated with the user and credential ID to be used for later authentication.

Authentication

The authentication is integrated into a single handshake. Thus, no additional round-trips are added to the more often used authentication. When the client wants to connect to the server, it also includes the FIDO2 indication in the *ClientHello*. The message type `0x7` indicates to the RP that the user wants to authenticate and not register. No additional data is included.

At this stage, the server is unaware of the user's identity, requiring a discoverable authentication. The user can only be determined from the response. The server prepares a FIDO2 request containing a challenge and optional fields, including the required RP ID. To ensure compatibility with FidoSSL, this identifier remains within the optional structure.

After the client receives the *CertificateRequest* message including the FIDO2 request extension, it prepares and performs the authentication with the attached token. If successful, an assertion is returned, containing information on the signature, key, user, and authenticator flags. All those are then packaged into the FIDO2 response and sent back to the server.

The server then uses the response to identify the correct user and the credential ID used. Based on these, the stored passkey can be retrieved from the database. The passkey, in addition to the client response, is then utilized to validate the authentication.

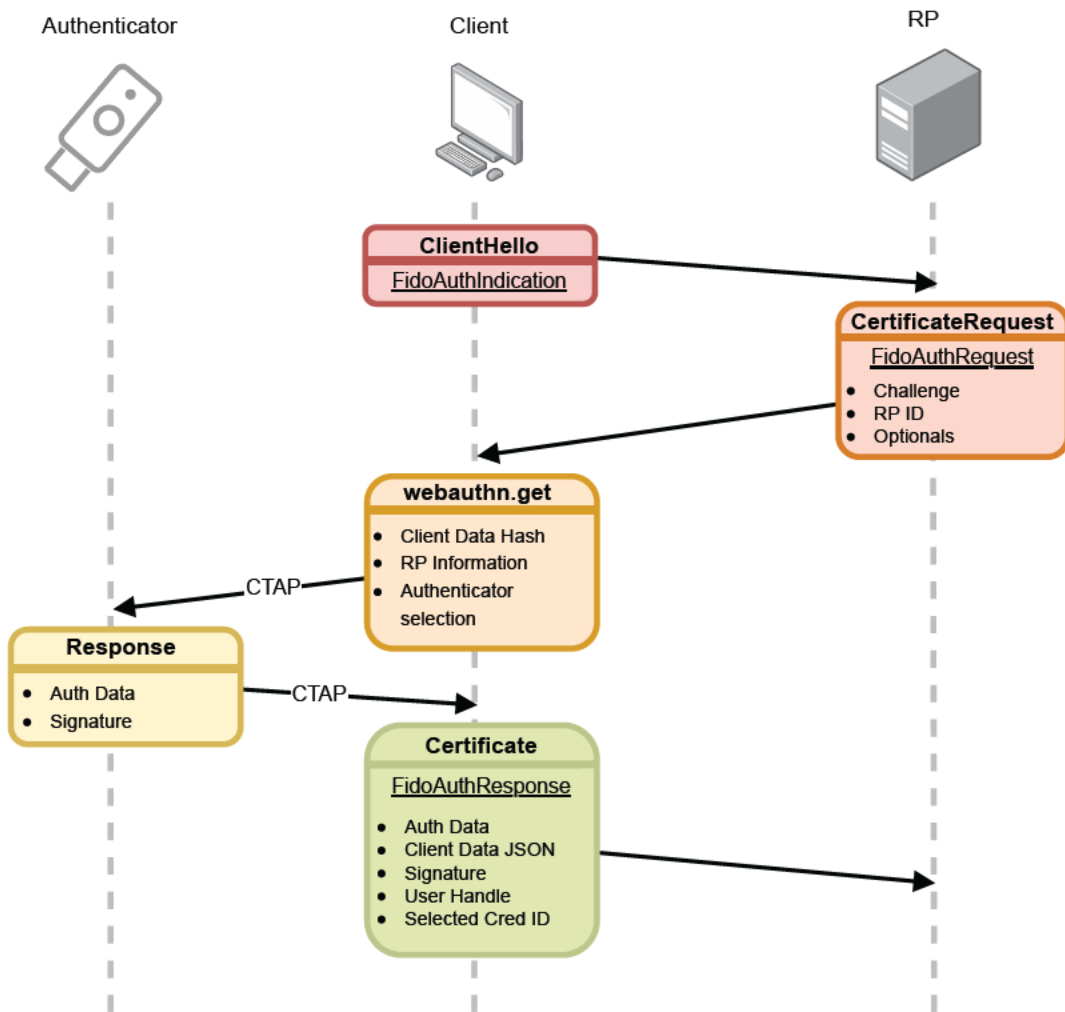

Figure 5: Overview of TLS-FIDO2 client authentication for FIDO2 token authentication

5. Implementation

Since Rust is a lesser-known language, more thought is given to explaining the implementation and considerations, which may differ from traditional languages such as C.

The direct modifications to Rustls are kept in a fork of the GitHub project repository. Most of the FIDO2 logic is separated into an additional library. All repositories used can be found in appendix C.

The choice to outsource most of the code has been influenced by the desire to keep possible modifications to Rustls small. Since most users of the library do not use client authentication and thus also not FIDO, it would be wasteful to add that much "dead"

code. Thanks to the modern Cargo build system for Rust, a library can expose optional features with optional dependencies. Another benefit of the separate Cargo project is the ability to include binary targets for registering (and authenticating) FIDO2 tokens out-of-band.

5.1. rustls

To integrate rustls-fido2 with Rustls, a few sections of code had to be modified. The first is the configuration of the TLS client/server. For this, the developers of the library used a builder pattern, where different aspects, such as server certificate and client certificate verifier, could be modified. Each stage of the "build" process requires some option. The process can be seen as a primitive state machine. For example, a server configuration will first need a certificate verifier or client verification to be explicitly disabled. During each step, the current state (such as *WantsVerifier*) will be consumed and a new state returned. A new state *WantsFido* was added to allow FIDO2 configuration for the server. This is separate from the client certificate verification, since some may want to have both options enabled at the same time. Client configuration works differently by just being another disjoint option for client authentication. Users likely either want to use a certificate or FIDO, and not both.

To hold all FIDO2 state, a `fido` field is added to the client and server configuration data structure. It includes both configuration values, such as the registration ticket, and fields to store the (pre)registration state, the database connection, and the response buffer for the client. While this mixes different types of information, splitting the configuration and data would have needed significantly more changes to how the Rustls state machine handles data. Since Rustls can handle multiple connections at the same time, a mutex-based approach was chosen to prevent threading issues. In Rust, variables that are shared between threads must also include some kind of reference counting, so they are only deleted if the last reference is dropped. Through this mechanism, mutexes cannot become invalid while still being used.

Extension Modification

The available extensions needed to be extended with all the different types sent with the FIDO2 extension. Fortunately, Rustls is designed in a way that makes adding new extensions simple. The encoding and decoding of the extension data were also added. Rustls uses the Rust trait system to provide generic functionality for serialization. A trait is somewhat equivalent to an interface in object-oriented programming languages. They define shared behavior, such as in this case, how to encode and decode the FIDO2 extension messages. The serialization framework serde with the serde_cbor library was selected. With it, serializations can be defined universally and conveniently. Further discussion on the challenges can be found further down.

Adding extension hooks

The message processing was modified for states where a FIDO2 extension message would be included. In general, the modifications consisted of checking if the extension was contained and then, depending on the current FIDO2 state, i.e., pre pre-registration, registration, or authentication, process the incoming extension and potentially generate a response. While the server only needs to reply to the FIDO2 indication, it requires some data flow manipulation so the reply is sent out with the correct message. The reply state is kept inside the connection-specific state machine, since remembering the correct connection in a multithreaded application context would be more challenging.

5.2. rustls-fido

Library selection

FIDO2 Server

The client- and server-side logic are split into separate files. When deciding between using a dedicated FIDO2 server and keeping the functionality embedded, the focus was on speed, ease of use, and conformity. A promising, dedicated solution was the FIDO2 authentication server developed by Line, line-fido2 [49]. It is written in Java, providing a REST API and official certification. While the feature set is extensive, there were concerns about making the implementation more complex through the need to interact with the server through the API. Furthermore, having a dedicated application running to facilitate FIDO2 authentication would hinder the ease of use and thus the testing of this prototype extension. Therefore, focus was put on finding a Rust library capable of providing comprehensive FIDO2 server functionality. While the Rust library store crates.io has many popular FIDO2 client libraries, only one, webauthn-rs, fulfilled all the needs of the project. It is open-source, security audited by the SUSE project team, and has more than one million downloads. They also pledged to strictly adhere to the WebAuthn 3 standard. The documentation is extensive, allowing developers to gain a quick overview of all the capabilities, such as resident-key-based authentication.

Storage

A disadvantage of the webauthn-rs library is that it does not handle storage. Even with discoverable authentication, it is required to save some user-related data during registration, enabling a user to be correctly identified during authentication. For storage, the decision was between file-based, NoSQL, and Structured Query Language (SQL) options. While file-based storage does not require additional dependencies, it would require manually associating the passkey data provided by the FIDO2 library with relevant authentication information, such as the user ID and usage counter, making it impractical for anything beyond basic testing. NoSQL databases offer flexibility and scalability but may not provide the relational integrity or consistency guarantees required for authentication systems. Furthermore, since the data will be structured, flexibility is not required. SQL databases, on the other hand, are well-suited for structured data like user credentials, offering ACID (atomicity, consistency, isolation, durability) compliance and robust querying.

Among the SQL options, PostgreSQL and SQLite stood out. PostgreSQL is a modern, fast, and feature-rich database designed for scalability, concurrency, and high availability, making it ideal for multi-server, load-balanced environments. However, for this prototype, SQLite's lightweight, serverless design and simplicity made it the more pragmatic choice. It eliminates the overhead of managing a separate database server and keeps the library easy to test and deploy. That said, if the system were to evolve into a multi-server, load-balanced architecture, transitioning to PostgreSQL would be beneficial for its support of concurrent connections, distributed transactions, and replication.

I have worked with SQLite before in Rust using the rusqlite library. While it is only synchronous and single-threaded for testing and prototyping, it should be more than enough.

FIDO2 token communication

For the client, a library was needed that enabled communication with FIDO-based tokens using the CTAP protocol. After researching available options, two promising libraries emerged: passkey-rs and authenticator, developed by 1Password and Mozilla, respectively.

The passkey-rs library advertises itself as a comprehensive WebAuthn and CTAP2 client implementation, which initially seemed like a strong match. However, upon closer inspection, it became clear that its support is primarily limited to its own virtual authenticator. It either lacked compatibility with physical USB-based tokens or was not documented. While perhaps good for benchmarking, support for physical FIDO2 tokens, preferably connected via USB, is essential.

On the other hand, the authenticator library is explicitly developed for CTAP communication over USB with security keys. The Mozilla developers even provided a binary example to manage security tokens, such as viewing and deleting currently stored FIDO2 entries. This made integrating the library into the project straightforward.

Serialization

To follow Panizza's OpenSSL implementation, the FIDO2 extension data needs to be CBOR-serialized. In Rust, Serde was the obvious choice for this. It is practically the de facto standard for serialization in Rust. It uses a trait-based system with derive macros. Serde lets developers serialize and deserialize complex data structures with almost minimal boilerplate. For CBOR specifically, the serde_cbor crate integrates seamlessly. Thanks to Serde, the code can be kept simple and explicit. The library is fast, well-tested, and widely used, with over 640 million downloads, which mattered for a security-sensitive feature like FIDO.

Program flow

For the server-side, the extension provides functions to start and finish registration and authentication of clients. The process needs to be split, since first a challenge is created and later on the authenticator's response is verified.

To register, the client transmits user data in the pre-registration phase, which is linked with an ephemeral user ID. Then, in the second phase of registration, the actual FIDO2 registration can be started. For this, a passkey registration challenge is created with

webauthn-rs and then sent to the client. A passkey registration data structure is also generated, which needs to be stored with the server to process the response. When the response is received by the server, it is used in combination with the stored data to finish registration. The resulting passkey data structure is stored with associated data in the SQLite database.

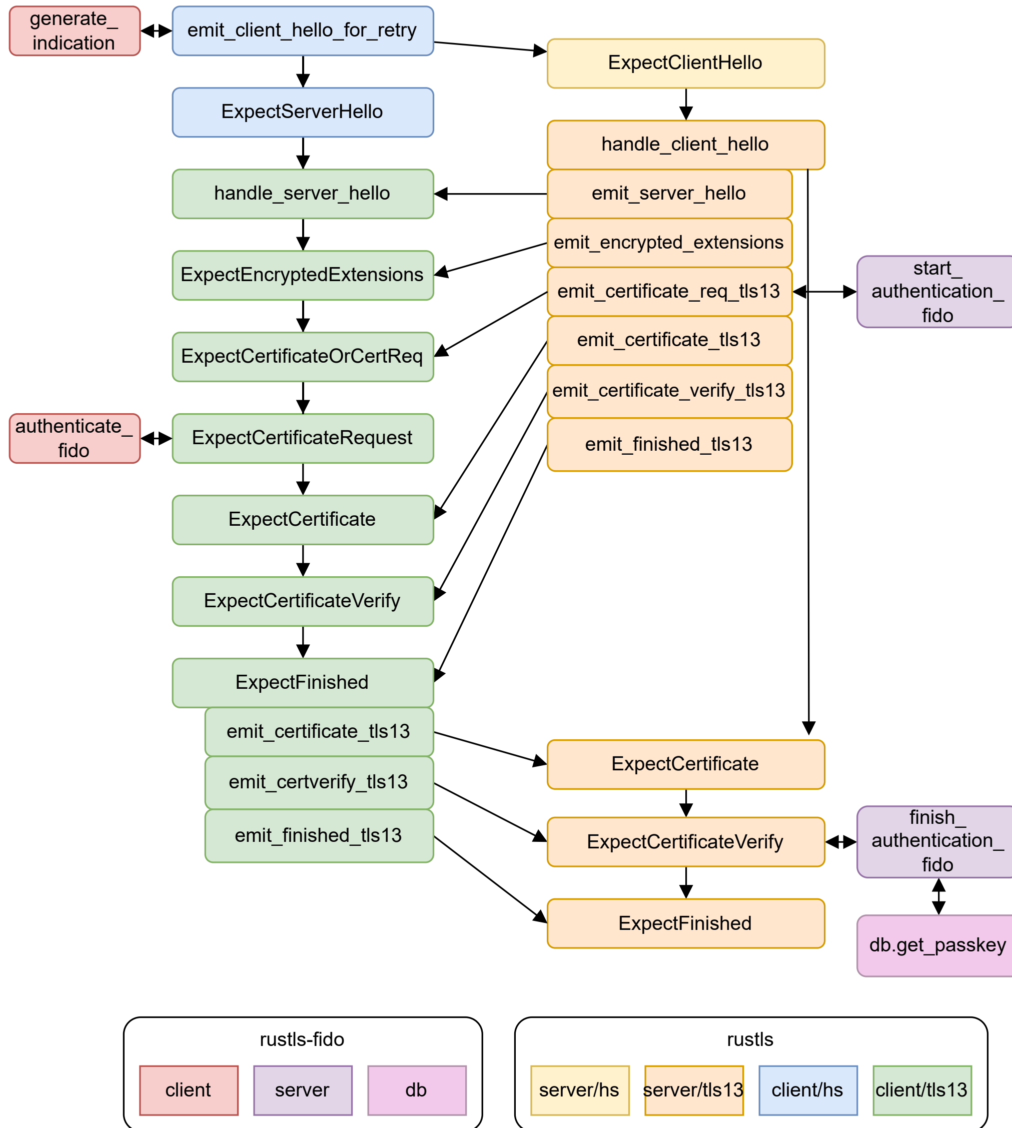

Figure 6: Flow of a handshake using FIDO2 client authentication between a Rustls client and server instance. Colors associate functions with their respective files in the rustls and rustls-FIDO2 projects.

During authentication, the process is similar. However, since only resident keys are used, the authentication challenge generation does not require any user information. To process the challenge result, the correct credential must first be identified. The authenticator should include the credential ID in the response, which then can be used to identify the correct passkey data structure stored in the server's database. With it, the authentication process can be finished. This is handled by the webauthn-rs library. It is important to verify afterward that the credential counter stored is not larger than the counter provided by the authenticator itself, which would indicate that the FIDO2 token has been copied or manipulated. If the check succeeds, the counter is updated in the database.

The client-side implementation can be simpler due to receiving only a single message containing all relevant request information. The authentication function takes the request sent from the server to prepare a CTAP-compatible data argument structure. The authenticator library uses a channel-based approach to communicate the result of the authentication challenge back to the extension. The channel is read until a result from the authenticator is received. The security key provides authentication data and a signature, which need to be CBOR-encoded and then retained in a response buffer before being sent out. For registration, the token returns an attestation object proving itself to be genuine.

5.3. Challenges

Extending Rustls to include FIDO2 authentication required overcoming various challenges. Firstly, understanding the project structure, as with any larger application, took a significant amount of time. Rustls enforces a strict separation of concerns with cryptography, I/O, and protocol logic split. With the actual protocol, the process is divided between the server, client, and common logic. Then, for each TLS version, such as the one this thesis uses, 1.3, a different state machine is employed. Each state, such as *ExpectCertificate*, consumes its own state when processed and generates a new one.

Another implementation detail involved matching the data produced by the authenticator library and the security token with the expectations of the FIDO2 server library webauthn-rs. Even though both adhered to the WebAuthn standard, some fiddling with the data was still required, such as encoding the attestation object manually into CBOR.

5.4. Cross-compatibility

One of the goals of this master's thesis was to extend Rustls so that it is cross-compatible with the FidoSSL implementation developed by Panizza. Thus, ultimately, all protocol design decisions had to be directly copied. This turned out to be somewhat difficult due to the data structure definitions printed in the thesis, which were not accurate with the provided implementation. This circumstance required taking definitions directly from the source code. Furthermore, some mistakes and omissions were contained in the extension. The bugs were so severe that a working implementation would not be possible without some modifications to the FidoSSL project. Most prominently, the CBOR-encoded authentication data provided in the attestation object during authentication was not

included in the client response. Furthermore, Panizza reused the same TCP connection to perform the second handshake during registration in their example client binary. This is not TLS 1.3 compliant and thus not possible with Rustls. I created a pull request on the associated repository, integrating all necessary changes into FidoSSL [50]. An updated message description can be found in the appendix C.

The last obstacle was matching the CBOR-encoded extension output generated by the Serde serialization with the one produced by FidoSSL. Various problems needed to be solved:

- Per default, Serde encodes every Rust vector as an array, which, for a byte vector, wastes a lot of space. However, a small helper library provides some functions to serialize these as byte strings.
- The FIDO2 authentication and pre-registration indication have the same structure and are only distinguishable by the message type value. Thanks to Serde's modularity, it is possible to write a custom implementation for the deserialization of the FIDO2 indication, using the message type as the distinguishing feature.
- Due to Panizza's decision to start the numbering of entries in the optionals map with 1, a custom serializer and deserializer were required for all map-based data structures.

After fixing these issues, FidoSSL can be used both as a server and client to register and authenticate for a proper TLS 1.3 handshake. For logs, see A. When using FidoSSL as the server, the cryptographic response from the FIDO2 token is not properly validated as required by the WebAuthn standard.

5.5. Integration with rumqtt

To build upon previous works, this thesis integrates the modified Rustls library within a Rust application. The ambition was to find an application protocol that uses TLS but does not support FIDO2 authentication in itself. Furthermore, there must be at least a decently popular application or library written in Rust that implements the protocol and uses Rustls for its secure connection handling. These requirements reduced the possible candidates significantly. Promising libraries, such as lettre, a mail client, only implement either the client or server side, but not both. Rumqtt is a library implementing the Message Queueing Telemetry Transport (MQTT) protocol in Rust. It provides both a client and a server while also having the option to use Rustls as its TLS backend.

For the client, no code inside the rumqttc library needed to be modified. An application wishing to use FIDO2 authentication would need to enable the *use-rustls* feature. Otherwise, the operating system's native library, such as OpenSSL under Linux, would be utilized. Then, in the actual application, one can supply a custom Rustls configuration to the rumqttc transport interface. When using the modified Rustls version, a FIDO2 client configuration can be specified. Only authentication is supported due to registration requiring a double handshake logic, which would overcomplicate the implementation. Registration can be done out-of-band by the registration application supplied with

the rustls-fido2 project. With it, a key will be generated on the FIDO2 token, and a corresponding entry in the database will be made.

Since the server is a standalone daemon, some code for the rumqttd needed to be modified. In general, a new feature flag, *verify-client-fido*, was added to the cargo project. Furthermore, rumqttd uses the Tokio TLS wrapper to simplify TLS handling, necessitating modification in the tokio-rustls project as well to change to the local modified Rustls version. In the rumqttd code, a FIDO2 server configuration was added. Configuration is possible using environmental variables.

After registering a YubiKey 5C NFC out of band, authentication of the client against the MQTT server, as well as sending and receiving messages, is possible.

5.6. Performance

Measuring the overall time penalty added through the introduction of the extension is difficult. If a USB-based token is used, usually some kind of action verification with a button press or fingerprint scan will be required. Platform-based authentication could use embedded security chips such as TPM 2.0. But even these will usually prompt the user. While there exists a project implementing a FIDO2 token in software named SoftFIDO2 [51], it has not been maintained for more than three years, making it incompatible with current Linux kernels and unable to compile. Therefore, measuring latency increases on the client-side does not seem sensible.

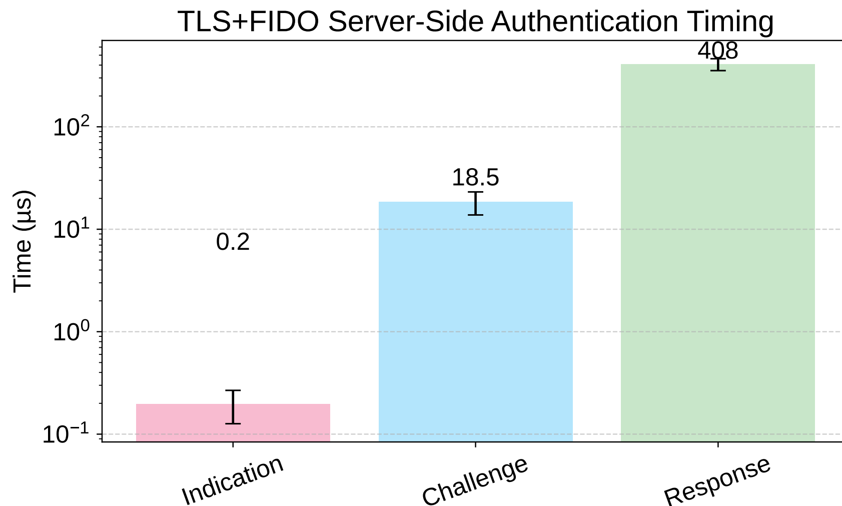

Figure 7: Server-side performance measurement for a TLS 1.3 handshake using FIDO2-based client authentication. Categorized by FIDO2 extension state. n=100. Data in table 2.

On the other hand, profiling the server could identify potential performance bottlenecks. TLS servers should be able to handle thousands of connections. If a handshake is extended

due to FIDO2 verification, this could significantly reduce the handshake rates. To measure impact, three different points of the handshake were measured: receiving the indication, generating a challenge, and verifying the response. It is expected that the verification will take the longest, due to requiring cryptographic verification. Furthermore, the database will also need to be queried, which, while not taking CPU resources, will still slow down the request.

For time measurement, the Rust function `Instant::now()` was used, which uses the monotonic clock available through the `clock_gettime` system call. The clock resolution, according to the `clock_getres` system call, is 1 nanosecond. In total, 100 authentications were measured. Over the data, I calculated the mean and standard deviation. The system used for testing was an Intel i7-4790k.

As expected, verifying the client's response is responsible for almost all the delay, with processing the indication and generating the challenge being negligible. In total, 426 microseconds of delay are added. When using the AES-256-GCM-SHA-384 cipher suite with X25519MLKEM768 for key agreement and no client authentication, the complete handshake takes 1.9 ± 0.1 ms. A simple certificate-based authentication takes 2 ± 0.2 ms. Thus, FIDO2-based authentication only adds 0.3 ms compared to using a certificate. In a server environment with modern processors and newer instruction sets, it is expected that the performance for the cryptographic operations would be significantly increased. It is also important to note that, while the TLS handshake is increased, no authentication later on will be required, freeing up compute resources. Furthermore, it is reasonable to assume that a WebAuthn handshake over HTTPS would likely incur a higher time penalty through the added protocol layer.

6. Discussion

Security

The primary goal of this work was to enhance TLS 1.3 with FIDO2 authentication while preserving the protocol's core security guarantees: confidentiality, integrity, and forward secrecy. By employing a double-handshake mechanism during registration, the implementation ensures that sensitive user data is transmitted exclusively over encrypted channels. Authentication itself requires only a single handshake, made possible through the restriction to resident keys.

However, the choice to use a double handshake for registration introduces added complexity and latency. Although this trade-off was necessary to maintain compatibility with existing TLS state machines and avoid invasive modifications, it raises concerns about the feasibility of widespread adoption. Future iterations of TLS could address this by introducing dedicated messages for client authentication, reducing the need for multiple round-trips, though this would likely require sufficient demand to be considered. Another approach could be through the use of an out-of-band pre-registration.

Compatibility

A key objective was to ensure compatibility with existing FIDO2 implementations, particularly Panizza's FidoSSL. Achieving this required careful alignment with its protocol definitions, which were not always clearly documented. Discrepancies in the CBOR serialization format and the handling of optional fields necessitated reverse engineering and direct code inspection. While these challenges were ultimately resolved, they highlight the need for standardized, well-documented protocols to facilitate interoperability across different implementations.

The successful integration with rumqtt, a Rust-based MQTT library, demonstrates the practical applicability of this extension. By enabling FIDO2 authentication for MQTT—a protocol traditionally reliant on weak or application-layer authentication—this work paves the way for stronger security in IoT and real-time communication systems. However, the current implementation lacks dynamic registration support within the MQTT server, limiting its immediate usability in production environments.

Usability

Despite its technical advantages, the adoption of FIDO2-based TLS authentication faces several challenges. The requirement for physical tokens, such as YubiKeys, or platform-based authenticators, like TPM 2.0, may deter non-technical users, especially in consumer applications where passwords remain the default. Educating users about the benefits of FIDO, such as resistance to phishing and credential theft, will be crucial for broader acceptance.

Modifying TLS libraries to support FIDO2 extensions requires significant effort, and not all libraries, such as OpenSSL or GnuTLS, may be willing to adopt these changes. The Rustls implementation demonstrates feasibility, but widespread adoption will depend on community interest and the perceived value of the extension.

Existing systems relying on certificate-based authentication may resist transitioning to FIDO2 due to the need for infrastructure changes. Hybrid approaches, supporting both certificate and FIDO2 authentication, could ease this transition.

Performance

Performance measurements indicate that the FIDO2 authentication process adds approximately 426 microseconds to the TLS handshake on the server side, adding a latency of around 20% to the process compared to no client authentication. Most of this overhead is attributed to cryptographic and database operations involved in verifying the FIDO2 response. While this increase is notable, it should be considered within the context of real-world deployments. Modern servers equipped with hardware acceleration for cryptographic operations could help mitigate this overhead.

Additionally, eliminating password-based authentication at the application layer may offset the increased handshake time by reducing the need for repeated authentication requests. On the client side, the most significant latency arises from user interaction, such as pressing a button on a YubiKey or providing a biometric scan. This interaction

is inherent to FIDO2's design and cannot be optimized without compromising security. Furthermore, password-based authentication will also usually require time to input the credentials.

Comparison with Previous Work

This implementation builds on the foundational work of Breitkopf, Freund, and Panizza, each of whom explored different aspects of FIDO2 integration with TLS.

Breitkopf's double-handshake approach, while not ideal for performance, was necessary to maintain compatibility with existing TLS state machines. Freund's proposal of using an external WebAuthn server introduced additional latency, which this thesis avoids by embedding the FIDO2 logic directly within the TLS library.

Panizza's work demonstrated the feasibility of integrating the authentication into a custom TLS extension for FIDO2, but the complexity of OpenSSL's codebase posed challenges. The Rustls implementation benefits from Rust's memory safety and modular design, making it easier to maintain and extend.

Limitations and Future Work

While this work advances the state of the art in TLS authentication, several limitations remain. The current implementation supports only resident keys, which limits its applicability in scenarios where storage space is limited. Extending support for non-resident keys would require addressing the challenge of securely transmitting user identifiers in the initial handshake. Due to the requirement for a double handshake, registration is more cumbersome to add to existing applications. Using an out-of-band approach could be preferable.

Performance measurements were conducted on a single machine with a limited number of connections. Evaluating the extension's behavior under high concurrency, such as thousands of simultaneous handshakes, would provide deeper insights into its scalability.

For broader adoption, the FIDO2 TLS extension would need to be standardized through bodies like the IETF. As of this thesis release, work is being done by Wedekind to create a draft RFC to push forward standardization efforts. When published, this work will be updated to conform to the proposed standard, unifying future efforts.

7. Conclusion

This thesis successfully integrates FIDO2-based client authentication into TLS 1.3 using the Rustls library, bringing FIDO2 support to a plethora of protocols. By leveraging hardware-backed cryptographic tokens, the work enhances security and usability for protocols like MQTT, which lack robust authentication mechanisms.

The implementation uses a double-handshake mechanism for registration to ensure encrypted transmission of sensitive data, while authentication is completed in a single handshake using resident keys. Compatibility with Panizza's FidoSSL ensures interoper-

ability, and integration with rumqtt demonstrates real-world applicability. Performance measurements show around 400-microsecond server overhead for FIDO2 authentication.

While the solution offers strong security benefits, adoption challenges persist, including implementation complexities, latency increases, and restrictions on resident keys. Future work could focus on support for non-resident keys and further scalability testing. This thesis lays the groundwork for broader adoption of passwordless, hardware-backed authentication in TLS, improving security across diverse digital communication protocols.

A. Connection Logs for Authentication

Rustls Server

```
[10:13:18 rustls_fido::server] FIDO2 server extension configured
[10:13:18 rustls_fido::server]        RP ID: localhost
[10:13:18 rustls_fido::server]        RP Name: localhost
[10:13:18 rustls_fido::server]        User Verification Policy: Discouraged
[10:13:18 rustls_fido::server]        Resident Key Policy: Required
[10:13:18 rustls_fido::server]        Authenticator Attachment Policy:
    CrossPlatform
[10:13:18 rustls_fido::server]        Token Timeout Policy: 60000
[10:13:18 rustls_fido::server]        Ticket: [4, 3, 2, 1]
[10:13:18 rustls_fido::server]        FIDO2 Authentication Mandatory: true
[10:13:18 rustls_fido::server]        Path to Database: ./fido.db3
Server listening on port 4443
[10:13:27 rustls::server::hs] decided upon suite TLS13_AES_256_GCM_SHA384
[10:13:27 rustls::server::tls13::client_hello] fido: Received Indication
[10:13:27 rustls::server::tls13::client_hello] fido: Starting Authentication
[10:13:27 rustls_fido::server] Creating Authentication Challenge
[10:13:27 rustls::server::tls13::client_hello] fido: Sending Request
[10:13:30 rustls::server::tls13] fido: Received Response
[10:13:30 rustls::server::tls13] fido: Finishing Authentication
[10:13:30 rustls_fido::server] Verifying Authentication Response
[10:13:30 rustls_fido::server] Response associated with User ID
    636f2aa5-3672-425a-8e02-b0ebecde1789
[10:13:30 rustls_fido::server] FIDO2 authentication successful
[10:13:30 rustls::server::tls13] client auth requested but invalid certificate
    supplied, still expecting certificate verify message
[10:13:30 rustls::common_state] Sending warning alert CloseNotify
```

FidoSSL Client

```
Configuring Client:
    Mode: Authentication
Sending packet: AUTH_INDICATION
Received packet: AUTH_REQUEST
    challenge: 3a120d8bc3aec0cd73876e8256ec8d17cec8d4d33e1be6fbbc66e63f17b747d1
    timeout: 60000000 ms
    rp id: localhost
Origin derived from SNI: https://localhost
Validated RPID: localhost
Generated client data: {
    "type":"webauthn.get",
    "challenge":"OhINi8OuwM1zh26CVuyNF87I1NM-G-b7vGbmPxe3R9E=",
    "origin":"https://localhost",
    "crossOrigin":false
}
Preparing CTAP with:
    Relying Party ID: localhost
    Client data hash:
        be0b1fb6ce310c58a6a1122c393fbfd98880fa32da196e48c745b385eca4a3f9
    User presence: TRUE
Discovered FIDO2 tokens: 1
Device path: /dev/hidraw8
    Manufacturer: Yubico
    Product: YubiKey OTP+FIDO+CCID
    Product ID: 1031
    Vendor ID: 4176
```

```
26 Running CTAP with device: /dev/hidraw8
27 Please touch the FIDO2 token
28 CTAP was successful
29 Sending packet: AUTH_RESPONSE
30     clientdata: {
31         "type":"webauthn.get",
32         "challenge":"OhINi8OuwM1zh26CVuyNF87I1NM-G-b7vGbmPxe3R9E=",
33         "origin":"https://localhost",
34         "crossOrigin":false
35     }
36     authenticator data:
37         49960de5880e8c687434170f6476605b8fe4a
38         eb9a28632c7995cf3ba831d97630500000035
39     signature:
40         3044022002606d17a9a78e0637871e9437fc4
41         eeacede0852d3c1d3aa16f61242cba7a36b02
42         2025e6bb2535e5f2ea5b8c94feeefb7a25a37
43         f2274fa45605eb00f9219b5c7d0d3
44     user id: 636f2aa53672425a8e02b0ebecde1789
45     cred id:
46         016b530ed7f6ab3971fd1915a8ca8c49df110
47         d74b7507ca9cbbf4537d49c26b9165fee22b5
48         d242059a54b0a36546296a
49 === TLS connection established.
50 Freeing all data allocated by FIDOSSL
```

B. Performance Data

Table 2: Performance Metrics (in microseconds) for a Rustls Handshake using the AES-256-GCM-SHA-384 cipher suite with X25519MLKEM768 for key agreement and FIDO2 client authentication

Operation	Mean	Stddev
Indication	0.197	0.070
Challenge	18.47	4.68
Response	407.6	54.1

C. Repositories

- Fork of the rustls TLS library [52]
- FIDO2 library for client authentication in Rustls [53]
- Fork of the tokio-rustls library [54]
- Fork of the rumqtt MQTT library [55]
- Rumqtt client with FIDO2 client authentication [56]

FIDO2 Messages

This section describes the FIDO2 extension messages. Some information is omitted for clarity. Exact details are in the source code. It serves as an updated version of the specification provided by Panizza [8].

Pre-Registration Indication

- **MESSAGE TYPE:**
  Value: 0x01

Pre-Registration Request

- **MESSAGE TYPE:**
  Value: 0x02

- **EPHEM USER ID:**
  Type: Byte String
  Length: 32 Bytes
  Description: Ephemeral identifier for the user, generated for pre-registration.

- **GCM KEY:**
  Type: Byte String
  Length: 32 Bytes
  Description: AES-GCM key to encrypt user-related values in further messages.

Pre-Registration Response

- **MESSAGE TYPE:**
  Value: 0x03

- **USER NAME:**
  Type: UTF-8 String
  Description: Username associated with the account.

- **USER DISPLAY NAME:**
  Type: UTF-8 String
  Description: A human-readable name for display purposes.

- **TICKET:**
  Type: Byte String
  Description: Encrypted registration ticket issued by the RP.

Registration Indication

- **MESSAGE TYPE:**
  Value: 0x04

- **EPHEM USER ID:**
  Type: Byte String
  Length: 256 Byte
  Description: Ephemeral user identifier linking first and second handshake.

Registration Request

- **MESSAGE TYPE:**
  Value: 0x05
- **CHALLENGE:**
  Type: Byte String
  Length: 32 Bytes
  Description: Random challenge from the RP.
- **RP ID:**
  Type: UTF-8 String
  Description: Relying Party identifier.
- **RP NAME:**
  Type: UTF-8 String
  Description: A human-readable name for the RP, mainly for display.
- **ENC USER NAME:**
  Type: Byte String
  Description: Encrypted username of the end-user.
- **ENC USER DISPLAY NAME:**
  Type: Byte String
  Description: Encrypted human-readable display name of the user.
- **ENC USER ID:**
  Type: Byte String
  Length: 16 Bytes
  Description: Encrypted unique identifier for the user.
- **PUBKEY CRED PARAMS:**
  Type: Array of Integers
  Length: 1–6 Bytes
  Description: List of possible public key algorithms.
  -7: COSE_ES256
  -35: COSE_ES384
  -8: COSE_EDDSA
  -25: COSE_ECDH_ES256
  -257: COSE_RS256
  -65535: COSE_RS1

- **OPTIONALS:**
  Type: CBOR map
  Description: Optional parameters.
  – TIMEOUT (Key 1): Integer, 4 Bytes. Maximum waiting time in ms.
  – AUTHENTICATOR SELECTION (Keys 2–4): Defines attachment, resident key policy, and user verification policy.
  – EXCLUDED CREDENTIALS (Key 5): List of credentials not allowed.

Registration Response

- **MESSAGE TYPE:**
  Value: 0x06

- **ATTESTATION OBJECT:**
  Type: Byte String
  Description: The attestation object as defined by the FIDO2 specification generated by the authenticator, containing attestation data and credential public key.

- **CLIENT DATA JSON:**
  Type: UTF-8 String
  Description: JSON-encoded client data, including challenge, origin, mode, and cross-origin.

Authentication Indication

- **MESSAGE TYPE:**
  Value: 0x07

Authentication Request

- **MESSAGE TYPE:**
  Value: 0x08

- **CHALLENGE:**
  Type: Byte String
  Length: 32 Bytes
  Description: Random challenge from the RP.

- **OPTIONALS:**
  Type: CBOR map
  Description: Optional authentication parameters.
  – TIMEOUT (Key 1): Integer, 4 Bytes. Maximum waiting time in ms.
  – RP ID (Key 2): UTF-8 String. RP identifier for this request. This is a required field.
  – USER VERIFICATION (Key 3): Policy defining whether user verification is required(1)/preferred(2)/discouraged(3).

  - ALLOW CREDENTIALS (Key 4): Array of allowed credentials. Restricts authentication to the listed credentials.
  - EXTENSIONS (Key 5): Array of extensions to augment authentication.

Authentication Response

- **MESSAGE TYPE:**
  Value: 0x09

- **CLIENT DATA JSON:**
  Type: UTF-8 String
  Description: JSON-encoded client data, including challenge, origin, mode, and cross-origin.

- **AUTHENTICATOR DATA:**
  Type: Byte String
  Description: Authenticator data as specified by FIDO2 containing flags, RP ID hash, and signature counter.

- **SIGNATURE:**
  Type: Byte String
  Description: Signature produced over client data and authenticator data.

- **USER HANDLE:**
  Type: Byte String
  Description: Identifier of the authenticated user, provided if available.

- **SELECTED CREDENTIAL ID:**
  Type: Byte String
  Description: The credential ID selected by the authenticator.

Bibliography

References

1. Pocong, A. *SSL/TLS Certificate Statistics and Trends for 2025* https://www.networksolutions.com/blog/ssl-tsl-trends/ (Sept. 30, 2025).
2. Kozierok, C. *The TCP/IP Guide: A Comprehensive, Illustrated Internet Protocols Reference* ISBN: 9781593270476. https://books.google.de/books?id=Pm4RgYV2w4YC (No Starch Press, 2005).
3. Cloudflare. *What is TLS (Transport Layer Security)?* Cloudflare. https://www.cloudflare.com/learning/ssl/transport-layer-security-tls/ (Sept. 30, 2025).
4. Rescorla, E. *The Transport Layer Security (TLS) Protocol Version 1.3* Request for Comments 8446. RFC 8446 (Internet Engineering Task Force, Aug. 2018). https://datatracker.ietf.org/doc/html/rfc8446.
5. FIDO Alliance. *Submission Request to W3C: FIDO 2.0 Platform Specifications 1.0* World Wide Web Consortium (W3C). https://www.w3.org/submissions/2015/02/ (July 11, 2025).
6. Breitkopf, T.-L. J. *FIDO2 als TLS-1.3-Erweiterung* MA thesis (Humboldt Unviersität zu Berlin, 2020). https://sar.informatik.hu-berlin.de/research/publications/SAR-PR-2020-04/SAR-PR-2020-04_.pdf.
7. Freund, M. *FIDO2-Erweiterung von TLS 1.3 in einer gängigen Kryptobibliothek* MA thesis (Humboldt Unviersität zu Berlin, 2021). https://sar.informatik.hu-berlin.de/research/publications/SAR-PR-2021-02/SAR-PR-2021-02_.pdf.
8. Panizza, J. *FIDO2 TLS 1.3 Extension: Strong EAP-TLS Authentication for 802.1X Networks* MA thesis (Humboldt Unviersität zu Berlin, 2024). https://sar.informatik.hu-berlin.de/research/publications/SAR-PR-2024-02/SAR-PR-2024-02_.pdf.
9. Geise, G. *FIDO2-TLS-1.3-Erweiterung in JSSE* MA thesis (Humboldt Unviersität zu Berlin, 2025). https://sar.informatik.hu-berlin.de/research/publications/SAR-PR-2025-01/SAR-PR-2025-01_.pdf.
10. Oppliger, R. *SSL and TLS: Theory and Practice* (Artech House, 2023).
11. Joseph A. Salowey Sean Turner, C. A. W. *TLS 1.3: One Year Later* IETF. https://www.ietf.org/blog/tls13-adoption/ (July 10, 2025), archived at https://web.archive.org/web/20250420000336/https://www.ietf.org/blog/tls13-adoption/ on Apr. 20, 2025.
12. wolfSSL. *TLS 1.3 Protocol Support* wolfSSL Inc. https://www.wolfssl.com/docs/tls13/ (Sept. 25, 2025), archived at https://web.archive.org/web/20250917103621/https://www.wolfssl.com/docs/tls13/ on Sept. 17, 2025.

13. Roy, S. *Pre-master Secret vs. Master Secret vs. Private Key vs. Shared Secret* Baeldung. https://www.baeldung.com/cs/pre-master-shared-secret-private-public-key (Sept. 25, 2025), archived at https://web.archive.org/web/20250429183628/https://www.baeldung.com/cs/pre-master-shared-secret-private-public-key on Apr. 29, 2025.
14. Et al., S. F. *Transport Layer Security (TLS) Application-Layer Protocol Negotiation Extension* Request for Comments 7301. RFC 7301 (Internet Engineering Task Force, July 2014). https://www.rfc-editor.org/rfc/rfc7301.
15. Ghorbani Lyastani, S., Schilling, M., Neumayr, M., Backes, M. & Bugiel, S. *Is FIDO2 the Kingslayer of User Authentication? A Comparative Usability Study of FIDO2 Passwordless Authentication* in *2020 IEEE Symposium on Security and Privacy (SP)* (2020), 268–285.
16. Cao, I. *Why Big Tech Is Striving for the World without Password* EnvZone. https://envzone.com/why-big-tech-is-striving-for-the-world-without-password/ (July 11, 2025).
17. W3C Web Authentication Working Group. *Web Authentication: An API for accessing Public Key Credentials Level 2* https://www.w3.org/TR/webauthn/ (July 11, 2025).
18. FIDO Alliance. *Client to Authenticator Protocol (CTAP) - Protocol Specification v2.2 Public Draft* https://fidoalliance.org/specs/fido-v2.2-ps-20250228/fido-client-to-authenticator-protocol-v2.2-ps-20250228.pdf (July 11, 2025).
19. Google. *Use a security key for 2-Step Verification* Google. https://support.google.com/accounts/answer/6103523 (Sept. 30, 2025).
20. Thompson, C. *How Rust went from a side project to the world's most-loved programming language* https://www.technologyreview.com/2023/02/14/1067869/rust-worlds-fastest-growing-programming-language/ (July 3, 2025), archived at https://web.archive.org/web/20250626110024/https://www.technologyreview.com/2023/02/14/1067869/rust-worlds-fastest-growing-programming-language/ on June 26, 2025.
21. Matsakis, N. D. & Klock, F. S. The rust language. *Ada Lett.* **34,** 103–104. ISSN: 1094-3641. https://doi.org/10.1145/2692956.2663188 (Oct. 2014).
22. The Computer Language Benchmarks Game. *pidigits* https://benchmarksgame-team.pages.debian.net/benchmarksgame/performance/pidigits-cpu.html (July 3, 2025), archived at https://web.archive.org/web/20250218053012/https://benchmarksgame-team.pages.debian.net/benchmarksgame/performance/pidigits-cpu.html on Feb. 18, 2025.
23. Klabnik, S. & Nichols, C. *The Rust programming language* (No Starch Press, 2023).
24. Jung, R., Jourdan, J.-H., Krebbers, R. & Dreyer, D. RustBelt: securing the foundations of the Rust programming language. *Proc. ACM Program. Lang.* **2.** https://doi.org/10.1145/3158154 (Dec. 2017).

25. Ben-Yehuda, A. *std::thread::JoinGuard (and scoped) are unsound because of reference cycles* https://github.com/rust-lang/rust/issues/24292 (July 3, 2025).
26. Overflow, S. *Developer Survey 2024* https://survey.stackoverflow.co/2024/technology#2-programming-scripting-and-markup-languages (July 3, 2025).
27. TIOBE. *TIOBE Index - Rust* https://www.tiobe.com/tiobe-index/rust/ (July 3, 2025).
28. Cimpanu, C. *Microsoft: 70 percent of all security bugs are memory safety issues* https://www.zdnet.com/article/microsoft-70-percent-of-all-security-bugs-are-memory-safety-issues/ (July 4, 2025), archived at https://web.archive.org/web/20250609063542/https://www.zdnet.com/article/microsoft-70-percent-of-all-security-bugs-are-memory-safety-issues/ on June 9, 2025.
29. Vaughan-Nichols, S. *Rust in Linux: Where we are and where we're going next* https://www.zdnet.com/article/rust-in-linux-where-we-are-and-where-were-going-next/ (July 4, 2025), archived at https://web.archive.org/web/20250114162752/https://www.zdnet.com/article/rust-in-linux-where-we-are-and-where-were-going-next/ on Jan. 14, 2025.
30. Google Security Team. *Integrating Rust into the Android Open Source Project* Google Security Blog. https://security.googleblog.com/2021/05/integrating-rust-into-android-open.html (July 4, 2025), archived at https://web.archive.org/web/20250514044122/https://security.googleblog.com/2021/05/integrating-rust-into-android-open.html on May 14, 2025.
31. Register, T. *Microsoft is busy rewriting core Windows code in memory-safe Rust* The Register. https://www.theregister.com/2023/04/27/microsoft_windows_rust/ (July 4, 2025), archived at https://web.archive.org/web/20250511225617/https://www.theregister.com/2023/04/27/microsoft_windows_rust/ on May 11, 2025.
34. rustls. *Documentation for rustls* (). https://docs.rs/rustls/latest/rustls/ (July 4, 2025).
37. Project, R. *Implementation Vulnerabilities* Rustls. https://docs.rs/rustls/latest/rustls/manual/_01_impl_vulnerabilities/index.html (July 4, 2025).
38. CVE Program. *CVE-2014-0160: OpenSSL Heartbleed Vulnerability* Official CVE record for the Heartbleed bug in OpenSSL. MITRE. https://www.cve.org/CVERecord?id=CVE-2014-0160 (July 9, 2025).
39. Beurdouche, B. *et al.* A messy state of the union: Taming the composite state machines of TLS. *Communications of the ACM* **60,** 99–107 (2017).
40. Cloudflare. *Why use TLS 1.3? | SSL and TLS vulnerabilities* Cloudflare. https://www.cloudflare.com/learning/ssl/why-use-tls-1.3/ (July 9, 2025).
41. Rustls Project. *TLS Vulnerabilities* Rustls. https://docs.rs/rustls/latest/rustls/manual/_02_tls_vulnerabilities/index.html (July 4, 2025).

42. Rustls Project. *Rustls features* Rustls. https://docs.rs/rustls/latest/rustls/manual/_04_features/index.html (July 4, 2025).

43. Rustls Project. *Rustls performance* Rustls. https://rustls.dev/perf/2024-10-18-report/ (July 4, 2025).

44. Aas, J. *Rustls Gains OpenSSL and Nginx Compatibility* Announcement of Rustls' OpenSSL compatibility layer for Nginx, enabling memory-safe TLS adoption. Memory Safety (Prossimo Initiative, ISRG). https://www.memorysafety.org/blog/rustls-nginx-compatibility-layer/ (July 9, 2025).

45. McArthur, S. & contributors. *reqwest: Add Rustls as default TLS backend* Pull request switching reqwest's default TLS backend from OpenSSL to Rustls. GitHub. https://github.com/seanmonstar/reqwest/pull/2752 (July 4, 2025).

46. *Overview of OpenVPN — OpenVPN Community Wiki* OpenVPN. https://community.openvpn.net/openvpn/wiki/OverviewOfOpenvpn, archived at https://web.archive.org/web/20220707113857/https://community.openvpn.net/openvpn/wiki/OverviewOfOpenvpn on July 7, 2022.

47. Farke, F. M., Lorenz, L., Schnitzler, T., Markert, P. & Dürmuth, M. *"You still use the password after all" – Exploring FIDO2 Security Keys in a Small Company* in *Sixteenth Symposium on Usable Privacy and Security (SOUPS 2020)* (USENIX Association, Aug. 2020), 19–35. ISBN: 978-1-939133-16-8. https://www.usenix.org/conference/soups2020/presentation/farke.

48. Cloudflare. *Good-bye ESNI, hello ECH!* https://blog.cloudflare.com/encrypted-client-hello/ (Sept. 17, 2025).

Software

32. [SW] rustls, *rustls* URL: https://github.com/rustls/rustls(July 4, 2025).

33. [SW Rel.] rustls, *rustls* version release/0.1, 2016. URL: https://crates.io/crates/rustls/0.1.0(July 4, 2025).

35. [SW Rel.] rustls, *rustls* version release/0.23.28, 2025. URL: https://github.com/rustls/rustls/tree/v/0.23.28(July 4, 2025).

36. [SW] Services, A. W., *aws-lc-rs* URL: https://github.com/aws/aws-lc-rs (July 9, 2025).

49. [SW] Line Corporation, *Line FIDO2 Server* URL: https://github.com/line/line-fido2-server(July 4, 2025).

50. [SW] Ehlert, E., *fidoSSL Pull Request: Adjust code to conform more with WebAuthn specs* 2025. URL: https://github.com/tummetott/fidoSSL/pull/1(June 19, 2025).

51. [SW] elleh, *softfido* URL: https://github.com/ellerh/softfido(Aug. 1, 2025).

52. [SW Rel.] Ehlert, E., *Fork of rustls* version commit/56ab5f2, 2025. URL: https://github.com/7ritn/rustls/commit/56ab5f296461e99326f2ae43cf7f295a353316aa(Oct. 20, 2025).
53. [SW Rel.] Ehlert, E., *rustls-fido* version commit/722aeda, 2025. URL: https://github.com/7ritn/rustls-fido/commit/722aeda27e1f18f4a1af944aa80bdbc882874fbf(July 4, 2025).
54. [SW Rel.] Ehlert, E., *Fork of tokio-rustls* version commit/f4597d5, 2025. URL: https://github.com/7ritn/tokio-rustls/commit/f4597d569b99c26ce401b99f34938f056fd74059(Sept. 17, 2025).
55. [SW Rel.] Ehlert, E., *Fork of rumqtt* version commit/4197382, 2025. URL: https://github.com/7ritn/rumqtt/commit/41973829bd4462cb0bff7f2b17422f0f2d237c46(Sept. 17, 2025).
56. [SW Rel.] Ehlert, E., *Client for rumqtt with FIDO2* version commit/bcacbc6, 2025. URL: https://github.com/7ritn/rumqtt-fido-client/commit/bcacbc6db0db4282d994f44b15a338c99d7bea3c(Oct. 20, 2025).

Selbständigkeitserklärung

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und noch nicht für andere Prüfungen eingereicht habe. Sämtliche Quellen einschließlich Internetquellen, die unverändert oder abgewandelt wiedergegeben werden, insbesondere Quellen für Texte, Grafiken, Tabellen und Bilder, sind als solche kenntlich gemacht. Mir ist bekannt, dass bei Verstößen gegen diese Grundsätze ein Verfahren wegen Täuschungsversuchs bzw. Täuschung eingeleitet wird.

Berlin, den October 22, 2025