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

FIDO2 TLS 1.3 Extension: Strong EAP-TLS Authentication for 802.1X Networks

Masterarbeit

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

eingereicht von:	Jonas Panizza
geboren am:	[redacted]
geboren in:	[redacted]
Gutachter/innen:	Prof. Dr. Jens Peter Redlich Prof. Dr. Joel Rybicki
eingereicht am:	verteidigt am:

Abstract

FIDO authenticators offer a promising method for client authentication in web applications, gaining popularity due to their strong security, phishing resistance, privacy protection, and high usability. These authenticators are hardly used in other protocols that require client authentication, as they were originally designed for web environments. This thesis proposes a TLS extension designed to integrate FIDO authentication and key registration directly into TLS at the transport layer, making it available to non-HTTP applications. The extension is implemented for `OpenSSL` and deployed as a C library. In order to evaluate and demonstrate it's practicality, the extension is integrated into `hostapd` and `wpa_supplicant` to establish an 802.1X EAP-TLS Wi-Fi network that uses FIDO hardware authenticators, replacing traditional X.509 client certificates.

Contents

Acronyms

2FA	Two-Factor Authentication
AES	Advanced Encryption Standard
AP	Access Point
API	Application Programming Interface
ASN.1	Abstract Syntax Notation One
CA	Certificate Authority
CBC	Cipher Block Chaining
CBOR	Concise Binary Object Representation
CCA	Client Certificate Authentication
CCMP	Counter Mode Cipher Block Chaining Message Authentication Code Protocol
CN	Common Name
CTAP	Client to Authenticator Protocol
DER	Distinguished Encoding Rules
DNS	Domain Name System
EAP	Extensible Authentication Protocol
EAPOL	EAP over LAN
ECDSA	Elliptic Curve Digital Signature Algorithm
EdDSA	Edwards-curve Digital Signature Algorithm
FIDO	Fast Identity Online
FTP	File Transfer Protocol
GCM	Galois/Counter Mode
GCMP	Galois/Counter Mode Protocol

HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
IANA	Internet Assigned Numbers Authority
IoT	Internet of Things
IV	Initialization Vector
JSON	JavaScript Object Notation
L2TP	Layer 2 Tunneling Protocol
LAN	Local Area Network
LDAP	Lightweight Directory Access Protocol
MAC	Message Authentication Code
MITM	Man-in-the-Middle
MQTT	Message Queuing Telemetry Transport
NIST	National Institute of Standards and Technology
NNTP	Network News Transfer Protocol
OSI	Open Systems Interconnection
OTP	One-Time Password
PKCS	Public-Key Cryptography Standards
PKI	Public Key Infrastructure
PMK	Pairwise Master Key
PoC	Proof of Concept
PSK	Pre-Shared Key
RADIUS	Remote Authentication Dial-In User Service
REST	Representational State Transfer
RFC	Request for Comments
RP	Relying Party

RSA	Rivest Shamir Adleman
RTT	Round-Trip Time
SAN	Subject Alternative Name
SHA	Secure Hash Algorithm
SMTP	Simple Mail Transfer Protocol
SNI	Server Name Indication
SRP	Secure Remote Password
SSH	Secure Shell
SSL	Secure Sockets Layer
TCP	Transmission Control Protocol
TLS	Transport Layer Security
TOFU	Trust On First Use
TPM	Trusted Platform Module
U2F	Universal 2nd Factor
VPN	Virtual Private Network
W3C	World Wide Web Consortium
WPA	Wi-Fi Protected Access

1. Introduction

1.1. Motivation

In the field of IT security and computer networks, securing client-server communications has remained one of the most important objectives. The Transport Layer Security (TLS) protocol plays a central role in securing these connections by ensuring confidentiality, integrity and authenticity. While server authentication is a well-established norm through the use of X.509 certificates, client authentication, particularly in outside of web-based environments, has not seen a parallel level of emphasis or development. This discrepancy leaves a gap in the security model, especially as the digital ecosystem evolves to include a large number of client devices, from mobile and desktop environments to the growing Internet of Things (IoT) landscape.

Traditionally, TLS relies on a range of authentication mechanisms, with client certificates being a widely recognized and robust method for client authentication. However, this approach, despite its strengths, is not without its challenges. Client certificates can be cumbersome to manage and deploy [19], especially if they are stored on secure hardware like external tokens or smart cards. The requirement for a Certificate Authority (CA) as part of the broader Public Key Infrastructure (PKI) to issue new certificates makes it an administrative challenge. This challenge is particularly evident in large-scale and dynamic networks like *eduroam*, a global service providing worldwide roaming access for the research and education community.

In the web environment, a promising new method of client authentication known as Fast Identity Online (FIDO) has been available for some years. This framework aims to replace conventional password authentication with phishing-resistant public key cryptography. FIDO uses dedicated authenticators, mostly external hardware tokens, that securely store the private key component or key material to derive the private key of a FIDO credential. Compared to traditional client certificates, FIDO offers significantly improved usability because a single authenticator can store key material for multiple platforms, providing the user with a single token that grants access to all their services. In addition, FIDO does not rely on a PKI, thereby eliminating associated administrative challenges. Instead of having a public key signed by a CA and shared between parties, it adheres to a one-key-per-service policy. Each public key is registered directly with the service during enrollment. This approach significantly enhances user privacy compared to traditional client certificates because public keys are unique to each service, preventing user tracking.

Despite its potential, FIDO's adoption has been largely confined to the web, limiting its use in other protocols that rely on client authentication. Recognizing this limitation, this thesis proposes a TLS 1.3 extension, that aims to bridge this gap by integrating FIDO directly into the TLS handshake process, thus decoupling it from its traditional Hypertext Transfer Protocol (HTTP)-based constraints. Since TLS cryptographically validates the server identity for each connection, it has the capability to effectively enforce FIDO's one-key-per-service policy.

By embedding FIDO directly into TLS, the extension enables strong client authentication across the entire spectrum of applications build atop TLS. This advancement holds particular promise for 802.1X TLS-Extensible Authentication Protocol (EAP) networks, offering them the opportunity to shift from the traditional Client Certificate Authentication (CCA) to the secure, user-friendly and privacy-preserving FIDO authentication method. This thesis provides a Proof of Concept (PoC) implementation of the aforementioned extension and integrates it into the EAP-TLS protocol, demonstrating that Wi-Fi networks such as eduroam can benefit from the new authentication method.

1.2. Structure of the Thesis

The structure of this thesis is outlined as follows: Section 2 offers an overview of all protocols involved in the proposed extension, providing the necessary context for the integration. It starts by explaining the TLS 1.3 handshake and the TLS 1.3 extension mechanism. This is followed by a step-by-step description of the FIDO registration and authentication ceremonies. Subsequently, an overview of the EAP-TLS handshake is given to understand the role of TLS in 802.1X authentication. Section 3 presents four examples of related work, where FIDO authentication is already used in non-web environments. Section 4 describes the methodology of the thesis, starting by defining design principles, requirements, and a network and threat model. It then explains how the FIDO message exchange could be integrated into the TLS handshake, it details how messages could be structured, encoded, and how key registration can be controlled by the server. This section also includes various integration details that must be considered when adapting FIDO to a non-web environment. The Section concludes with how the outcomes of FIDO ceremonies are communicated to the other peer using TLS alerts. Section 5 presents two PoC implementations: First, the proposed TLS extension as a C library which can be used in conjunction with `OpenSSL`, secondly the use of the extension in EAP-TLS Wi-Fi networks. Section 6 evaluates the implementations, conducts worst and average case estimations of message sizes, and assesses the practicality of the extension. The Section also discusses possible improvements and future work on this topic. Section 7 concludes this thesis by summarizing the most important findings.

The thesis also includes an extensive set of appendices. The first appendix defines the fields, structure, and encoding of the newly defined messages of the protocol. The second appendix provides a guide on how to create X.509 certificates for a test environment of TLS applications. The third appendix includes the necessary configuration files for `hostapd` and `wpa_supplicant` needed to set up an EAP-TLS Wi-Fi network that uses FIDO authentication. Finally, the last appendix contains a helper script that facilitates the development of client-server applications.

2. Theoretical Framework

This thesis builds upon the foundational protocols of TLS, FIDO and EAP-TLS, each of which serves as a prerequisite for comprehending the proposed extension. The subsequent chapter will explore the fundamentals of these protocols, laying the groundwork for the following methodology.

2.1. TLS 1.3

TLS is a crucial cryptographic protocol that ensures secure communication over computer networks, most commonly the internet. Initially developed as Secure Sockets Layer (SSL) by Netscape in the 1990s [26], TLS has become a robust standard for securing data transmissions between applications and servers. It operates at the transport layer of the Open Systems Interconnection (OSI) model, ensuring privacy, integrity, and authentication in communications.

TLS has undergone several iterations since its inception, each version refining and enhancing security, performance, and functionality. Following its predecessor, TLS 1.0 was introduced as an upgrade to SSL 3.0 to address inherent security vulnerabilities [26, p. 61]. Subsequent versions, TLS 1.1 and TLS 1.2, introduced more cryptographic algorithms and improved protections against attacks such as Cipher Block Chaining (CBC) padding attacks and Browser Exploit Against SSL/TLS (BEAST) [16, p. 30-32].

A significant step was made with TLS 1.3, which not only further enhanced security by removing outdated cryptographic algorithms and reducing the risk of misconfigurations but also improved connection latency through a more concise handshake process [16, p. 27]. One of the notable advancements in TLS 1.3 is its reduction in Round-Trip Times (RTTs) during the handshake phase. Whereas TLS 1.2 and earlier versions require two full round trips between client and server to complete the handshake, TLS 1.3 has effectively reduced this to just one round trip, significantly speeding up the initial connection setup. This enhancement is crucial for performance, particularly in applications where speed and low latency are important, such as in cellular networks. Furthermore, TLS 1.3 introduces ”0-RTT” resumption, allowing clients to send encrypted data to the server in the same round trip as the initial handshake, under certain conditions. This feature can further reduce latency for subsequent connections to the same server, at the cost of some security trade-offs [7].

For the purposes of this thesis, the focus will be exclusively on TLS 1.3. This choice is motivated by TLS 1.3’s advancements in security and efficiency, which make it the most relevant version for addressing contemporary cybersecurity challenges and ensuring the secure transmission of data in modern computer networks. The following subsections outline the TLS 1.3 handshake and custom extension mechanism. Both parts are essential for the following methodology.

2.1.1. TLS 1.3 Handshake

Figure 1 illustrates the protocol messages exchanged during a TLS 1.3 handshake between client and server. The explanations of the entire handshake process are based on Request for Comments (RFC) 8446 [21].

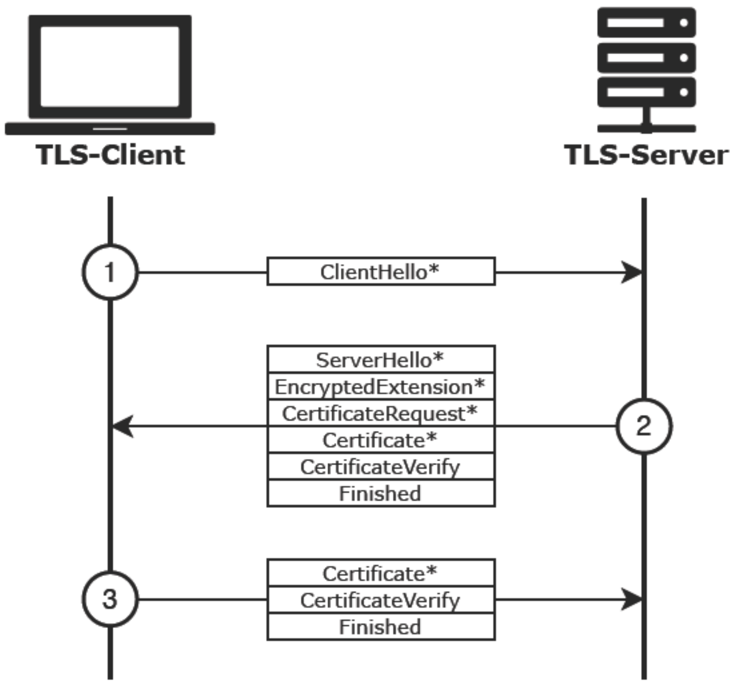

Figure 1: TLS 1.3 Handshake with Client Certificates. Graphic created by author based on specifications from RFC 8446 [21].

1. The process begins with the client sending a *ClientHello* message to the server, which outlines the cypher suites supported by the client, includes a session ID for potential session resumption, and offers key shares for the anticipated key agreement protocol. Anticipating the cypher suite allows the handshake to potentially save an entire round trip compared to TLS 1.2.

2. The server responds with several messages, beginning with a *ServerHello* to agree on the cryptographic suite and including its own key share. From this point forward, all subsequent messages are encrypted. This encryption is made possible by the completion of a shared secret negotiation between the client and the server within the *ServerHello* message. The shared secret is derived through a key exchange mechanism, often based on Diffie-Hellman or its elliptic curve variant, which is part of the cryptographic suites agreed upon during the *ClientHello* and *ServerHello* messages. This mechanism utilizes the key shares from both the client and the

server to securely establish a shared secret without it ever being transmitted over the network.

Upon establishing the shared secret for encryption, the server proceeds to dispatch *EncryptedExtensions*, which may carry additional extensions necessary for the session. Next, if the server requires client authentication, it sends a *CertificateRequest* indicating the types of certificates supported and the allowed CAs, as part of its response. This is followed by the server's *Certificate* message, where it presents its own X.509 certificate to the client. The server also sends a *CertificateVerify* message to prove possession of the private key corresponding to the public key in its certificate.

The server concludes this phase with a *Finished* message, which contains a Message Authentication Code (MAC) computed over all previous handshake messages. This MAC, derived from the shared secret, assures the client of the integrity and authenticity of the handshake process, confirming that the messages have not been tampered with. It signifies that the server's part of the handshake is complete and establishes a secure, encrypted channel for exchanging application data. The verification of this MAC by the client ensures that both parties have a consistent view of the handshake and are in possession of the same shared secret, marking the transition to secure communication.

3. If the server requests client authentication, the client responds with a series of messages to complete the mutual authentication process. Initially, the client sends a *Certificate* message, provided it possesses a suitable certificate that it is willing to share. This certificate should be issued by a CA that is part of a trust chain ending with a CA trusted by the server, as specified in the server's *CertificateRequest* message. Following this, the client sends a *CertificateVerify* message, which uses a digital signature to prove that the client holds the private key corresponding to the public key in the certificate it presented.

   The client completes its response with a *Finished* message, analogous to the server's earlier *Finished* message, but computed from the client's perspective. This message includes a MAC that covers all handshake messages exchanged from the *ClientHello* up to and including the server's *Finished* message. This MAC is derived from keys that are generated from the shared secret established during the handshake. By successfully verifying this MAC, the server can confirm the integrity and authenticity of the handshake messages exchanged with the client. This verification ensures that the handshake has not been tampered with and that both parties have agreed upon the same parameters and shared secrets. The *Finished* message from the client effectively signals the completion of the mutual authentication process and confirms the readiness of both parties to secure application data exchange over the encrypted channel. TLS 1.3 enables the client to send encrypted application data immediately along with the *Finished* message. This capability underpins the classification as a 1-RTT handshake, despite the handshake technically extending beyond a single request-response cycle.

The PKI plays a crucial role in this handshake, as it underpins the trustworthiness of the certificates exchanged. Certificates issued by recognized CAs within the PKI framework help ensure that the entities involved in the communication are indeed who they claim to be. This handshake mechanism thus establishes a secure channel protected by encryption and authenticated by PKI, where both parties can be assured of the other's identity (if client authentication is used), and that the data exchanged thereafter maintains confidentiality and authenticity.

2.1.2. TLS 1.3 Custom Extensions

TLS 1.3 introduces predefined extensions such as the "Pre-Shared Key Extension" and the "Early Data Extension", designed to fulfill a variety of cryptographic needs and enhancements, as documented in RFC 8446 [21, p. 52-55]. Beyond these predefined options, TLS 1.3 also supports the integration of custom extensions. These extensions enable developers to tailor the protocol to meet unique requirements or expand its capabilities beyond the core specifications. Custom extensions adhere to the structural and encoding guidelines established by TLS 1.3, yet they are distinguished by unique extension type codes not allocated by the Internet Assigned Numbers Authority (IANA). This design ensures that custom extensions can coexist with standard extensions without causing conflicts or interoperability issues.

Custom extension data can be integrated into several handshake messages. Specifically, the *ClientHello* and *ServerHello* messages can incorporate custom extensions, facilitating the preliminary exchange of additional functionalities or demands between clients and servers. It is important to note, however, that since these initial messages are sent unencrypted, any embedded extension data also remains unencrypted. As the handshake process progresses and encryption is established, subsequent messages like *EncryptedExtensions*, *CertificateRequest*, and *Certificate* can carry custom extensions. This feature permits the secure transmission of sensitive or application-specific information. Nonetheless, not all handshake messages in TLS 1.3 support the inclusion of custom extension data. For example, the *CertificateVerify* and *Finished* messages are not allowed to carry custom extensions. The "*" notation in Figure 1 indicates that a handshake message can be supplemented with extension data.

While the primary aim of extensions is to augment the message they accompany logically, it is also feasible to transmit arbitrary data within their payload. For instance, a custom extension in the *Certificate* message might include data unrelated to the certificate itself.

2.2. FIDO2

The *FIDO Alliance* is an open industry association that brings together companies such as PayPal, Google, Microsoft, Meta, Lenovo, and Yubico to create a new client authentication method based on asymmetric cryptography that does not rely on passwords. FIDO authentication uses authenticators, which can be either external hardware tokens—like USB security keys, NFC, or Bluetooth variants—or integrated into the platform running the client software, known as platform authenticators. These authenticators store cryptographic key material that is never shared with any party. Possession of the authenticator serves as proof of identity. In addition, factors such as PINs, passwords, and biometrics can be used locally to unlock the authenticator.

The original FIDO protocol, FIDO 1.0, was the first iteration of the FIDO standard. It included the Universal 2nd Factor (U2F) protocol and the less widely adopted Universal Authentication Framework (UAF). The part of U2F dealing with communication between the authenticator and the client was later named Client to Authenticator Protocol (CTAP)1 [34]. In 2016, the FIDO Alliance collaborated with the World Wide Web Consortium (W3C) to standardize FIDO for the web, resulting in the WebAuthn L1 authentication API. At the time of this writing, FIDO2 is the latest iteration of the FIDO standard, including the newer versions CTAP 2.0 and WebAuthn L2.

Throughout this thesis, "FIDO" will often be used as an umbrella term; however, strictly speaking, it specifically refers to the latest FIDO specification, FIDO2, which builds on CTAP 2.0 and WebAuthn L2. Before providing a detailed exploration of the FIDO internals, we introduce the key terminology essential for understanding its operation and implementation:

- **Relying Party (RP)**: The service requiring user authentication, typically a website or application.
- **Client**: The user's device, such as a smartphone or computer, requesting authentication.
- **Authenticator**: The device, such as a hardware token, mobile device, or platform-integrated authenticator, employed to verify a user's identity with the RP.
- **Ceremony**: A series of steps performed during authentication or registration, involving interaction between the client, relying party, and authenticator.
- **Credential**: One keypair registered with a specific RP, which can be either a *Discoverable Credential* (or Resident Key), stored directly on the authenticator and enabling user authentication without specific credential ID, or a *Non-Discoverable Credential* (or Non-Resident Key), which is stored by the RP in an encrypted form and requires a unique credential ID for identification.
- **Assertion**: Proof of possession of a credential, generated by the authenticator during authentication.

- **Attestation**: A process whereby an authenticator provides cryptographic proof of its integrity and identity to the RP during registration.
- **User Presence**: A simple gesture, like touching a button, indicating the user is physically present.
- **User Verification**: A stronger form of verification, such as a PIN, fingerprint, or facial recognition, confirming the user's identity.

At its core, FIDO consists of two components: WebAuthn and CTAP. WebAuthn, short for Web Authentication, is a web standard that defines a JavaScript Application Programming Interface (API), implemented by modern web browsers such as Firefox, Chrome, and Safari. JavaScript applications running in a client's browser can use this API to interact with authenticators connected to the user's platform. CTAP is the protocol used for communications between the client platform and the FIDO authenticator, regardless of the type or form factor. CTAP revolves around two fundamental operations: `authenticatorMakeCredential` for registering new credentials and `authenticatorGetAssertion` for authentication with existing credentials. Within FIDO terminologies, these are referred to as ceremonies rather than protocols. This distinction is made due to the direct participation of the human user, often through user presence checks. In essence, ceremonies highlight the active involvement of users in authentication processes, reinforcing security through human verification steps.

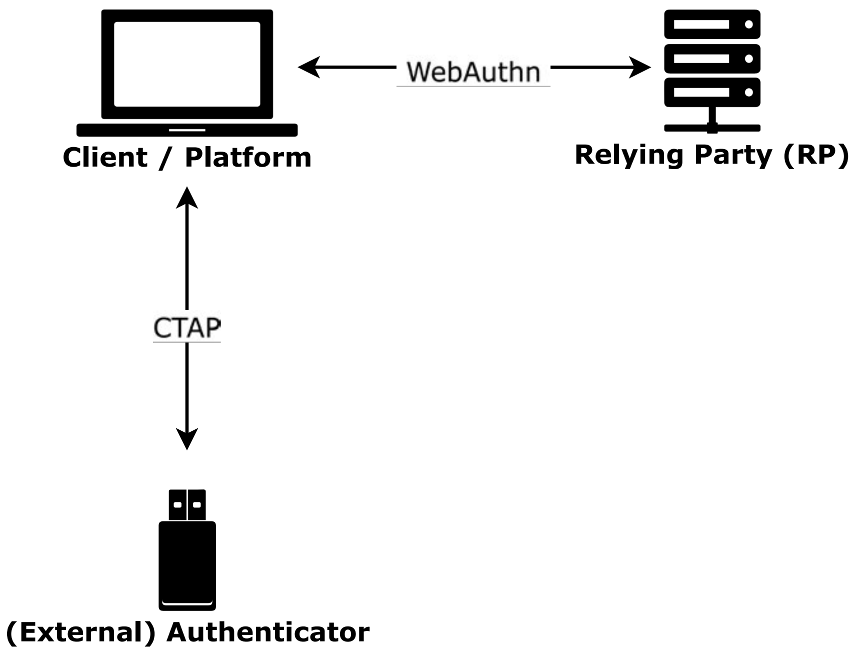

Figure 2: FIDO2 Architecture. Graphic created by author based on [4].

Figure 2 illustrates the architecture of FIDO, which comprises both WebAuthn and CTAP. To provide an overview, let's consider a typical FIDO ceremony at a high level, without going into the details. The sequence is based on the WebAuthn L2 specification [31]:

1. The client requests a web page with Hypertext Transfer Protocol Secure (HTTPS).
2. The RPs web application sends JavaScript code to the client's browser as part of the web page.
3. This JavaScript code, which represents the RPs logic and functionality, calls the WebAuthn API to perform authentication or registration operations.
4. The WebAuthn API, provided by the browser, ensures the request's origin matches the RP ID for security. Once verified, the API uses the CTAP to establish communication with the authenticator connected to the client's platform, sending a specific request to perform the desired operation.
5. Upon receiving commands from the WebAuthn API, the authenticator performs the necessary operations based on the requested action. For example, during authentication, the authenticator may prompt the user to perform a biometric verification or provide evidence of user presence.
6. The authenticator generates a response, like an assertion, depending on the specific operation requested by the WebAuthn API.
7. The authenticator sends the response back to the client device through the CTAP.
8. The WebAuthn API receives the response from the authenticator via the CTAP and the JavaScript application running in the client's browser forwards it to the RP via HTTP.
9. The RP's web application verifies the response, depending on the context of the operation.
10. If the verification is successful, the RPs web application grants access to the requested service or completes the registration process, allowing the user to proceed with the intended action on the web page.

These steps provide a broad overview of the control flow, omitting specific details regarding the contents of request and response packets. However, to integrate FIDO into the TLS handshake, it is crucial to comprehend the creation, verification, and contents of these packets. The subsequent sections will elaborate on the key aspects of registration and authentication in detail.

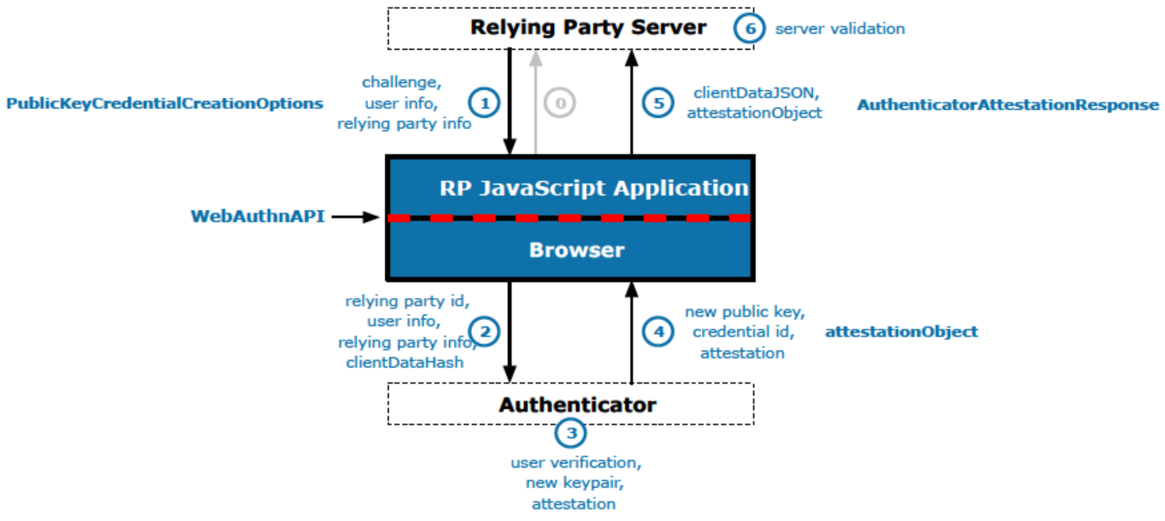
Relying Party Server
6 server validation
PublicKeyCredentialCreationOptions
challenge, user info, relying party info
1
0
5
clientDataJSON, attestationObject
AuthenticatorAttestationResponse
RP JavaScript Application
WebAuthnAPI
Browser
relying party id, user info, relying party info, clientDataHash
2
4
new public key, credential id, attestation
attestationObject
Authenticator
3
user verification, new keypair, attestation

Figure 3: Webauthn Registration Flow (Source: W3C [33])

2.2.1. FIDO2 Registration

The FIDO2 registration process is a foundational step in establishing a secure, passwordless authentication mechanism between the client and the RP. This process creates a unique cryptographic credential associated with the user's authenticator and registers it with the RP. The registration flow depicted in Figure 3 illustrates this process in detail, which can be broken down into several key steps:

0. The user initiates a key registration, usually by clicking a button on the RPs web page. This action triggers an HTTP POST request to the RPs Representational State Transfer (REST) API endpoint `webauthn/register-begin`. Because not every user may be authorized to enroll a new credential, a preliminary authentication step is often required. This preliminary step usually involves traditional authentication methods, such as entering a username and password or email verification and may be augmented by additional verification methods like One-Time Passwords (OTPs) or existing FIDO credentials. It's important to note that this initial authentication phase falls outside the scope of the FIDO2 key registration process.

1. Following the initial request, the RP constructs a *PublicKeyCredentialCreationOptions* object, which encapsulates all necessary parameters for the registration ceremony. This comprehensive set of parameters includes the RPs identifier, the RPs name, a unique challenge, and various specifications such as the desired type of authenticator, the level of attestation required, user verification needs, and the cryptographic algorithms the RP is prepared to use for the newly created public key credentials. Additionally, this object encompasses user information, including a

unique user ID assigned by the RP, as well as the user's name and a display name that is human-readable. These user details—specifically the user name and display name—are linked back to the user's identity as verified in the preliminary authentication step outlined previously. For an exhaustive breakdown of all parameters, please refer to the detailed listing provided in Section 5.4 of [31].

After assembling the *PublicKeyCredentialCreationOptions* object, the RP securely transmits it to the client as a JavaScript Object Notation (JSON) payload through an HTTP response. The JavaScript Application then calls the `navigator.credentials.create()` method, passing the object as an argument. This action effectively delegates the operation to the browser.

2. In the next step, the browser forwards the request to the authenticator to begin the creation of a public key credential, utilizing the `authenticatorMakeCredential()` function. During this process, the browser generates *clientData*, which includes the operation type, denoted by the string "webauthn.create", the challenge issued by the RP (encoded in Base64url), and the origin. The origin in web security is a combination of the scheme (in this case always HTTPS), host (domain or IP), and implicitly the port of a URL, which together define the source of a web request. By encapsulating the scheme and host, the origin effectively anchors the registration request to the specific web application that initiated the call to the WebAuthn API.

   Additionally, the browser performs a validation check to ensure that the RP ID specified in the WebAuthn request corresponds to the origin's registerable domain suffix, thereby confirming that the request is legitimately intended for the designated RP. Should an RP ID not be provided, the origin itself is used as the RP ID.

   The JSON-serialized *clientData*, after being hashed with Secure Hash Algorithm (SHA)256, and combined with user and RP details, *pubKeyCredParams*, along with any additional optional parameters specified by the server, is transmitted to the authenticator using CTAP. This information is encoded in Concise Binary Object Representation (CBOR) to ensure efficient processing. Detailed information on all these parameters can be found in Section 5.1 of [31].

3. The authenticator, upon receiving the request, asks the user to confirm their presence and intent to register. This user interaction could be a simple gesture like tapping a button on the authenticator device, entering a PIN, or performing a biometric verification such as a fingerprint scan, depending on the authenticator's capabilities and the level of user verification required by the RP. Once the user's consent is obtained, the authenticator generates a new public/private key pair specifically for this registration. For discoverable credentials, the private key is securely stored on the authenticator. In contrast, non-discoverable credentials are not stored on the authenticator; instead, the private key is encrypted with a master secret unique to the authenticator and then sent back to the RP as a wrapped private key. This approach offers the significant advantage that an authenticator

can generate and manage a practically unlimited number of credentials without the need for extensive memory capacity. It is especially beneficial for hardware tokens, which typically have limited storage, enabling them to serve numerous RPs and user accounts.

4. The *attestationObject*, which is sent back to the client, is a CBOR-encoded structure that includes the *attestationStatement* and *authenticatorData*. The *authenticatorData* itself encompasses the newly generated public key and a credential ID (which in turn is the wrapped private key in case of a non-discoverable credential), while the *attestationStatement* provides proof of the authenticator's authenticity to the RP. This statement contains information about the authenticator model, its cryptographic key, and may also include a certificate from the manufacturer and a signature. It is important to acknowledge that the provision of attestation is optional; should the RP decide against requiring attestation, a Trust On First Use (TOFU) approach is adopted instead. The detailed CBOR encoded structure of the *attestationObject* can be found in Section 6.5 of [31].

   The browser gathers the *attestationObject* and encapsulates it within a *PublicKeyCredential* object, which is formatted as a JSON structure. Alongside the *attestationObject*, this object contains additional details, including the type of public key generated and the *clientData* utilized during the registration process. Upon completion, the `navigator.credentials.create()` method returns this JSON object to the JavaScript application as its output.

5. Upon receiving the PublicKeyCredential object, which encapsulates the *attestationObject*, the JavaScript application initiates an HTTP POST request. This request is directed to the `webauthn/register-finish` endpoint on the RPs server. The payload of this request is the *AuthenticatorAttestationResponse*, comprising both the *attestationObject* and *clientDataJSON*. These elements are serialized in JSON format.

6. The next step involves a series of validations to ensure the integrity and authenticity of the data provided. The RP first decodes the *AuthenticatorAttestationResponse* from its JSON serialization to extract the *attestationObject* and *clientDataJSON*. The RP then validates the *clientDataJSON* to confirm that the credential creation was initiated by a legitimate request from the user. This validation checks that the challenge in the *clientDataJSON* matches the challenge originally sent by the RP in the *PublicKeyCredentialCreationOptions*, and that the operation type is indeed "webauthn.create".

   Next, the RP processes the *attestationObject* to verify the *attestationStatement* and to extract the newly generated public key and credential ID. Depending on the RPs policy, the signature of the attestation is validated against a PKI. Once the attestation is verified and the credential is validated, the RP stores the credential ID and public key associated with the user's account in a database. This step completes the user's registration process, enabling the newly registered credential to be used for authentication in future login attempts.

7. The final step in the FIDO2 registration process involves the RP responding back to the client with the outcome of the registration attempt. Once the RP has successfully validated the *AuthenticatorAttestationResponse*, including the *attestationObject* and *clientDataJSON*, and stored the new credential, it prepares a response to inform the client of the successful registration. This response is typically sent as an HTTP response to the JavaScript application.

   The response typically includes a status message indicating the successful registration of the credential, information about the authenticator if attestation was performed, and may also include additional details relevant to the user, such as a unique user identifier. It is important for the JavaScript application to handle this response appropriately, often by updating the user interface to notify the user of the successful completion of the registration process. It is worth noting that this final response from the RP to the client is not depicted in the illustration provided in Figure 3. The figure focuses on the core technical aspects of the FIDO2 registration process and omits the final step of the ceremony, where the outcome is communicated.

2.2.2. FIDO2 Authentication

The authentication process in FIDO2 is designed to ensure secure and efficient user verification using previously registered credentials. Key to this process is the authenticator's capability to cryptographically prove that it is in possession of its private key. The authentication flow is depicted in Figure 4, illustrating the sequence of steps and interactions between these components:

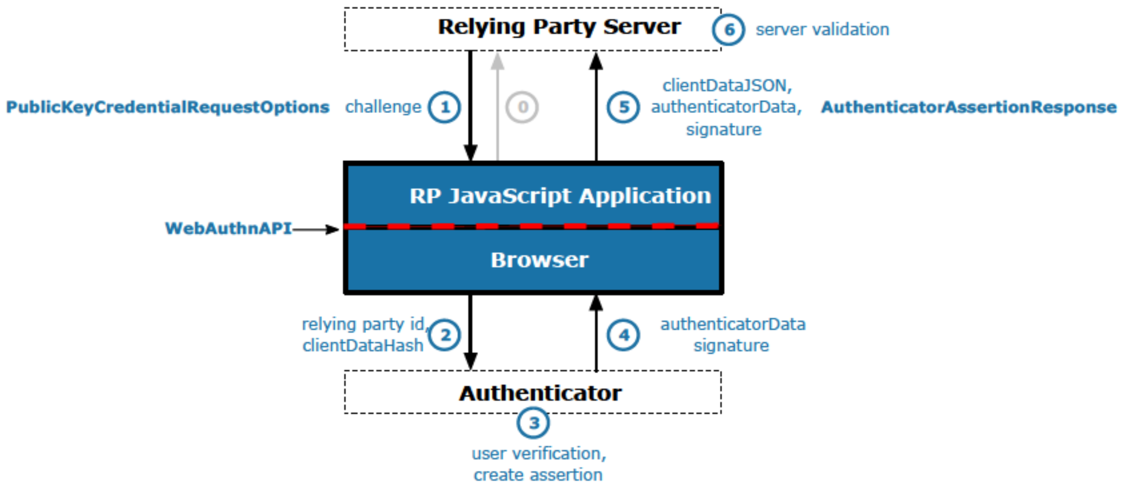
Relying Party Server
6 server validation
PublicKeyCredentialRequestOptions
challenge 1
0
5 clientDataJSON, authenticatorData, signature
AuthenticatorAssertionResponse
RP JavaScript Application
WebAuthnAPI
Browser
relying party id, clientDataHash 2
4 authenticatorData signature
Authenticator
3 user verification, create assertion

Figure 4: Webauthn Authentication Flow (Source: W3C [32])

0. The authentication process typically begins when the user attempts to access a service or application requiring authentication. The user interacts with the RP's web interface, usually by entering a username or selecting an account. This action triggers an HTTP POST request to the RPs `webauthn/authenticate-begin` endpoint. This step is crucial for the RP to initiate the WebAuthn authentication process and prepare the necessary parameters for the authenticator.

1. The RP, upon receiving the authentication request, creates a *PublicKeyCredentialRequestOptions* object. This object contains several essential parameters, including the RP ID, a new unique challenge, and user verification requirements. Crucially, it specifies *allowedCredentials*, which are the credential IDs registered during the registration process. However, including *allowedCredentials* in the *PublicKeyCredentialRequestOptions* can be optional when the authenticator is in possession of discoverable credentials. With discoverable credentials, it can identify the correct key using only the RP ID.

   Moreover, the RP ID parameter in the *PublicKeyCredentialRequestOptions* is also subject to conditions: While it is generally necessary to specify the RP ID if it differs from the RPs origin, this parameter is optional when the RP ID is the same as the origin. In such cases, the origin itself is used as the RP ID. It is essential to use the same RP ID for both the registration and authentication ceremonies. If different RP IDs are used, the authenticator won't match the credentials, causing the authentication to fail.

   After preparing the *PublicKeyCredentialRequestOptions*, the RP sends it to the client as a JSON payload via an HTTP response. The client's JavaScript application, receiving this payload, calls the `navigator.credentials.get()` method of the WebAuthn API, passing the object as an argument. For a comprehensive list of the *PublicKeyCredentialRequestOptions* parameters, refer to Section 5.5 [31]

2. The browser, acting as an intermediary between the JavaScript application and the authenticator, forwards this request to the authenticator through CTAP. During this step, the browser constructs *clientData*, similar to the registration process. The *clientData* includes the operation type "webauthn.get", the challenge from the RP, and the *origin*. The browser then hashes the JSON-serialized *clientData* using SHA-256. The hashed *clientData*, along with the *PublicKeyCredentialRequestOptions*, is transmitted to the authenticator via CTAP.

3. In scenarios where *allowedCredentials* are not specified in the request—typical for discoverable credentials—the authenticator itself determines the appropriate credential. It does this based on the RP ID and the user ID, both of which are part of the request. The authenticator internally locates the credential that matches these identifiers. If there are multiple matching credentials, the authenticator may prompt the user to select the desired one.

   For non-discoverable credentials, where *allowedCredentials* are explicitly provided in the request, the authenticator uses the provided credential ID to directly locate the corresponding user's private key. In both cases, once the appropriate credential is identified, the authenticator requests user interaction to confirm the intention to authenticate, usually by checking user presence and/or doing user verification. This interaction can vary from a simple gesture to more stringent methods like a PIN or biometric verification, tailored to the authenticator's capabilities and the RPs's requirements. After obtaining the user's consent, the authenticator then generates an assertion using the identified private key.

4. The authenticator then sends the response back to the client. This response, known as an *AuthenticatorAssertionResponse*, includes the *authenticatorData* and the *signature*. The *authenticatorData* contains information about the authenticator and the user interaction—just like during key registration—, while the *signature* is generated by signing the concatenation of the *authenticatorData* and the hashed *clientData* using the user's private key.

5. The client's JavaScript application receives the *AuthenticatorAssertionResponse* and forwards it to the RPs server by making an HTTP POST request to the '`webauthn/authenticate-finish`' endpoint. The payload of this request includes the *AuthenticatorAssertionResponse*, which contains the *authenticatorData*, the *signature*, and the *clientDataJSON*. The full specification of the *AuthenticatorAssertionResponse* can be found in Section 5.2.2 of [31].

6. Upon receiving the *AuthenticatorAssertionResponse*, the RP performs a series of validations to ensure the authenticity and integrity of the authentication attempt. The RP verifies the *signature* using the user's public key that was registered and stored during the initial registration process. It also checks the *clientDataJSON* to confirm that the operation type is "webauthn.get" and that the challenge matches the one sent in the *PublicKeyCredentialRequestOptions*. Additionally, the RP validates the *authenticatorData* to ensure the correct credential ID was used and that the user was present during the authentication process. If all validations are successful, the RP confirms the user's identity by granting access to the requested service or application.

7. The RP then communicates this outcome back to the client, typically via an HTTP response. This communication informs the JavaScript application of the successful or unsuccessful authentication, enabling it to react to the result by either providing or denying access to the requested web resource. It is important to note that, just like in the registration steps provided earlier, this final message from the RP to the client is not depicted in Figure 4.

2.3. EAP

EAP is an authentication framework widely used in wireless networks and Point-to-Point connections. Rather than specifying how to authenticate users, EAP allows protocol designers to build their own EAP methods, subprotocols that perform the authentication transaction [9, p. 130]. Unlike traditional authentication mechanisms that operate at the application layer, EAP operates directly on the link layer, allowing it to support multiple authentication methods without depending on the transport protocol. The 802.1X standard specifies how EAP should be encapsulated and transported across networks, whether it is a wired Local Area Network (LAN) or a wireless LAN. The standard defines the encapsulation of EAP within Ethernet frames and operates on the Data Link Layer. In practice, 802.1X secures the network by preventing unauthorized access until the user or device successfully authenticates with the network authentication server. This is typically achieved by proving possession of the private key corresponding to a registered digital certificate, or by submitting credentials through a secure tunnel.

EAP exchanges are composed of requests and responses. The authenticator sends requests to the system seeking access, and based on the responses, access may be granted or denied [9, p. 131]. At the conclusion of an EAP exchange, the user has either authenticated successfully (EAP-Success) or has failed to authenticate (EAP-Failure). Success and Failure frames are not authenticated in any way [9, p. 134]

802.1X outlines three key entities in the authentication dialogue. The *Supplicant* (or *Station* for wireless networks) refers to the end-user device requesting access to network services. The authenticator (or Access Point (AP) for wireless networks) serves as the gatekeeper to network access. It forwards all incoming requests to an *Authentication Server*, like a Remote Authentication Dial-In User Service (RADIUS) server, for verification and processing. EAP methods define the actual authentication mechanism used during the EAP exchange between the supplicant and the authenticator. Some of the most commonly used EAP methods include:

- **EAP-TLS**: Uses TLS with client certificates for mutual authentication. Despite its security, EAP-TLS is not widely adopted because it requires every network user to have a digital certificate. The complexity of generating, distributing, and verifying these certificates poses significant challenges. Organizations with an existing PKI find it easier to implement, many others opt for alternative methods to avoid the burdens of establishing a PKI [9, p. 137].

- **EAP-TTLS** (or tunneled TLS): Uses TLS to create a secure tunnel with server-side authentication only, eliminating the need for client certificates. The purpose of the TLS tunnel is to provide encryption for another authentication protocol that verifies the client's identity to the network. The TLS tunnel is often called the "outer" authentication, as it protects the "inner" authentication process. The inner authentication can employ various methods, from straightforward passwords to more complex challenge-response mechanisms. EAP-TTLS is widely adopted as it allows organizations to integrate with existing authentication infrastructures, such as Windows domains, Lightweight Directory Access Protocol (LDAP) directories, or Kerberos realms, without the necessity of establishing a separate PKI system.

- **EAP-PEAP** (or protected EAP): Similar to EAP-TTLS, it creates a secure TLS tunnel to protect subsequent authentication exchanges. The slight difference between TTLS and PEAP is in the way the inner authentication is handled.

TTLS uses the encrypted channel to exchange attribute-value pairs (AVPs), while PEAP uses the encrypted channel to start a second EAP exchange inside of the tunnel [9, p. 138].

- **EAP-MD5**: A simple method that uses Message-Digest Algorithm (MD5) hash function for authentication. Due to its lack of mutual authentication and encryption, it is considered less secure than TLS-based methods.
- **EAP-FAST**: Developed by Cisco as an alternative to PEAP and EAP-TTLS, it uses a protected access credential (PAC) for establishing a TLS tunnel and supports multiple inner authentication methods.

2.3.1. EAP-TLS

This thesis introduces a unique modification to an 802.1X network that uses EAP-TLS. To understand the context of this modification, the EAP-TLS handshake process is examined in detail in the following section. As a matter of notation, packets transmitted as part of an EAP method exchange are written Request/Method when they come from the authenticator, and Response/Method when they are sent in response [9, p. 134]. From the *Supplicant* to the *Access Point*, the protocol is EAP over LAN (EAPOL), as defined by 802.1X. From the *Access Point* to the *Authentication Server*, EAP is carried in RADIUS packets. Some documentation may refer to it as "EAP over RADIUS." Figure 5 illustrates the sequential steps that occur during the handshake process:

1. The process begins with the presumption that the *Supplicant* is already associated with the *AP*. The *Supplicant* initiates the handshake by transmitting an EAPOL Start message, signaling the beginning of the authentication sequence.
2. The *Authenticator* sends an EAP Request/Identity message to the *Supplicant*, soliciting the user's identity.
3. In response, the *Supplicant* provides an EAP Response/Identity packet that includes its identity information back to the *Authenticator*.
4. The *Authenticator* then relays this identity information to the *Authentication Server* encapsulated within a RADIUS Access-Request packet, initiating the backend authentication process.
5. Following the exchange of identity information, the authentication process advances to the crucial phase where a full TLS handshake with client certificates is performed. Unlike a standard TLS handshake, as described in Section 2.1.1, the EAP-TLS handshake involves the *AP* as an intermediary. Each TLS message generated during this handshake is encapsulated within EAP request and response packets for transport. Communication is bidirectional: the *Supplicant* sends these packets to the *AP* via EAPOL, which then forwards them to the *Authentication Server*

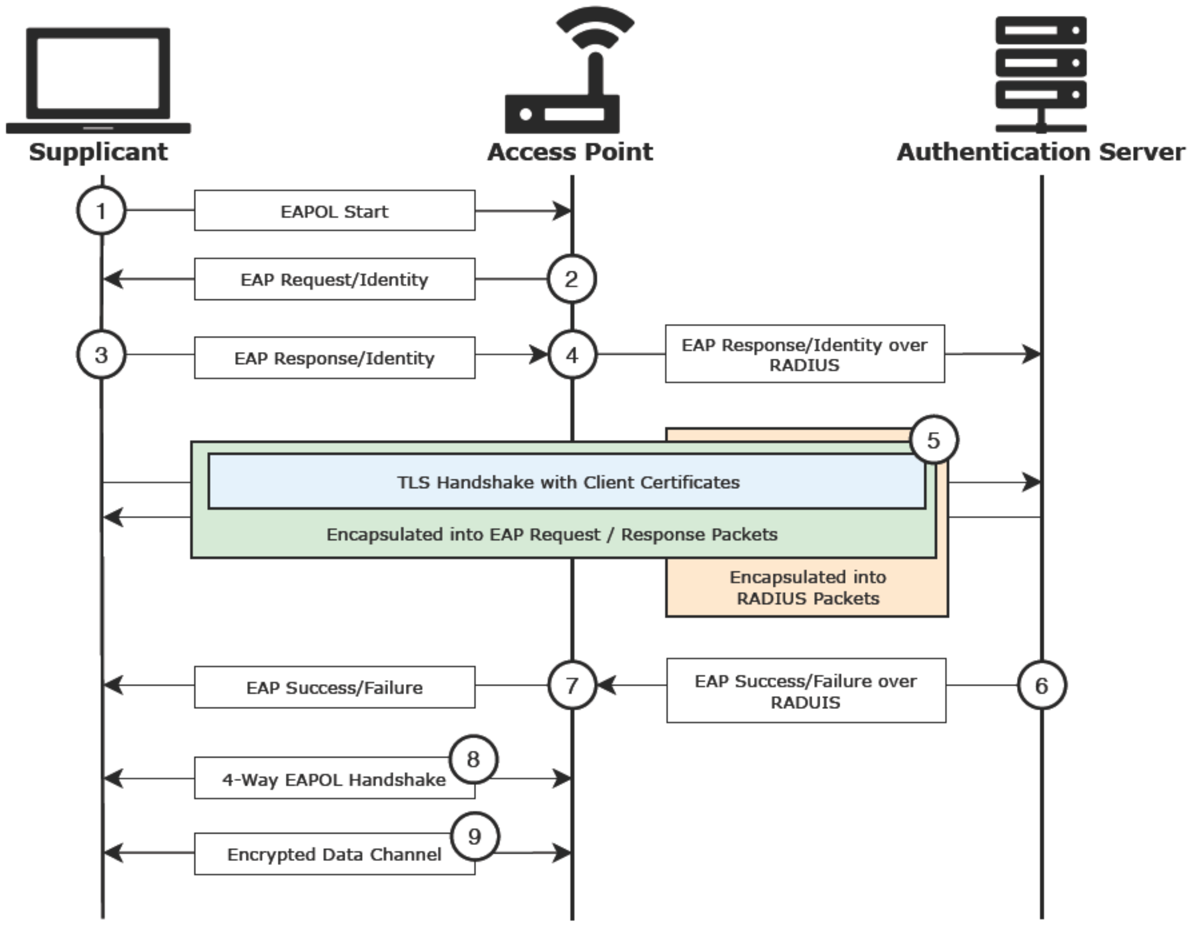
Supplicant
Access Point
Authentication Server
1
EAPOL Start
2
EAP Request/Identity
3
EAP Response/Identity
4
EAP Response/Identity over RADIUS
5
TLS Handshake with Client Certificates
Encapsulated into EAP Request / Response Packets
Encapsulated into RADIUS Packets
EAP Success/Failure
7
EAP Success/Failure over RADUIS
6
4-Way EAPOL Handshake
8
Encrypted Data Channel
9

Figure 5: EAP-TLS Handshake in 802.1X. Graphic created by author based on RFC 5216 [24].

encapsulated within RADIUS packets, and vice versa, ensuring a complete run of the TLS handshake.

Once the handshake is completed, the TLS tunnel is not utilized for the transmission of any application data. Instead, it is closed and the established shared secret is retained for subsequent use in the 4-Way EAPOL handshake.

6. If the TLS handshake is successful, the *Authentication Server* sends an EAP Success packet to the *Access Point*, which is transmitted within a RADIUS packet.

7. Upon receiving the EAP Success packet, the *Access Point* then relays this packet to the *Supplicant* via an EAPOL frame, indicating the successful authentication of the *Supplicant*.

8. Following the successful EAP authentication, the 4-way EAPOL handshake between the *Supplicant* and the *AP* is conducted. This phase utilizes the shared secret derived from the TLS handshake as input for generating the Pairwise Master Key (PMK). The PMK is then used to derive the session's encryption keys, ensuring secure communication under the Wi-Fi Protected Access (WPA)2/3 protocols.

9. Once session keys are derived, a secure data channel is activated. While Counter Mode Cipher Block Chaining Message Authentication Code Protocol (CCMP) is commonly employed in WPA2 for robust encryption, WPA2/3 networks might utilize Galois/Counter Mode Protocol (GCMP) in WPA3 for enhanced security, depending on network settings and device capabilities.

3. Related Work

The essence of this thesis revolves around investigating the potential for FIDO, an authentication framework initially designed for web applications, to be adapted for integration within the TLS protocol. This attempt requires the disentanglement of FIDO from its native HTTP and JavaScript environments, alongside a conceptual abstraction of certain implementation specifics, thereby facilitating its application in alternative contexts that rely on client authentication. The subsequent section on related work outlines various studies and explorations concerning such an adaptation. These include both successful implementations and theoretical investigations that, while not yielding fully operational outcomes, contribute significantly to the discourse on extending FIDOs utility beyond its original web-centric domain.

3.1. Passwordless VPN

In the work titled *"Passwordless VPN using FIDO Security Keys: Modern Authentication Security for Legacy VPN Systems"* [10], Emin Huseynov explores an innovative method to incorporate FIDO authentication within legacy Virtual Private Network (VPN) protocols, notably Layer 2 Tunneling Protocol (L2TP), which typically do not support advanced authentication mechanisms. The paper emphasizes the significance of Two-Factor Authentication (2FA) in VPN systems to protect against phishing and other security threats. Most existing VPN systems, however, lack native support for 2FA, leading to workarounds that are vulnerable to social engineering attacks.

To address these challenges, the paper proposes a web-based VPN portal solution that enables the use of FIDO security keys as either a single or a second-factor authentication method. This solution necessitates that the user possesses a modern web browser and requires the VPN service to provide a public-facing web interface, in addition to the VPN server. The client connects to this web interface and performs a traditional FIDO authentication. Upon successful authentication, the FIDO authentication server engages the RADIUS server to create a temporary user record in its database. The temporary username and password are transferred to the end user in text format, allowing the user to copy and paste the credentials into the VPN client. Alternatively, this can be done in a more user-friendly manner, such as through a batch file containing all the necessary credentials and settings, which automatically establishes the VPN connection when executed. Since the FIDO authentication server and the VPN server are located behind the same private firewall, the communication between them is considered secure. Upon the establishment of the VPN connection with the temporary credentials, said credentials are immediately deleted from the server's database to prevent credential reuse.

While this approach does not fully integrate FIDO into the VPN protocol, it constructs a bridging infrastructure that logically connects both protocols. It is critical to acknowledge, however, that this solution's practicality is diminished due to the cumbersome process of transferring temporary credentials between systems, affecting overall user experience.

3.2. FIDO2 authentication in OpenSSH

Secure Shell (SSH) operates on a client-server model to enable secure communication over unsecured networks. In this protocol, the client—often an SSH-agent—initiates a connection to the SSH-server. After successful authentication, which typically involves public-key cryptography, a secure and encrypted channel is established between them. This channel ensures confidentiality and authenticity in data exchange, allowing for secure remote login, file transfers, and command execution. SSH has seen various implementations, with *OpenSSH* being one of the most widely recognized and utilized.

Beginning with *OpenSSH 8.2* (released in February 2020), built-in support for FIDO and U2F hardware authenticators has been introduced. Unlike the core SSH protocol, which is thoroughly outlined in RFC 4251 to RFC 4256, there is no formal standard detailing the integration of FIDO within the SSH framework. As a result, the use of FIDO as an authentication method is a distinct feature of OpenSSH, potentially limiting compatibility across different SSH implementations. Given the absence of a formal standard, it is challenging to precisely understand how this integration operates. Nevertheless, analysis of `OpenSSH`'s changelog, examination of git commit messages, and review of the source code facilitate certain insights into this extension [1] [2].

OpenSSH supports Public-Key Cryptography Standards (PKCS)#11 for standard cryptographic operations with hardware tokens and smart cards. However, directly integrating FIDO with PKCS#11 has proven to be unfeasible. As noted by the `OpenSSL` developer team, "the U2F protocol cannot be trivially used as an SSH protocol key type as both the inputs to the signature operation and the resultant signature differ from those specified for SSH. For similar reasons, integration of U2F devices cannot be achieved via the PKCS#11 API" [17]. These constraints required the adoption of a new middleware library approach to bridge the gap between OpenSSH and FIDO hardware tokens. This middleware can be configured within OpenSSH via the *"SecurityKeyProvider"* directive in the `sshd_config` file or through the `SSH_SK_PROVIDER` environment variable. This approach enables OpenSSH to interface with a wide array of FIDO hardware tokens, including those that may be developed in the future, without requiring significant changes to the core OpenSSH codebase.

FIDO authentication in OpenSSH introduces two new key formats:

- `sk-ecdsa-sha2-nistp256@openssh.com`
- `sk-ssh-ed25519@openssh.com`

The former utilizes the Elliptic Curve Digital Signature Algorithm (ECDSA) algorithm over the National Institute of Standards and Technology (NIST) P-256 curve, while the latter employs the Edwards-curve Digital Signature Algorithm (EdDSA) over the elliptic curve 25519. The *"sk-"* prefix stands for "Security Key" and indicates that these private/public key files require an external hardware token.

In addition to the underlying signature primitive, these key formats incorporate additional information within both the public and private keys, as well as within the signature object itself. In the public key file, an *application* string has been incorporated, representing the RP ID of the FIDO protocol. By default, this application string is set to *"ssh:"*, although it can be overridden during key enrollment. In the private key file, several attributes have been introduced. These include the same *application* string as in the public key, a *Flags* byte containing information on whether user verification and user presence checks are required, and a *key handle* corresponding to the credential ID in the FIDO protocol.

The ceremony for enrolling discoverable and non-discoverable credentials diverges significantly from the traditional FIDO protocol. Unlike the standard FIDO flow, it does not involve the RP (in our case, the SSH-server). Instead, keys are generated locally on the client machine using `ssh-keygen`, similar to typical SSH key generation. Throughout this process, a hardware token must remain attached. Once the private and public key files have been generated, the public key component can be transferred from the client to the server using the command line tool `ssh-copy-id`. In contrast to conventional FIDO, the user ID is not determined by the RP, but is left empty. Given that an SSH endpoint is inherently tied to a single user account, there is usually no need for the SSH-server to multiplex between users. Typically, each SSH server is associated with a single user account, although exceptions may exist, such as shared environments. If multiple SSH resident keys on a single token are desired, then it may be necessary to override the default RP ID or user ID values using the `ssh-keygen "-O application="` or `"-O user="` options.

The integration of FIDO into OpenSSH represents a significant deviation from the original FIDO protocol. It liberates the RP from the necessity of maintaining state in the form of a database. The only user context an SSH-server needs to retain is the public key and the RP ID, both stored within the new formats of public key files. In addition, this deviation from the FIDO protocol unintentionally introduces a security enhancement for non-resident keys, making them more resilient against stolen FIDO tokens. An attacker in possession of a stolen FIDO token without PIN protection can authenticate with any RP registered with the credentials on the token, regardless of whether it is discoverable or non-discoverable. In the case of non-discoverable keys, the credential ID required to derive the private key on the token is provided by the RP during authentication. However, in OpenSSH, the key handle is stored on the client's device, requiring the attacker to have both possession of the token and somehow obtain the private key file under `$HOME/ssh/`. Nevertheless, once users set a PIN on their tokens, stolen tokens become ineffective for attackers.

3.3. EAP-FIDO

In October 2023, Stefan Winter and Jan-Frederik Rieckers from Deutsches Forschungsnetz (DFN) published an Internet-Draft entitled *"EAP-FIDO"* within the Internet Engineering Task Force (IETF), proposing a new EAP method leveraging FIDO authentication [22]. Although still a work in progress at the time of writing, the draft's general procedure is outlined clearly. Similar to EAP-TTLS, as described in Section 2.3, the protocol consists of two phases: the TLS handshake phase (the outer authentication) and the FIDO exchange phase (the inner authentication). In the TLS handshake phase, TLS is employed to authenticate the server to the client. Subsequently, the established TLS tunnel safeguards the inner FIDO authentication, which authenticates the client to the server. Upon successful completion of the FIDO exchange, the client and server can implicitly derive keying material from the TLS handshake phase. The messages involved in the inner FIDO authentication are encoded in CBOR format before being transmitted via the TLS record layer. These messages adhere to a custom format defined within the protocol specification, encompassing elements such as message types, attributes, and error codes.

It is important to note that the EAP-FIDO draft focuses exclusively on the authentication phase of utilizing FIDO as EAP-method. Key enrollment and management processes are beyond the scope of this document. Instead, these processes are expected to be carried out using traditional HTTP-based web interfaces, which provide a well-established, secure, and user-friendly mechanism for enrolling FIDO keys with the respective services.

The PoC implementation of the EAP-FIDO draft also served as an initial investigation into the practicality and efficiency of using this novel EAP method in eduroam networks. Considering eduroam's emphasis on user experience and relatively modest security demands, utilizing platform authenticators such as the Trusted Platform Module (TPM) and Apple's Secure Enclave could provide robust FIDO authentication while ensuring the convenience of network users by eliminating the need for external hardware tokens. Platform authenticators, combined with silent authentication (which entails no user verification or presence checks), would maximize usability, offering users a seamless experience where they are not involved in any interaction.

3.4. FIDO as TLS Extension

In his thesis *"FIDO2 als TLS-1.3-Erweiterung"*, written in German in 2020, Tom-Lukas Johann Breitkopf lays the groundwork for the concept that this research seeks to further explore and expand upon: the integration of the FIDO authentication ceremony within the TLS handshake as a TLS extension [6]. Breitkopf outlines an architecture for such integration including the detailed structure and encoding of messages exchanged between peers. Although Breitkopf's thesis does not address key enrollment, it does focus on the authentication ceremony, specifically addressing both discoverable and non-discoverable credentials. He introduces two modes of operation within the TLS-FIDO extension: *"FI-mode"* for authentication with discoverable credentials, maintaining the protocol flow of the original FIDO specification, and *"FN-mode"* for non-discoverable credentials. Unlike the FI-mode, FN-mode effectively doubles the TLS handshake process by executing two consecutive TLS handshakes in a row, each augmented with extension data to ensure the secure transmission of the client's identity.

While Breitkopf uses the standard TLS extension mechanism described in RFC 8446 to signal the desire for FIDO authentication to the remote peer, the FIDO challenge-response messages are not carried as extension data within existing TLS handshake messages. Instead, he incorporates new message types into the handshake, positioned precisely where the server's *CertificateRequest* and the client's *Certificate* (as depicted in Figure 1) would typically be sent. Yet, these messages diverge by carrying FIDO challenge-response content, maintaining the traditional sequence, but altering the payload to facilitate FIDO authentication. Despite introducing new message types to the handshake, Breitkopf's modified TLS library maintains interoperability with existing TLS implementations. Should a peer not recognize the FIDO extension, it simply disregards the initial signal for FIDO authentication, allowing the communication to proceed under the standard TLS protocol.

Breitkopf not only offers a detailed specification and critical assessment of alternatives but also brings his concepts to life through a PoC using the `tlslite-ng` Python library. The selection of this TLS library is advantageous for its ease of prototyping and rapid development capabilities. However, it is noteworthy that `tlslite-ng` is not commonly used in production environments, highlighting a distinction between its use for conceptual demonstration and practical application. To bridge that gap, another thesis (authored by Mario Freund) addresses the need for a more production-suited implementation of the proposed concept [8]. Freund's focus lies in the implementation of the FIDO-TLS extension using `GNUtls`, a TLS library written in the low-level language C. Given that Freund's work predominantly builds upon the concepts and foundational research provided by Breitkopf, this thesis does not receive further attention in the current discussion.

3.5. Summary

The investigation of FIDO authentication in various settings, as outlined in the previous sections, demonstrates its adaptability and potential for broader application beyond its original design for web applications. The development of a bridge infrastructure for using FIDO with legacy VPN systems, its incorporation into OpenSSH, the introduction of a new EAP method, and the proposal to integrate FIDO as a TLS extension are indicative of the versatility and growing relevance of FIDO in different technological domains.

In the following chapter, significant focus will be placed on examining Breitkopf's methodology for integrating FIDO into the TLS protocol. Building upon Breitkopf's foundational ideas, this thesis extends his concept to not only cover the authentication ceremony, but also to integrate the credential enrollment process into the TLS handshake. Additionally, it seeks to extend and improve the specification of message types, focusing on elevating the security and operational efficiency of the overall system.

4. Methodology

In traditional network architecture, as described by the OSI model, each layer is assigned specific responsibilities, irrespective of its underlying internal structure and technology. This hierarchical approach not only simplifies the understanding and troubleshooting of network issues but also establishes a clear conceptual separation between the functions of each layer [13, p.77].

According to the OSI model, TLS should ensure confidentiality, integrity, and endpoint authentication for data transmission at the Transport Layer. Traditionally, TLS has been designed to provide this mutual authentication, however, in practice, the mechanisms available for client authentication within TLS—such as client certificates, TLS-Pre-Shared Key (PSK), and Kerberos—have not achieved widespread popularity or usability. The complexity inherent in managing client certificates, along with the operational challenges of deploying TLS-PSK or Kerberos in diverse environments, have often led to client authentication being either omitted or addressed through alternative solutions. Emerging as a notable advancement, FIDO aims to resolve many of these usability and security challenges but traditionally does so at the Application Layer. This deviation introduces a cross-layer design where authentication extends beyond its intended OSI layer, embedding authentication logic into higher-level application processes.

This deviation from the OSI model is not just theoretical but has practical implications. Implementing authentication at the Application Layer requires that each application protocol independently incorporates its own authentication mechanism. Although it is entirely feasible to integrate FIDO to protocols such as Simple Mail Transfer Protocol (SMTP), EAP, Lightweight Directory Access Protocol (LDAP), File Transfer Protocol (FTP), Message Queuing Telemetry Transport (MQTT), Network News Transfer Protocol (NNTP) and OpenVPN individually, integrating it directly into the Transport Layer presents a more universally applicable and protocol-agnostic approach. Applications that rely on the Transport Layer can inherently utilize its functions, ensuring that FIDO's authentication capabilities are readily available across a wide range of services without the need for individual protocol adjustments.

The core idea of this thesis is to integrate FIDO into the TLS protocol, thereby restoring the conceptual integrity of the OSI model and aligning security mechanisms with the appropriate layer. This integration aims to standardize authentication processes across different protocols by embedding them directly within the Transport Layer, rather than at the Application Layer. There is no need for the application layer to implement a protocol-specific authentication mechanism or the protocol logic that terminates the TLS channel in the event of an authentication failure. Once the TLS channel is successfully established, the application can be assured that all required safety properties, including endpoint authentication, are fulfilled.

The following chapter details the methodology for integrating FIDO with TLS, covering design principles and requirements, the network and threat model, and strategies for embedding FIDO ceremonies as TLS extensions. It also discusses the definition of message types, their structure, and encoding. Additionally, the chapter explores how errors are communicated within the system via TLS alerts and discusses the communication of

FIDO outcomes to the application layer.

4.1. Design Principles & Requirements

To ensure the successful integration of FIDO authentication and key enrollment with the TLS protocol, it is crucial to define a robust set of design principles and requirements that guide the development process. These requirements are presented in descending order of importance.

- **Effectiveness**: The extension must uphold the existing security properties of TLS, providing confidentiality, integrity, and server authentication. Additionally, it should enhance TLS by integrating client authentication according to the FIDO standard, thereby augmenting the protocol's security framework with a robust, client authentication mechanism.
- **Privacy**: User identity and authentication details must be protected throughout the authentication process. This includes safeguarding sensitive information from unauthorized access or exposure during and after the authentication transaction.
- **Interoperability**:
  - The extension must be compatible with TLS implementations that do not support the extension, ensuring backward compatibility and broad applicability.
  - The extension should seamlessly integrate with existing FIDO tokens, enabling users to utilize their current authentication devices without encountering any disruptions or complications.
- **Protocol Agnostic**: In keeping with the original layer segregation of the OSI model, the extension should be agnostic to application layer protocols. This principle ensures that the solution can be universally applied across different application contexts that build on TLS.
- **Efficiency**: The integration's impact on performance is a critical consideration. Specifically:
  - The runtime of the FIDO ceremonies within TLS should not significantly exceed that of traditional FIDO authentication executed over HTTP.
  - The number of RTTs should not increase substantially, to prevent degradation in connection establishment times.
  - Packets sent over the wire should maintain a comparable size to, or preferably be smaller than, those observed with traditional FIDO ceremonies.
- **Scalability**: The solution must support scalability in terms of both the number of FIDO devices and users it can accommodate. This scalability is essential for widespread adoption and the long-term viability of the integration.

- **Modularity**: The design of the extension should enable easy integration into or removal from the TLS protocol stack. This flexibility facilitates the adoption of the extension in diverse environments and use cases.
- **Usability**: The user experience should be as seamless and straightforward as possible. This involves minimizing user interactions, thereby reducing the complexity and potential for user error.
- **Standardization**: Where possible, the extension should adhere to existing standards to promote interoperability and consistency across implementations.

4.2. Network & Threat Model

A network model is presented that illustrates the relationship between clients, servers, and adversaries within a potentially insecure network environment. This model forms the basis for the following discussions and analyses. The key components and their interactions within the model are detailed as follows:

- **Client**: The entity that integrates both TLS client and FIDO client capabilities. Communication between the client and the FIDO authenticator is assumed to be secure.
- **Server**: The entity that integrates both TLS server and FIDO server capabilities. Communication between the server and the FIDO backend, whether it be a database or an external FIDO authentication server, is considered secure.
- **Adversary**: The adversary is an entity that attempts to compromise the security of the client-server communication. The capabilities of the adversary in this model are the following:
  - **Eavesdropping**: Listening to communications between the client and the server.
  - **Man-in-the-Middle (MITM) Attacks**: Intercepting and potentially altering the communication between the client and server.
  - **Replay Attacks**: Capturing data from the network and retransmitting it to create unauthorized transactions or sessions.
  - **Phishing Attacks**: Deceiving users into providing sensitive information by masquerading as a trustworthy entity.

  The model assumes that the adversary has the capability to exploit vulnerabilities in the communication protocol and the network but does not have the ability to breach cryptographic functions.
- **Insecure Network**: The network connecting the client and server is considered insecure and is susceptible to the forms of interception and attacks as mentioned in the adversary's capabilities. This includes networks like the internet or any other environment where inherent security cannot be guaranteed.

- **Message**: This term describes a collection of related attributes that together form a meaningful unit in a communication process. Each attribute in a message contributes to its overall purpose, ensuring semantic cohesion. Messages are the basic units in communication protocols, encapsulating the necessary information to perform a specific operation or convey a particular piece of data. The message type acts as an identifier, distinguishing each message's role and function within the communication sequence.
- **Packet**: A packet is an assembly of one or more messages prepared for transmission across a network. It represents the physical packaging of these messages into a single, coherent unit that is sent over the network.

4.3. Extending TLS Peers

This subsection outlines the enhancements made to both the traditional TLS client and server to support FIDO authentication and key registration functionalities.

The extension of the TLS client involves equipping it with the capabilities to access and interact with FIDO authenticators. These authenticators can be external hardware tokens, such as USB security keys, or platform authenticators that are built directly into the user's device, such as biometric sensors or TPMs. The integration assumes that the TLS client, now also functioning as a FIDO client, has access to these authenticators through appropriate APIs. With this extension, the TLS client's responsibilities are expanded to include initiating and managing FIDO authentication and key registration ceremonies:

1. Detection of FIDO authenticators connected to the system and communication with them via CTAP.
2. Integration of the FIDO authentication and key registration process within the TLS protocol flow. Failure in FIDO authentication impacts the TLS handshake, enhancing security by ensuring only authenticated sessions are established.
3. Management of FIDO credentials, ensuring secure storage and retrieval during authentication or key enrollment.

The extension of the TLS server involves equipping it with the capabilities to interact with and verify FIDO authentication data provided by clients, as well as to register and manage FIDO keys. The specification does not dictate whether storage of credential information on the server side should occur via an internal/external database or through interaction with an external FIDO authentication server. This flexibility allows implementers to choose the most appropriate method based on their specific security needs, infrastructure, and operational preferences. Regardless of the backend architecture, the server must be capable of the following:

1. Verification of FIDO authentication data, such as signatures and attestation statements, which are sent by the client.
2. Storage of credential information, such as credential IDs, public keys and user data.
3. Changing the state of the TLS session based on the outcome of FIDO ceremonies.

4.4. Integration into the TLS handshake

Section 2.2 highlights that both FIDO ceremonies involve a series of message exchanges between the client and the server across a potentially unsecured network. These messages can be broadly classified into four abstract types: *Indication*, *Request*, *Response* and *Result*. Specifically, the initial messages in Step 0 of Section 2.2.1 and 2.2.2 are identified as registration or authentication *Indication*. The subsequent messages in Step 1 of both ceremonies fall under the *Request* category, either requests for registration or authentication. The messages in Step 5 represent registration or authentication *Response* messages. Lastly, step 7 corresponds to the *Finished* type.

4.4.1. Single Handshake

The primary strategy of this research involves encapsulating FIDO protocol messages for both authentication and key enrollment ceremonies within the TLS handshake. As detailed in Section 2.1.2, TLS 1.3 provides a mechanism for including arbitrary extension data within specific handshake messages. In a TLS 1.3 handshake that does not involve client certificates, the extension mechanism allows arbitrary data to be exchanged between the client and server within a single round-trip. Specifically, the client can transmit data to the server within the *ClientHello* message. The server, in turn, can include additional extension data in its response, which may be part of the *ServerHello*, *EncryptedExtensions*, or *Certificate* messages. When client certificates are used in a TLS 1.3 handshake, there is an additional opportunity for the client to send arbitrary data back to the server within its *Certificate* message, utilizing the extension mechanism. This capability extends the data exchange sequence to include a third transmission of arbitrary data: First from the client to the server in the *ClientHello*, second from the server to the client within the *ServerHello*, *EncryptedExtensions*, *Certificate*, or *CertificateRequest* messages, and third, from the client back to the server in the *Certificate* message of the client. Each of these steps utilizes the extension mechanism to transmit additional, potentially custom data, along with the standard required handshake information.

Integrating FIDO ceremonies within the TLS 1.3 handshake framework is feasible if the final FIDO message type, the *Result*, is not transmitted during the handshake itself. This approach allows the first three FIDO message types—*Indication*, *Request*, and *Response*—to align with the three exchange windows provided by TLS 1.3 when client certificates are used. It will be demonstrated in Section 4.8 that TLS alerts can be used to transmit the necessary information of the *Result* message, thus ensuring complete transmission of all FIDO message types. Following this strategy, Figure 6 illustrates an example of such integration, highlighting potential TLS handshake messages where

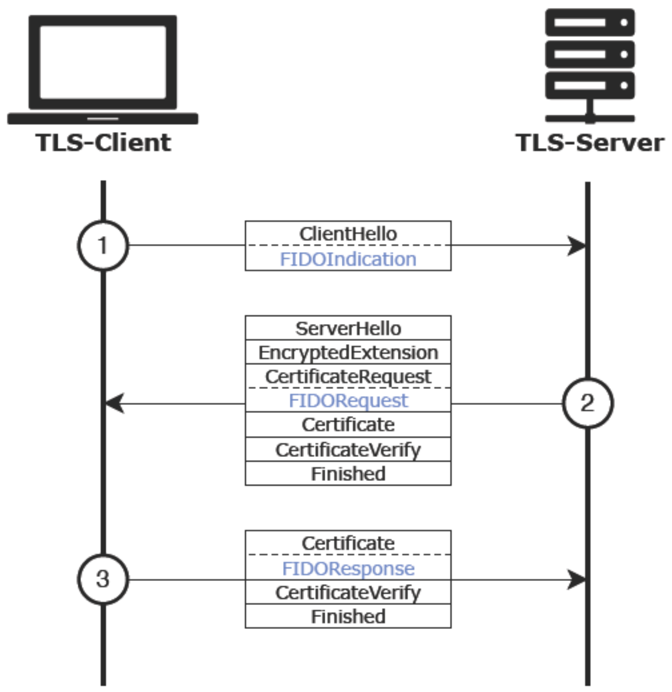

Figure 6: TLS1.3 Handshake with FIDO Messages Carried as Extension Data. Graphic created by author.

the FIDO messages could be appended. In the figure, message types indicated in blue represent new additions to the TLS protocol. Additionally, a dashed line between two message types signifies that the upper message carries the lower message as extension data.

The proposed integration of FIDO into TLS as an extension aims to replace existing client authentication methods. However, it may initially appear counterintuitive that this integration relies on a TLS handshake that includes client certificates. The reason for this approach is strategic: Using the client's *Certificate* as a carrier medium for the *FIDOResponse* allows FIDO to be integrated with only the official TLS extension mechanism. This does not require any fundamental changes to the TLS library. This method provides the necessary three-message exchange window required for both FIDO ceremonies. Moreover, the use of client certificates in this integration does not require server validation of these certificates, nor does it compel the client to provide a valid certificate; a placeholder can be used instead. This flexibility ensures that the authentication security is primarily managed by the FIDO mechanism rather than relying on TLS client certificate validity, which simplifies compliance and reduces the operational burden on clients.

4.4.2. Vulnerabilitiy of the Single Handshake Approach

On closer examination, the proposed integration exhibits some vulnerabilities that need careful consideration and mitigation. As explained in Section 2.1.2, the *ClientHello* message in TLS is not protected. For authentication with discoverable credentials, the *Indication* doesn't contain sensitive data; it merely signals the RP that authentication with discoverable credentials is intended. Since this information is not confidential, the proposed integration is suitable in this context. However, during key enrollment and authentication with non-discoverable credentials, the *Indication* message must include the client's identity. Given the privacy requirement of this study, protecting the client's identity is essential. Therefore, this straightforward approach is not viable here, as it would expose sensitive identity information in an unprotected *ClientHello* message.

Breitkopf, in his thesis, explores various strategies to address this design flaw [6, p. 27-30]. The first strategy he discusses is encrypting the *Indication* with the server's static public key. However, he concludes that this method lacks forward secrecy, leading to the decision not to pursue it further. Another proposed idea is the use of a dynamic, ephemeral user identity for each new connection, previously agreed upon in an earlier interaction. This, however, fails to provide a solution for bootstrapping the very first connection between peers, making it impractical for production. An alternative strategy involves delaying the transmission of the user's identity until after the first RTT, when a shared secret is established, and the handshake messages are encrypted. While this is a valid approach, it fundamentally alters the TLS 1.3 handshake to an extent that it becomes incompatible with the TLS extension mechanism. This would require a restructuring of the TLS state machine to introduce an additional RTT, conflicting with the requirement for Modularity. Although this strategy is not used in this thesis, a detailed discussion of the idea is presented in Section 6.3. Finally, Breitkopf suggests a double TLS handshake as a potential solution. This concept is illustrated in Figure 7.

4.4.3. Double Handshake

The rationale behind the double handshake approach is tailored to circumvent the confidentiality limitations of the *ClientHello* message. Recognizing that achieving confidentiality at this point is not feasible, this method shifts the transmission of the user's identity to a preceding TLS handshake. In this initial handshake, the sensitive information is securely embedded within a *Response* message, thus ensuring its protection. The subsequent handshake then references this earlier exchange in its non-protected *Indication* message. The server temporarily stores all information received in the first handshake and later accesses it using the reference provided in the second handshake. In order to introduce an unambiguous terminology, the first handshake's *Indication*, *Request*, and *Response* messages will be termed *PreIndication*, *PreRequest*, and *PreResponse*.

The concept of a double handshake introduces a mechanism for securely transferring the user's identity to the server, all while preserving privacy from potential eavesdroppers. This process unfolds as follows:

1. **PreIndication**: The client initiates the process by sending a *PreIndication* message.

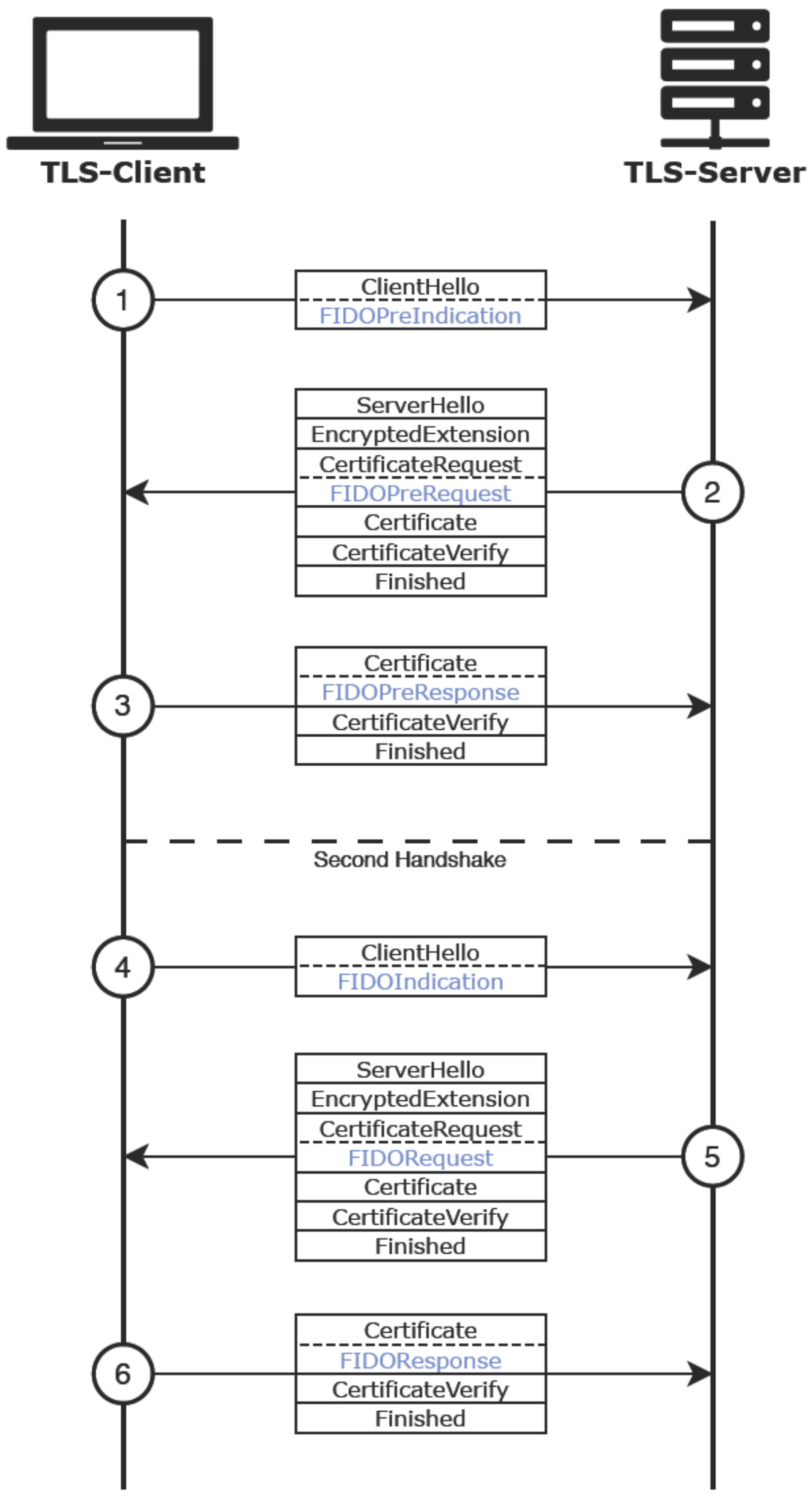

Figure 7: Two Consecutive TLS 1.3 Handshakes, carrying FIDO messages as Extension Data. Graphic created by author.

This message signals the client's intention to either authenticate with or register a new FIDO key. Given that this initial communication does not contain sensitive information, the lack of encryption at this stage does not pose a security risk.

2. **PreRequest**: Acknowledging the client's initial request, the server sends back a *PreRequest* message. This message includes a unique, ephemeral reference that the client will use in the subsequent handshake.

3. **PreResponse**: The client sends its user identity within the *PreResponse* message, protected from potential eavesdropping.

4. **Indication**: In the second handshake, the client sends the previously provided ephemeral reference. This reference links the current handshake with the previously established context. Like all *Indication* messages, no confidentiality is provided here; however, this does not compromise security as the ephemeral reference alone gives no insight into the user's identity to an adversary.

5. **Request**: Utilizing the ephemeral reference to retrieve the cached identity information, the server constructs a *Request* packet. This packet is created according to the requirements of FIDO's authentication or registration ceremony.

6. **Response**: Finally, the client responds with a *Response* packet that, again, aligns with the FIDO ceremony for either registration or authentication.

While this procedure appears secure at a glance, it inadvertently introduces another vulnerability. During step 4, an adversary could execute a MITM attack, intercepting and replaying the *Indication* message that contains the ephemeral reference. This action would enable them to subsequently obtain the server's *Response* message. The ephemeral reference used by the client does not reveal the user's identity directly to the adversary. However, once the server interprets this reference to generate the response, the message it sends back inherently contains information specific to the user. This could include, but is not limited to, credential IDs of previously enrolled credentials, details regarding the authenticator, or the client's username in plaintext. Thus, while the adversary does not initially know the client's identity, the server's response inadvertently exposes sensitive client information.

In order to address this design flaw, an additional security measure can be implemented within the double handshake. Specifically, the server can employ symmetric encryption to safeguard all sensitive information included in its *Request* message. The key material required for this encryption is exchanged during the first handshake. It is sent from the server to the client within the *PreRequest* message, alongside the ephemeral reference. This key is uniquely tied to the reference, making it ephemeral as well. It is used only once: To encrypt the sensitive data in the server's *Request* message, and subsequently, to decrypt this information at the client side. Using this additional security measure, the revised sequence of operations is as follows:

1. **PreIndication**: The client initiates the process by sending a *PreIndication* message, signaling its intention to authenticate with or register a new FIDO key. Since this initial communication does not contain sensitive information, the absence of encryption at this stage poses no security risk.

2. **PreRequest**: In acknowledgment of the client's initial request, the server sends back a *PreRequest* message, which is confidentially transmitted. This message includes a unique, ephemeral reference and a symmetric encryption key. The client stores both the reference and the key for use in the subsequent handshake.

3. **PreResponse**: The client transmits its user identity within the *PreResponse* message, protected from potential eavesdropping.

4. **Indication**: During the second handshake, the client sends the previously received ephemeral reference. This reference serves to link the current handshake to the previously established context. As with all *Indication* messages, no confidentiality is provided here, yet this poses no security risk since the ephemeral reference itself does not reveal any identifiable information about the user.

5. **Request**: Using the ephemeral reference to access the stored identity information, the server constructs a *Request* message. If the message contains sensitive, client-related data, then it is encrypted using the previously shared symmetric key. The content of the *Request* message adheres to the requirements of FIDO's authentication or registration ceremony.

6. **Response**: The client decrypts the encrypted content of the *Request* packet using the symmetric key, then sends back a *Response* packet that conforms to the FIDO ceremony for either registration or authentication.

In determining the appropriate symmetric encryption algorithm for the double handshake strategy, it is essential to select a protocol that ensures both security and compliance with established TLS standards. Advanced Encryption Standard (AES) Galois/Counter Mode (GCM) emerges as a highly suitable choice due to several key advantages it offers, which align well with the requirements of this extension. This choice is particularly advantageous because AES GCM operates without padding, thereby minimizing transmission overhead. Most importantly, the choice of AES GCM is strongly supported by compliance requirements. According to RFC 8446, a "TLS-compliant application must implement the `TLS_AES_128_GCM_SHA256` cipher suite" [21, Section 9.1]. This mandate ensures that by selecting AES GCM, the proposed security mechanism adheres to the standards required for modern TLS implementations, guaranteeing that our encryption approach is not only robust but also standardized across platforms that adhere to TLS specifications.

Typically, AES GCM requires a random Initialization Vector (IV) for each new ciphertext to ensure cryptographic security. However, in this specific application, each GCM key is ephemeral and designed for a single use only. This approach allows for the IV to be constant for each encryption process, as the key itself changes with every new encryption. This eliminates the need for a random IV for each operation, thus simplifying the encryption process and reducing overhead. While it is technically feasible to omit transmitting the constant IV along with the ciphertext, to maintain clarity and explicit cryptographic practice, it may be beneficial to include it. Transmitting the IV explicitly ensures that the encryption process remains transparent and verifiable, adhering to best practices in cryptographic implementations.

4.5. Control Mechanisms for Key Registration

The original FIDO registration ceremony lacks control mechanisms to determine which users are permitted to register a new FIDO key with the RP. As explained in Section 2.2.1, the outer web context typically conducts a preliminary authentication step, acting as a gatekeeper for new key enrollments. However, since FIDO registration on the transport layer lacks this outer web context, incorporating a control mechanism into the registration ceremony becomes necessary. While this might initially appear as a chicken-and-egg problem—requiring authentication to register keys that provide authentication—several concepts exist to address this issue.

One approach is to delegate responsibility to TLS. TLS already supports various client authentication methods (such as CCA, Kerberos, TLS-Secure Remote Password (SRP), TLS-OpenID, TLS-OAuth), which, as outlined earlier, are not widely adopted due to their individual difficulties. Despite the limited adoption of these methods, using them to authenticate a user during a new FIDO key registration could prove beneficial. With an authenticated client, the RP can then decide, based on its configuration, whether the client is authorized to register a FIDO key with the service. Alternatively, both peers may utilize a shared secret such as an OTP, PSK, or "ticket" known prior to the registration ceremony. The distribution of this shared secret can occur out of band or through various key distribution mechanisms, utilizing the same physical connection between the client and server as TLS [25, Section 7.3].

Initially, TLS-PSK was considered as a solution. This mechanism anchors the shared secret in the TLS protocol and would align with the standardization requirement. However, this idea posed difficulties because the PSK mechanism is also incorporated into TLS through an extension. Attempting to develop a TLS extension that builds upon another TLS extension resulted in a cross-extension dependency that was practically infeasible [21, Section 4.1.1]. Consequently, this thesis will adopt the concept of a ticket, assumed to be distributed to the peers prior to the registration ceremony out of band. During the handshake, the client sends its ticket to the server in the encrypted *Pre-Registration Response*. The server compares it to its own list of tickets, proceeding with the registration ceremony if a match is found, or aborting otherwise.

Tickets can be generated randomly or may contain semantic meaning, combined with sufficient entropy to prevent forgery. Examples of semantic information:

- **User Name or User ID:** Specifies the identity of the user associated with the ticket, ensuring that this identity is securely bound to the FIDO identity created during the registration process.

- **Expiration Date:** Indicates the date and time until which the ticket remains valid. This helps enforce time-bound constraints on the registration ceremony.
- **Issuer Information:** Provides details about the entity or system that issued the ticket, allowing the server to verify the authenticity and trustworthiness of the ticket.
- **Purpose or Usage Context:** Describes the intended purpose or context for which the FIDO keys can be enrolled using the ticket, ensuring that they are used appropriately.
- **Authenticator Properties:** Specifies the properties that the FIDO authenticator must fulfill for successful key enrollment. These properties may include the authenticator type, capabilities, and supported authentication methods. Verification of these properties occurs during the attestation step of the registration ceremony.

4.6. Message Structure and Encoding

Having outlined the integration of FIDO messages into the TLS handshake as extension data in Section 4.4, this section focuses on the message structure and encoding, specifically those exchanged between the client and the RP. The structure of messages between the client and the authenticator has already been specified by the CTAP. TLS extension data is inherently binary, allowing for a variety of encoding schemes. To determine the most appropriate encoding method, it is useful to consider the encoding practices of the related protocols:

WebAuthn relies on binary serialization of JSON objects for communication within HTTP requests and responses. Although this method facilitates easy parsing and aids in debugging due to its readability, it is far from being space-efficient, as JSONs textual format, consisting of characters such as curly braces, square brackets, colons, and commas, inherently consumes more space compared to binary serialization formats [27, p.13].

In contrast, CTAP utilizes CBOR encoding for its transactions. CBOR is a schema-less binary serialization specification offering compact binary representation for self-describing data. It supports diverse data types such as numbers, strings, booleans, arrays, and maps, with a layout enabling efficient representation of complex data structures. CBORs design prioritizes resource efficiency, making it suitable for use in environments with memory and processor constraints [27, p.60-63] [5].

TLS, notably, adopts a custom binary encoding scheme. This scheme organizes data into fixed-length fields sequenced in a predetermined order, with variable-length data managed through an extra length field. Moreover, specific elements of TLS, like X.509 certificates, are formatted using Abstract Syntax Notation One (ASN.1), employing Distinguished Encoding Rules (DER) for the encoding process. ASN.1 supports a variety of data types, including integers, real numbers, strings, booleans, enumerations, unions, and lists, making it highly versatile for encoding diverse data structures. However, unlike CBOR, ASN.1 requires schemas to be predefined, which can add complexity and overhead to the encoding and decoding processes. Furthermore, while ASN.1 provides efficient space utilization through its encoding rules, it may not be as flexible or lightweight as CBOR in resource-constrained environments [27, p.23-26] [11].

The upper size limit for TLS extension data is critically important in the design and encoding of FIDO messages, as all such messages are encapsulated within TLS extension data fields during the handshake. The maximum size for a TLS record, according to the TLS 1.3 specification, is 2^{14} bytes (16 kilobytes) [21, Section 5.1], encompassing the entire payload, headers, and other data within the record. Consequently, the actual space available for the sum of all extension data is slightly less than 2^{14} bytes, considering the space occupied by other elements of the handshake message. If a TLS message exceeds the 16 kilobytes limit of the record layer, TLS automatically fragments it across multiple records [21, Section 5.1]. Thus, in theory, there is no upper size constraint for extension data. However, to ensure robustness and efficiency, this thesis aims to avoid scenarios where TLS must fragment records due to long extension data. It is important to note that there are instances where TLS already fragments handshake messages, even without extensions. For example, the *Certificate* message can contain a potentially very long chain of Rivest Shamir Adleman (RSA) certificates with long keys, which can exceed the 16 kilobytes limit. Apart from the *Certificate* message, TLS handshake messages usually do not exceed 512 bytes. Therefore, if fragmentation is not desired, it is estimated that the available space for the sum of all extension data is approximately $2^{14} - 512 = 15872$ bytes.

In web-based FIDO, data exchanged between peers is serialized using JSON, which exhibits significant space inefficiency due to its verbose textual format and key-value structure. These attributes present opportunities for optimization. One potential improvement could involve organizing all data fields, both required and optional, in a predetermined, sequential format, thereby eliminating the necessity for keys. Optional fields should also follow this sequence, and their presence can be compactly encoded using a single byte, where each bit clearly indicates whether a specific optional field is present. While this approach significantly reduces packet size, it requires a custom encoding scheme. Such a custom scheme is difficult to parse and extend due to its reliance on precise bit patterns. Additionally, the complex nature of this custom encoding makes the implementation error-prone, complicating both development and future modifications.

Given the size constraints, the requirement for standardization and the complexity associated with a custom binary encoding scheme, this thesis adopts a hybrid encoding strategy, choosing CBOR over alternatives such as ASN.1 and proprietary binary formats. The preference for CBOR stems from its efficiency, compactness, and inherent schema-less, self-descriptive properties. Furthermore, the use of CBOR aligns with its established use within the FIDO protocol, ensuring uniformity in data encoding across components and eliminating the need to introduce novel encoding mechanisms.

The proposed extension suggests encoding required data fields in a predefined sequence within a CBOR array, removing the necessity for keys. Optional fields are encoded in a CBOR map, which facilitates presence checks through simple key lookups. Each key's overhead is minimized to a one-byte identifier by assigning a unique enum value to each optional field, avoiding the space-consuming verbose text strings typically used for keys. This method substantially reduces the storage space required for keys while preserving the flexibility and accessibility advantages of the CBOR map structure. CBOR inherently embeds the size and data type of each element within its encoding, thus eliminating the need for explicit length fields. Some optional data elements are grouped into tuples. If present, these tuples are stored in a sub-array, preserving their relational structure. Furthermore, certain optional tuples may exhibit zero or multiple occurrences, in which case they are organized into nested CBOR arrays. Enums are frequently used within the specification to represent specific choices or states. As CBOR does not natively support enum types, these are encoded as integers. CBOR automatically uses tiny integers for integer values between 0 and 23, encoding them in a single byte to minimize the data size.

Based on the analysis of message structure and encoding strategies detailed in this section, specifications for each message type are precisely defined in Appendix A. Messages are generally structured to adhere to the upper size constraints imposed by the TLS record layer (without fragmentation), with notable exceptions being messages A.5, A.11, and A.12. These messages have the potential to exceed the maximum record size due to their inclusion of extensive lists of `excludeCredentials`, `allowCredentials`, or custom FIDO extensions. The FIDO protocol, in general, does not specify limits on the number of these elements. A detailed examination of how the number of allowable list entries in specific message types impacts TLS record fragmentation will be further explored in Section 6.2 of this thesis.

4.7. Determination of Origin and Relying Party ID

In the FIDO specification, the origin is a security-critical concept that ensures the authentication request comes from and returns to a trusted source. The origin typically includes the scheme (e.g., https), the host or domain (e.g., example.com), and, implicitly, the standard port for the scheme [28]. It is used to ensure that authentication requests and responses are tied to the specific website or web application, thereby preventing phishing and MITM attacks by ensuring that FIDO credentials are not shared across different origins. In the WebAuthn process, the origin is obtained from the secure context of HTTPS, thus HTTPS is responsible for validating this origin before FIDO initiates the authentication request. In a FIDO-TLS extension, where the HTTPS context is absent, the concept of origin must be adapted to fit a non-web environment, thereby falling back to the underlying TLS mechanism.

In TLS, the server domain is typically identified by either the Common Name (CN) or the Subject Alternative Name (SAN) field of its certificate. The CN field historically served as the primary means of specifying the server's domain. However, due to security concerns and the evolving nature of internet standards, its usage has been deprecated in favour of the SAN field [20, Section 3.1]. The client determines its intended connection destination by specifying either a domain name or an IP address. Additionally, this intended endpoint can be further refined using Server Name Indication (SNI), an extension to the TLS protocol. The SNI allows the client to indicate the hostname at the beginning of the TLS handshake. This feature proves particularly valuable for servers hosting multiple domains (virtual hosts) under a single IP address, as it enables the server to present the correct certificate for the requested domain, enhancing the security and compatibility of the connection process. RFC 6066 strongly recommends the use of SNI, particularly when clients connect via Domain Name System (DNS) names [3, Section 3]. To ensure that the client's target endpoint matches the server's verified identity, TLS authenticates the server by comparing the SNI (or the hostname, if SNI is unavailable) with the DNS domains or wildcard entries listed in the SAN field (or, if the SAN field is missing, the CN field).

In a non-web context where FIDO is integrated within TLS but lacks the traditional HTTPS context, the SNI proves to be a viable candidate for the domain component of the origin. By utilizing the SNI to define the domain part, the origin can be effectively constructed to ensure that FIDO authentication and key registration are securely anchored to a specific server identity. If the SNI is not available, the origin could potentially fall back to the hostname or IP address of the underlying connection. Nevertheless, since the use of SNI is explicitly advocated for in RFC 6066, the proposed extension will focus on utilizing the robust definition provided by the SNI.

The construction of the origin would typically combine the SNI-derived domain with a predetermined scheme. In traditional FIDO contexts, the `https://` scheme specifically indicates the application layer protocol in use. However, in a pure TLS context where HTTPS is not employed, the concept of a "scheme" does not inherently apply because TLS functions at the transport layer, securing data transmission without specifying the application protocol. One could argue for the adoption of the conventional `https://` prefix in the origin definition, even when the underlying protocol is not HTTP. This approach could offer several advantages, particularly in maintaining compatibility with existing FIDO infrastructure and allowing a smoother integration with systems that expect traditional web origins. However, this adaptation must be carefully considered against the backdrop of protocol purity and the semantic implications. It would be essential to ensure that such a usage does not mislead system components or administrators about the underlying communication protocols in use. The decision whether to adopt the `https://` prefix in non-HTTP TLS contexts for compatibility is **intentionally left open for discussion**.

As mentioned in Section 2.2.2, the RP ID for a WebAuthn operation is set to the origin's effective domain. This default can be overridden by the RP, as long as its value is a registrable domain suffix of or is equal to the caller's origin's effective domain [31, Section 3]. The proposed extension preserves the concept of allowing the RP ID to override the origin; it merely adapts the method by which the origin is derived to accommodate environments where the traditional HTTPS context is absent.

4.8. The Finished Message

As outlined in Section 4.4, the FIDO key registration and authentication ceremonies both conclude with a *Finish* message. Given the constraints of TLS 1.3 extensions, which prohibit a full two-round trip message exchange, this section will explore an effective alternative for communicating essential outcomes of these processes. The *Finish* messages of registration or authentication ceremony contain the following data:

- **Key Registration *Finish* Packet:**
  - **Attestation Object**: Typically includes attestation data related to the registration process, such as signature, X.509 certificate chain, and the format of the attestation.
  - **Authenticator Data**: Provides details about the authenticator and registration process, such as credential ID, public key, signature counter, and flags indicating e.g. user presence, user verification.
  - **Client Data**: Encodes information such as the challenge, origin, and type of the client data involved in the registration process.
  - **Device Information**: Includes details about the device used for registration, such as its name, type, and certifications.
  - **Certificate**: Provides the certificate in PEM format.
  - **Status**: Indicates the status of the registration process.
- **Authentication *Finish* Packet:**
  - **Authentication Data**: Contains details about the authenticator and authentication process, including credential ID, public key, signature counter, user presence, and user verification flags.
  - **Client Data**: Includes information such as the challenge, origin, and type of the client data involved in the authentication process.
  - **Username**: Indicates the username associated with the authentication process.
  - **Status**: Indicates the status of the authentication process.

Most of the data in the *Finish* messages—such as the attestation object, authenticator data, client data, device information, username or certificate—is already known to the client. This data originates from the authenticator, passes through the client, and is sent to the RP, primarily in CBOR encoding. The RP's role is to decode this information, verify it, and typically send back a JSON-parsed version upon successful verification. However, this retransmission of already known data from the RP back to the client is not strictly necessary. The client, being informed of the ceremony's status, can independently parse and utilize the data without needing it echoed back by the RP in a JSON format. Given this understanding, the essential information that needs to be communicated can be condensed to just the status of the registration or authentication process.

4.8.1. TLS Alerts

Since transmitting known data back to the client is unnecessary, it is possible to avoid sending the *Finish* message by directly using TLS alerts to convey the essential status of the FIDO ceremony. This approach utilizes the existing TLS alert mechanism, traditionally employed to signal various session states, to efficiently communicate a range of possible statuses. This strategy avoids the need for extra messages during the handshake process.

While TLS itself can be extended via extensions, its alert system unfortunately cannot be directly extended to include new alert types without altering the base TLS implementation. To adhere to the requirement of not modifying existing TLS implementations, existing TLS alert codes can be repurposed to signal the outcome of FIDO operations. RFC 8446 defines TLS 1.3 alerts as follows [21, Section 6]:

```
enum {
    close_notify(0),
    unexpected_message(10),
    bad_record_mac(20),
    record_overflow(22),
    handshake_failure(40),
    bad_certificate(42),
    unsupported_certificate(43),
    certificate_revoked(44),
    certificate_expired(45),
    certificate_unknown(46),
    illegal_parameter(47),
    unknown_ca(48),
    access_denied(49),
    decode_error(50),
    decrypt_error(51),
    protocol_version(70),
    insufficient_security(71),
    internal_error(80),
    inappropriate_fallback(86),
    user_canceled(90),
    missing_extension(109),
    unsupported_extension(110),
    unrecognized_name(112),
    bad_certificate_status_response(113),
    unknown_psk_identity(115),
    certificate_required(116),
    no_application_protocol(120),
    (255)
} AlertDescription;
```

While the CTAP has a huge amount of very specific errors, the actual errors being communicated to the RP are abstracted for privacy reasons. When an error occurs during an authentication event such as a fingerprint validation error, the authenticator provides detailed feedback to the client. However, when relaying this information to the RP, the errors are typically generalized. In the context of WebAuthn, FIDO uses standard `DOMException` error names to communicate these errors [29, Section 2.8.1]. These names are well-integrated within the web API ecosystem but are less meaningful in a non-web context where browser-based handling is irrelevant. Therefore, to maintain consistency with the underlying TLS protocol and simplify the error handling process, `DOMException` names are omitted in favor of utilizing the TLS alert mechanism. The following typical outcomes of FIDO operations are defined, maintaining a balance between informative and privacy-preserving error reporting:

- **Success**: The operation (authentication or registration) completed successfully.
- **Invalid Credentials**: The credentials provided were incorrect or not recognized.
- **Device Incompatibility**: The device used is not compatible with the requested operation.
- **Timeout**: The operation timed out due to user inactivity or device response delays.
- **Internal Error**: An unspecified error occurred within the authenticator or the TLS integration.

To align FIDO outcomes with TLS alert codes, the following mappings can be established:

- **Success**: No alert is necessary; normal operation continuation.
- **Invalid Credentials**: Mapped to `access_denied (49)`, indicating that the provided credentials do not grant access.
- **Device Incompatibility**: Mapped to `illegal_parameter (47)`, repurposed to indicate unsupported device characteristics.
- **Timeout**: Mapped to `user_canceled (90)`, this alert signifies that the operation ended due to a timeout. This could be because data processing took too long or required user action wasn't completed promptly.
- **Internal Error**: Mapped to `internal_error (80)`, used for reporting unspecified system failures.

4.8.2. Communication to the Application Layer

TLS alerts effectively communicate the outcome of the FIDO ceremony to the application layer by signaling the status of the authentication or registration processes. These alerts are inherently designed to be propagated up through the network stack. While TLS alerts provide a concise indication of status, the application layer often requires access to more detailed information than just the outcome. In order to fully replicate the functionality of traditional FIDO implementations, the application must gain access to the full content of the original *Finished* messages. This ensures that all relevant data, beyond just the outcome status, is available for processing and utilization within the application layer.

In practice, this might involve extending the TLS library with specific API functions that allow applications to query the TLS layer for items such as user names, credential IDs, public keys, and user verification details. These APIs would provide a standardized method for accessing FIDO session data, just like the application would normally query the TLS layer for information about the TLS sessions.

With the proposed extension, the RP no longer echoes parsed data from the authenticator's CBOR-encoded messages back to the client. This change significantly alters the client's responsibility in handling CBOR-encoded data. Traditionally, the client merely acts as a conduit, forwarding CBOR-encoded data to the server without needing to decode it. However, with this new approach, the client must now parse the CBOR blob coming from the authenticator to provide the application layer with the necessary data. According to the WebAuthn specification, in traditional FIDO implementations, the client is not required to perform CBOR operations. Instead, it relies on the server to handle the decoding process. Consequently, with the proposed extension, there is an implicit requirement for the client to be capable of CBOR decoding. With the extension already employing CBOR encoding and decoding for all messages transmitted between the client and the RP, the necessity for the client to handle CBOR decoding is inherently satisfied, without the need for additional modifications.

5. Implementation

This chapter details the practical execution of the concepts proposed in earlier sections through two implementations, both developed in C code. The first implementation presents a PoC that demonstrates the feasibility of embedding FIDO directly within the TLS 1.3 protocol. This PoC serves to validate the theoretical models discussed previously, showing that the proposed TLS extension can function within the defined constraints and capabilities. The second part of the implementation applies the FIDO2 TLS 1.3 Extension within an EAP-TLS framework, illustrating its application in real-world scenarios such as Wi-Fi authentication.

Both implementations aim to showcase the realization of the concept more than the creation of a secure implementation ready for broad deployment. This emphasis helps to highlight the extension's potential to enhance protocol functionality, while also recognizing that further refinements are necessary for production-level deployment.

5.1. FIDO2 TLS1.3 Extension

5.1.1. TLS Library

When extending the TLS protocol, the choice of an appropriate TLS library is fundamental. Various libraries like `OpenSSL`, `WolfSSL`, `GnuTLS`, and forks of `OpenSSL` such as `LibreSSL` and `BoringSSL`, offer different functionalities and have their unique strengths. All these libraries are primarily written in C, a language chosen for its efficiency and control over system resources, which is crucial in high-performance environments required by cryptographic operations. While TLS libraries exist in many programming languages, the need for high performance makes a low-level language an essential choice. Initially, there was interest in implementing the extension using a TLS library written in Rust. However, the absence of a widely used application capable of showcasing the extension's use in Wi-Fi authentication, particularly one written in Rust, led to the decision to use a more established library. Thus, `OpenSSL`, with its extensive documentation, wide adoption, and robust integration with other popular software and libraries, was chosen. At the time of this writing, the latest version available is `OpenSSL 3.2.1`.

5.1.2. Structure

This subsection outlines the structure of the implementation, providing an overview of the system's components and their interactions. The PoC is designed for use with external FIDO hardware tokens via USB. For this purpose, the implementation uses `libfido2`, a C library that enables communication with FIDO authenticators and supports essential user authentication and credential management operations. Notably, `libfido2` provides an API that implements CTAP, facilitating direct communication between clients and authenticators through CTAP operations.

There are multiple approaches to implementing the FIDO backend on the server side. The proposed specification deliberately leaves the backend implementation open-ended, allowing for different strategies that may better suit specific requirements or use cases.

One possible approach is to have a dedicated FIDO server that either resides on the same machine as the TLS server or on a separate machine. This setup enhances modularity and allows the FIDO functionalities to be scaled independently from the TLS server. It also facilitates maintenance and updates without impacting the TLS operations directly. However, this approach may introduce additional latency. Latency arises from inter-process communication when both servers are on the same machine, or from network communication when the servers are separate. Secure configurations are required to protect the data, whether in transit or during process exchanges. Another approach is to integrate the FIDO server functionalities directly within the TLS server. This method simplifies the system architecture by reducing the components involved and eliminating the need for inter-server communication, which can lower latency and improve data flow. While this integration can make the system easier to manage and potentially more secure, as it reduces the number of points vulnerable to attack, it also concentrates the workload on a single server, which may affect scalability and performance under high demand.

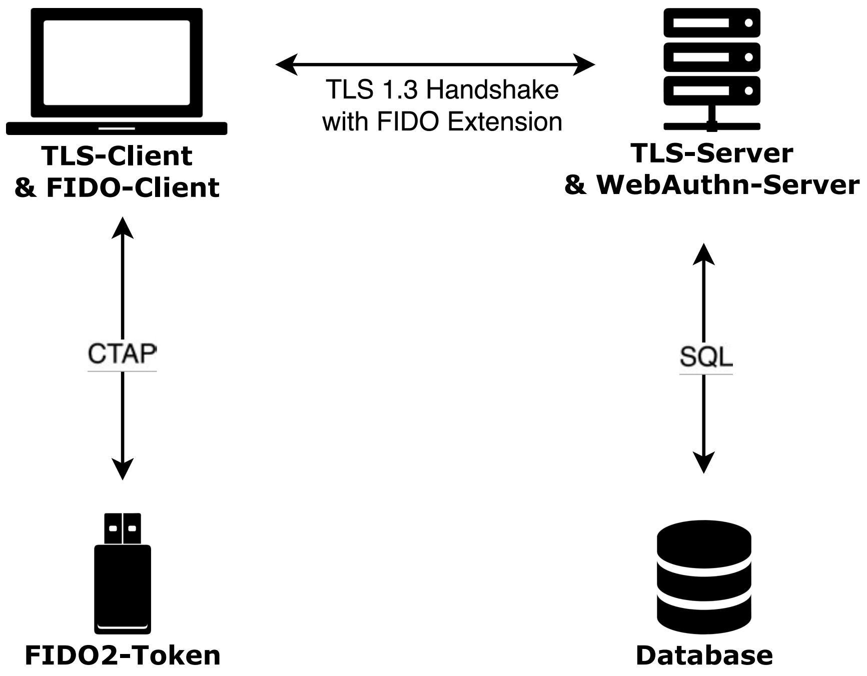

Figure 8: Structure of the FIDO2 TLS 1.3 Extension. Graphic created by author.

Both integration strategies—embedding the FIDO server functionality into the TLS server and maintaining it as a separate entity—are valid approaches for consideration. This implementation chooses to integrate the FIDO server functionality directly into the TLS server. This choice takes advantage of a simplified architecture and reduced latency, thus increasing the efficiency of the system. At the time of writing, there is no complete implementation of a FIDO server in C. However, `libfido2` offers API calls that can also be used on the server side, such as verifying a FIDO assertion. Nonetheless, custom development is necessary to handle the complete verification procedure, parse CBOR blobs generated by the authenticator, and to manage the storage of new FIDO credentials and user data.

For storage, this implementation utilizes a SQL backend, specifically `SQLite3`. `SQLite3` offers a lightweight and efficient solution for storing FIDO credentials and user data. Its simplicity and ease of integration make it well-suited for PoC implementations where a full-fledged database management system may be unnecessary. However, it is important to note that in production environments, the choice of backend database should be carefully evaluated based on factors such as scalability, reliability, and security requirements. Figure 8 illustrates the structure of the implemented extension.

5.1.3. Scope and Limitations

This section describes the boundaries and constraints under which the FIDO2 TLS 1.3 extension implementation operates. Given the largo scope of the FIDO2 specification, a focused approach was necessary to manage the workload in a feasible manner and focus on the novel concepts essential to adapting FIDO to a non-web environment. The implementation is limited to the following functionalities:

- **Key Registration**: The implementation supports the registration of only discoverable credentials. This process utilizes the double handshake approach as outlined in Section 4.4.3 to protect the user's identity by securely transmitting sensitive information in the second phase of the handshake. This method ensures that sensitive identity data is protected from potential eavesdroppers during the initial unencrypted exchange. Both TLS handshakes can be executed over the same Transmission Control Protocol (TCP) socket, thereby avoiding the overhead of a second TCP connection.

- **Authentication**: The system supports authentication using discoverable credentials, utilizing the single handshake approach as detailed in Section 4.4.1. This approach is chosen because no sensitive identity information is transmitted during authentication with discoverable credentials, minimizing the complexity and ensuring security. Currently, the implementation does not support non-discoverable credentials; however, the architecture is designed to be extendable. Adding support for non-discoverable credentials is feasible using the double handshake approach described in Section 4.4.3, which allows for the secure transmission of identity information.

- **Key Registration Control**: Key registration is regulated by a shared secret, specifically a ticket as outlined in Section 4.5. Both the client and server are pre-configured with this ticket. The ticket may possess semantic meaning. However, the server does not implement mechanisms to validate this semantic content. In the current implementation, the server can only be configured with a single ticket. Additionally, the server does not invalidate the ticket after it has been used. For practical use, particularly to support multiple clients, the server should ideally manage a list of tickets. Each ticket should be invalidated after use. This configuration would allow the registration of several clients, with each authenticated through their own individual ticket.

- **Concurrency Limitations**: The TLS extension does not support multiple concurrent FIDO authentication or key registration processes within the same TLS session.
- **Attestation**: Attestation during key enrollment is not implemented. Instead, a TOFU approach is adopted, wherein the server trusts the first contact it has with an authenticator.
- **FIDO Extensions**: The implementation does not support FIDO extensions, focusing instead on core functionalities.
- **Authenticator Support**: The client only supports external USB hardware tokens and does not integrate with platform authenticators.
- **Cryptography Support**: On the server side, only the most prevalent public key algorithm, *ES256* (ECDSA with SHA256 on a P-256 curve), is supported [23, Section 8.1] . However, the server is structured to allow easy expansion to other algorithms by extending the parsing capabilities of CBOR blobs.
- **Message Specification and Encoding**: Communication between the client and server adheres to the message specifications and encoding methods outlined in Section 4.6 and detailed in Appendix A. This ensures a consistent and efficient format for data transfer, utilizing CBOR for encoding the data, which aligns with FIDO specifications for compact and effective data representation.
- **Error Handling**: Errors occurring on the client side are pushed to `OpenSSL`'s error stack, allowing the client application to be informed about the specific type of error encountered. For communication with the server, only three generalized error types are transmitted via TLS alerts, as detailed in Section 4.8.1: `SSL_AD_ACCESS_DENIED`, `SSL_AD_INTERNAL_ERROR`, and the `SSL_AD_USER_CANCELED` error. This method restricts the detail of client-side errors exposed to the server while still providing necessary contextual information for security and operational purposes.
- **API Extensions for Success Outcomes**: While the implementation effectively handles error notifications through `OpenSSL`'s error stack and TLS alerts, it does not extend `OpenSSL`'s client API to provide additional information following the successful completion of a FIDO ceremony. Although theoretically feasible, this implementation does not include an API that would allow applications to query detailed results, such as user names or credential IDs, directly from the TLS layer after a successful authentication or registration event. This limitation acknowledges the potential for future enhancements that could include these capabilities to fully

align with traditional FIDO operations where such data is accessible via a finished message, as discussed in Section 4.8.2.

5.1.4. Registering the Extension

`OpenSSL` allows for the addition of custom TLS extensions via callback functions, which provide a flexible mechanism for developers to enhance the protocol without altering the core source code [18]. These callbacks are designated to handle specific tasks related to the lifecycle of an extension: Adding extension data to outgoing messages, parsing incoming extension data, and freeing any allocated resources. `OpenSSL` separates `SSL_CTX` objects from `SSL` objects. The `SSL_CTX` serves as a configuration template, while the `SSL` object, as an instance derived from a `SSL_CTX`, handles individual connections. This separation enables configurations to be reused across multiple TLS connections, optimizing resource management and simplifying the setup process. The custom callback functions, which encapsulate the logic of the TLS extension, are registered using the `SSL_CTX_add_custom_ext()` function. This registration links the callbacks directly to the `SSL_CTX` object. Consequently, any `SSL` object that is instantiated from this `SSL_CTX` will inherit the custom extension capabilities, allowing the defined logic to be executed as part of the TLS handshake process for every individual connection.

Each custom extension is identified by a unique extension type code. Official extensions use codes that are registered with the IANA. However, for this PoC, a non-official, custom code, `0x1234`, is used. This code falls within the range reserved for private use [12], ensuring it does not conflict with any officially registered extension codes. It is critical that both the client and server register this extension; if only one peer registers it, the TLS protocol mandates that unknown extensions be ignored, which would prevent the extension from functioning.

The entire extension is deployed as a static library. An application that wishes to utilize this extension links against this library and registers the provided callback functions to the `SSL_CTX` object. This approach ensures that the custom extension logic is seamlessly integrated into the application's existing TLS infrastructure.

5.1.5. Configuring the Extension

Configuration of the extension is managed through an options table provided during the registration of the callback function. This table enables specific FIDO configurations to be set, tailoring the extension's behaviour to meet particular operational requirements. The central configuration on the client side is the `mode` parameter, which is essential as it dictates the primary function of the operation—either registering a new user or authenticating an existing one. Unlike the client, the server side does not have a `mode` parameter because it is always reactive; it adapts its behaviour based on the client's initial request, aligning with the mode specified by the client to facilitate the appropriate FIDO ceremony. In future implementations, the server could be configured with a policy that dictates which modes a client is permitted to execute. Each option listed below has a length constraint, with maximum lengths specified in Appendix A.

Client Options

- **mode**: Specifies the operation mode, which can be either `FIDOSSL_REGISTER` for registering a new user or `FIDOSSL_AUTHENTICATE` for authenticating an existing user. This parameter is mandatory as it defines the primary function of the operation.
- **user_name**: The username associated with the FIDO device. This field is mandatory for the key registration mode, but not for authentication with discoverable credentials. The value is a UTF-8 string.
- **user_display_name**: A displayable name for the user, primarily used for user-friendly UI displays. This parameter is optional and will default to the value of **user_name** if not provided. The value is a UTF-8 string.
- **ticket_b64**: A base64 encoded ticket, used specifically in registration mode to control key registration processes.
- **pin**: The Personal Identification Number for the authenticator. This is optional and is required only if the authenticator demands it for user verification or device unlocking.
- **debug_level**: An integer indicating the level of debugging information to output. This is optional.

Server Options

- **rp_id**: The identifier of the RP. This is optional; if not provided, it is derived from the origin.
- **rp_name**: The name of the RP, which is mandatory.
- **ticket_b64**: A base64 encoded ticket used by the server to control a new key registration. The ticket is not needed in authentication mode.
- **user_verification**: A policy enum indicating the level of user verification required (`REQUIRED`, `PREFERRED`, `DISCOURAGED`). Optional, with the default being `PREFERRED`.
- **resident_key**: A policy enum indicating the support or requirement for resident keys (`REQUIRED`, `PREFERRED`, `DISCOURAGED`). Must be set to `REQUIRED` as the PoC currently only implements discoverable credentials.
- **auth_attach**: Indicates the type of authenticator attachment used (`PLATFORM`, `CROSS_PLATFORM`). Must be set to `CROSS_PLATFORM` because the PoC currently does not support platform authenticators.
- **transport**: Specifies the transport medium (`USB`, `NFC`, `BLE`, etc.). Must be set to `USB` because the PoC currently only implements USB tokens.

- **timeout**: The maximum time in seconds allowed for the operation before timing out. This is optional.
- **debug_level**: An integer indicating the level of debugging information to output. This is optional.

5.1.6. Dependencies

The FIDO2 TLS 1.3 Extension was designed to operate with as few dependencies as possible to ease its integration and maintenance. However, the implementation ultimately requires five essential libraries. The following section details these dependencies and explains their roles in supporting the extension's functionality:

- **OpenSSL (libssl and libcrypto)**: As an extension for `OpenSSL`, the libraries of `OpenSSL` are inherently a dependency.
- **libfido2**: Essential for interfacing with FIDO devices, this library offers high-level APIs for WebAuthn and CTAP, facilitating communication with FIDO authenticators and handling credential management and user authentication.
- **tinycbor**: This library is used to encode and decode CBOR formatted data and is known for its very low resource footprint.
- **SQLite3**: Selected for its simplicity and lightweight footprint, `SQLite3` provides the database functionality for storing FIDO credentials and user data on the server side.
- **libjansson**: While most JSON dependencies typical of WebAuthn have been removed in this extension, JSON is still required for constructing and parsing the `ClientdataJSON`. This library facilitates these operations with JSON, although it is possible to manually build and parse the `ClientdataJSON`, potentially eliminating the need for this dependency.

5.1.7. Repository Access and Usage Instructions

The implementation of the FIDO2 TLS 1.3 extension is available in a public Git repository, which contains all necessary files and comprehensive instructions for incorporating the extension into C projects. The `README.md` file in the repository provides detailed instructions on configuration, building, installation, and integration of the extension. Additionally, it includes practical C code examples demonstrating usage. For testing the extension's functionality, refer to the dedicated test code provided in the `test/` directory within the repository.

Repository URL:	`https://github.com/tummetott/fidoSSL`
Commit Hash:	`8e38d849eb355d1d0f768620d1afce7f111c16e0`

This specific commit hash can be used to access the exact version of the code that was used and referred to in this document. By doing so, the content remains unchanged and secure against any future modifications, preserving the accuracy and reliability of the thesis.

5.2. EAP-TLS with FIDO2 TLS1.3 Extension

To demonstrate the practicality and versatility of the FIDO2 TLS 1.3 Extension, it has been implemented within an EAP-TLS framework, which is commonly used in wireless network setups such as *eduroam*. As outlined in Section 2.3.1, EAP-TLS traditionally utilizes TLS with client certificates to achieve mutual authentication. By incorporating the FIDO2 TLS 1.3 Extension, the traditional reliance on client certificates in EAP-TLS is replaced with FIDO authentication mechanisms. In this setup, the TLS handshake packets, which now include FIDO authentication messages as extension data, are encapsulated within the EAP request and response packets.

5.2.1. Software Packages

The implementation utilizes two of the most prominent software packages in wireless network management and security: `hostapd` and `wpa_supplicant`. `hostapd` is a user space daemon software enabling a network interface card to act as an *Access Point*. It allows for the configuration of various parameters related to Wi-Fi Access Points, including authentication mechanisms, encryption methods, and network settings. `hostapd` supports a wide range of security protocols, including WPA2 and WPA3, as well as authentication mechanisms such as 802.1X/EAP [14]. When using EAP for authentication, `hostapd` can be configured to utilize either an external RADIUS server or an internal one. `hostapd` is commonly used in scenarios where a Linux-based device needs to provide Wi-Fi access, such as setting up public Wi-Fi hotspots, enterprise Wi-Fi networks, or home Wi-Fi routers.

`wpa_supplicant` complements `hostapd` by operating as the *Station* on the client side, managing wireless connectivity and authentication and serving as the counterpart to `hostapd`'s role as an Access Point. Like `hostapd`, `wpa_supplicant` supports a broad array of security protocols and authentication mechanisms, including those necessary for engaging with EAP-based authentication frameworks like EAP-TLS [15]. While `hostapd` handles the authentication, association, and overall management of the network from the APs side, `wpa_supplicant` ensures that client devices can securely and effectively connect to these networks. It manages the setup and maintenance of the client's network connections, handles the negotiation of network authentication as specified by `hostapd`, and maintains the ongoing integrity and security of the connection. At the time of this writing, the latest available version of `hostapd` and `wpa_supplicant` is 2.10.

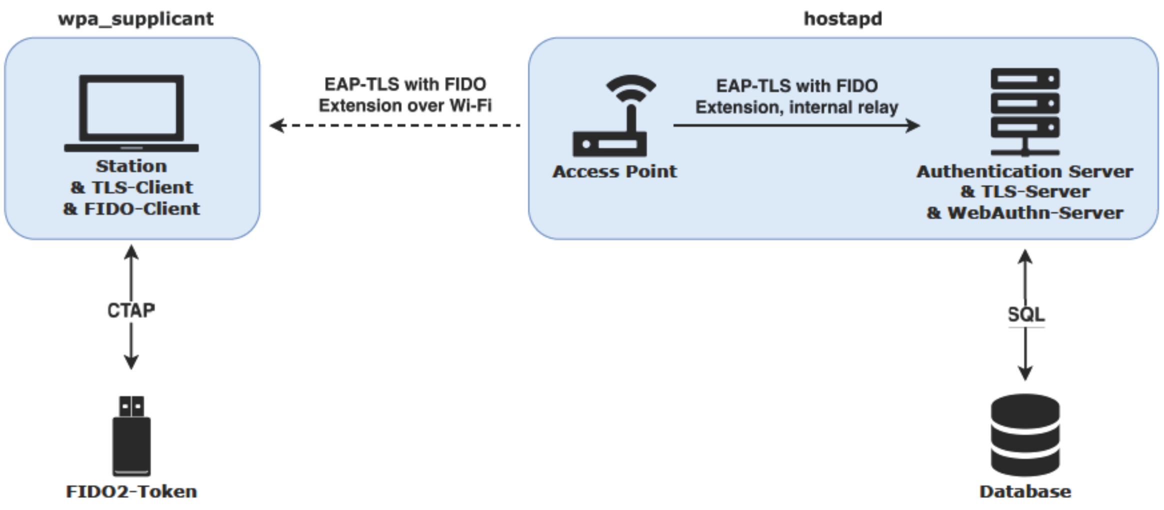

Figure 9: Structure of the EAP-TLS with FIDO2 TLS 1.3 Extension. Graphic created by author.

5.2.2. Structure

The structure of the Wi-Fi network using EAP-TLS with FIDO authentication is shown in Figure 9. The `wpa_supplicant` serves as both a TLS-client and a FIDO-client, interacting with an external FIDO hardware authenticator via CTAP. `hostapd` combines the functionalities of the Access Point and Authentication Server, which in turn includes the capabilities of both a TLS-Server and a WebAuthn Server. Internally, `hostapd` manages communications between the Access Point and the Authentication Server without using RADIUS, as both functionalities are integrated within the same process. The Authentication Server interfaces with a SQLite3 database through SQL queries to retrieve essential data, such as public keys and user information, crucial for validating FIDO credentials. The communication between the Station and the Access Point is performed over a Wi-Fi network. Following successful authentication, the Wi-Fi network employs either WPA2 or WPA3 security protocols. Depending on the specific setup, it uses CCMP (AES in CCM mode) or GCMP (AES in GCM mode) for encrypting the data payload, ensuring secure transmission over the air.

5.2.3. Key Registration

Within this implementation, `hostapd` is designed to authenticate clients using the FIDO2 TLS 1.3 extension and deliberately excludes key enrollment processes. Integrating FIDO key registration within the Wi-Fi connection sequence would deviate from standard network practices and is therefore avoided. Typically, the enrollment process for key material or certificates used in EAP methods is managed out-of-band and is not a component of the EAP protocol itself. Consequently, FIDO credential registration is more effectively handled in a controlled environment, such as a web interface. This approach facilitates a more interactive session, allowing users to register new keys without the pressures of immediate network access. For instance, in the case of *eduroam*, users could log into their institution's website, such as a university portal, to register new FIDO credentials with an authenticator. This ensures that once they attempt to connect to the network via Wi-Fi, their devices or external hardware tokens are already configured and ready for secure authentication using the previously enrolled credentials. This approach mirrors how client certificates are usually registered and managed outside of the direct Wi-Fi authentication process. For this PoC, keys can be registered by using the test code provided in the repository of the first implementation, as detailed in Section 5.1.7

5.2.4. Implementation Details

The PoC was designed to be "quick and dirty", implementing shortcuts that resulted in only minimal modifications to the existing codebase. The implementation is intentionally simple to emphasize that integrating the FIDO2 TLS 1.3 Extension into an established system like EAP-TLS can be achieved with relatively few changes. This approach demonstrates the practicality, yet it also highlights the potential for a cleaner and more refined implementation in future developments. For instance, on the client side, options such as the PIN for the FIDO token and the debug level are hardcoded. While this method effectively demonstrates the extension's feasibility, it is not intended for production use. In a more developed setup, these options could be made configurable through the `supplicant.conf` file. On the server side, however, the configuration is handled with greater care. FIDO options are not hardcoded but are instead made configurable through the `hostapd.conf` file, demonstrating how FIDO configurations could be integrated into existing config files.

Since EAP-TLS traditionally relies on client certificates, `wpa_supplicant` is programmed to return an error if no certificate is provided. In the PoC, this behavior was not modified, so a client certificate must still be presented to the supplicant. However, the server does not verify this certificate because FIDO authentication supersedes the need for client certificates. Ideally, in a fully developed implementation, `wpa_supplicant` should be adjusted so that presenting a client certificate—valid or not—is no longer necessary.

The proposed extension requires the use of TLS 1.3, which means that both `hostapd` and `wpa_supplicant` must be configured to use this protocol version during the handshake. Although current versions of these tools support TLS 1.3, it is not enabled by default and must be explicitly activated. This involves setting the compile-time configuration option `CONFIG_EAP_TLSV1_3=y` to ensure that TLS 1.3 capabilities are included in the binary. Additionally, TLS 1.3 must be explicitly enabled in the runtime configuration within the `hostapd.conf` file. The need for these multiple steps to activate TLS 1.3 suggests that its adoption in Wi-Fi networks is not very widespread at the time of writing.

5.2.5. Repository Access and Additional Resources

The second implementation is available in the following public git repository:

Repository URL:	`https://github.com/tummetott/hostap-fido2`
Commit Hash:	`b7a1eea00e00db292ea8f4f22598f3c0d1d8b864`

Instructions for establishing a local PKI and generating necessary certificates are provided in Appendix B. This appendix outlines the process for creating a test CA, as well as client and server certificates using `OpenSSL`. Configuration files for `hostapd` and `wpa_supplicant`, which include the settings necessary to test the implementation, are detailed in Appendix C. Additionally, Appendix D provides a convenience script that uses `tmux` to simplify simultaneous launching and restarting of client and server instances.

6. Evaluation & Future Work

The previous chapter demonstrated the practical feasibility of integrating FIDO authentication and registration within the TLS handshake, as outlined in Section 4. This chapter evaluates the proposed TLS extension, taking into account more than just its effectiveness. It also explores ideas for improving the integration and outlines directions for future research.

It has been shown that FIDO authentication using discoverable credentials can be efficiently implemented using the single handshake method. This approach does not extend the round trips required for TLS, maintaining a protocol flow similar to that of TLS with client certificates. However, for key registration or authentication using non-discoverable credentials, a single handshake would compromise the user's identity. To address this issue, a double TLS handshake method with symmetrically encrypted FIDO payload was proposed. Although this method doubles the original RTT and adds complexity, its impact might be less concerning given the anticipated decline in the relevance of non-discoverable credentials. Today's newer generations of FIDO authenticators are equipped to support discoverable credentials, thanks to enhanced storage capacities in hardware tokens and increased integration of platform authenticators in devices like laptops, smartphones, and tablets. Current FIDO tokens already possess enough storage to accommodate all the discoverable credentials an average user might need [35], and storage is expected to increase significantly according to Moore's Law. Therefore, non-discoverable credentials may become less relevant in the future. By offering a solution that is both effective and efficient for discoverable credentials, this extension is well-positioned to support the predominant method of FIDO authentication expected in the future. However, key registration remains a critical component that requires detailed evaluation, especially given its reliance on the less efficient double handshake method.

6.1. Key Registration

First, it is important to note that key registration, compared to authentication, is a less frequent process. Consequently, the efficiency of the enrollment process may not be as critical as that of authentication. Furthermore, registration does not necessarily need to occur within a TLS handshake. As discussed in Section 5.2.3, out-of-band methods, such as a traditional web interface, are often more suitable, particularly in scenarios where users register FIDO credentials for interactive use. A web interface provides greater flexibility and enhances user experience by interactively guiding users through setting up their security requirements. For example, it can dynamically inquire whether user presence checks or verifications are needed, or which type of biometrics the user wants to use.

Conversely, integrating FIDO key registration within TLS may be advantageous for non-interactive scenarios, such as with IoT devices. Upon initial activation, these devices can automatically perform a FIDO key registration with their cloud service via TLS, creating credentials that are stored, for example, on a platform authenticator. Since the security requirements are typically predefined by the device's application, user interaction may not be necessary. These devices could be pre-configured in the factory with a ticket that authenticates the key registration and includes semantic information defining the security parameters.

The PoC provided does not implement any semantic meaning of tickets; however, they could include various pieces of information such as the user name, expiration date, or issuer details, as discussed in Section 4.5. Instead of merely adding random bytes to each ticket to ensure sufficient entropy to prevent forgery, the semantic information could be directly signed or used to generate a MAC. This signature or MAC would be generated using a private key or secret known only to the issuing service. To verify a ticket, the service can use the corresponding public key (in the case of a signature) or the same secret key (in the case of a MAC). If the verification process confirms the ticket's authenticity and integrity, the service can trust the ticket's semantic information. This method ensures that any tampering with the ticket would be detectable, as the signature or MAC would not match if the content were altered. This approach has the advantage that the service does not need to be configured with a potentially very long list of valid tickets; it only requires holding a single secret or public/private key pair.

6.2. Message Size Evaluation

In Section 4.6, it was detailed that FIDO messages should not exceed the approximate upper limit of 15,872 bytes in order to avoid message fragmentation of the TLS record layer. It was also noted that messages of types A.5, A.11, and A.12 might potentially exceed this limit due to their variable list of credentials or custom extensions. To evaluate the impact of the size constraint on the variable parameters of these messages, a worst-case and average-case analysis of the message size will be conducted. This analysis will focus on the largest message type, type A.5, as size constraints are expected to be less stringent for the other packet types. While Appendix A already provides detailed upper and lower size limits for each component of a message, it omits consideration of how CBOR encoding influences the overall size of the message. To address this gap, Appendix E presents a comprehensive analysis of the maximum message size for message type A.5, accounting for CBORs overheads. The Appendix yields the following formula for for calculating the maximum message size:

$$\text{Max message size} = 1191 + N \times 261 + M \times 4358\,\text{bytes} \tag{1}$$

where N is the number of `excludeCredentials` and M is the number of possible FIDO extensions. Let's consider a practical scenario with $M = 2$, implying the use of two FIDO extensions. Substituting into the formula with the upper limit of 15,872 bytes, the permissible value of N, representing `excludeCredentials`, is calculated as follows:

$$15872 = 1191 + N \times 261 + 2 \times 4358 \tag{2}$$

Solving for N and rounding down to the nearest integer yields 22. It should be noted that this is a conservative estimate. It assumes that all elements within the message, such as credential IDs and user names, occupy their maximum potential sizes, which is rarely the case in typical implementations. Credential IDs of 256 bytes and user names of 255 characters represent extreme cases, not common usage scenarios. Therefore, in practical applications, the number of possible `excludeCredentials` could be significantly higher than this estimate.

Given the conservative estimates used previously, a more practical example involves assuming average field sizes, rather than maximal. For instance, if we consider the *Challenge* field size to be typically 32 bytes instead of the maximal 64 bytes, and the name fields (*RP Name*, *User Name*, *User Display Name*) are more commonly around 50 bytes each instead of 255, the dynamics of the message size change notably. Similarly, if the *Credential ID* within `excludeCredentials` averages 128 bytes rather than 256 bytes, and assuming the *Extension Data* is typically around 250 bytes, the formula for maximum message size becomes:

$$15872 = 544 + N \times 132 + M \times 512\,\text{bytes} \tag{3}$$

Solving this with $M = 2$, rounding down to the nearest integer results in N equal to 108. This more realistic average case scenario illustrates that during a key registration ceremony involving two FIDO extensions, the RP is still able to accommodate a list of more than 100 `excludCredentials`, without TLS fragmenting the message. However, such a large list is highly unlikely in practical scenarios. The `excludCredentials` list is primarily intended to prevent a user from inadvertently registering the same authenticator multiple times with the same RP. Typically, users register a credential for each of their hardware tokens or, in the case of platform authenticators, for each of their devices (e.g., phone, laptop, tablet). This generally results in only a handful of credentials being registered with a single RP. Therefore, the likelihood of the credential list exceeding ten entries is rather low. This calculation demonstrates that fragmentation of TLS handshake messages does not usually occur in practice with the proposed FIDO extension.

6.3. Modifying the TLS Library

One of the requirements of this work was to integrate FIDO into the TLS handshake only using the TLS extension mechanism. The core library of `OpenSSL` is not modified; instead, the extension is registered by the overlying application using `OpenSSL`'s official callback mechanism. This approach has the benefit of isolating the FIDO logic from TLS, thereby simplifying maintainability, auditing, and deployment. However, it has also led to some clearly unfavorable design decisions, such as reusing TLS alerts as discussed in Section 4.8.1 and the requirement for the server to send a `CertificateRequest` as detailed in 4.4.1. Eliminating the restriction of not modifying the TLS library would lead to a cleaner design. Firstly, TLS could be enhanced with specific FIDO alerts, thus eliminating the current ambiguity of some TLS alerts. Secondly, introducing new

TLS message types would be possible. Rather than performing two consecutive TLS handshakes for non-discoverable credentials or key registration, the TLS handshake could be substantially altered to safely accommodate the entire FIDO ceremony in a single handshake. A hypothetical, heavily modified TLS handshake integrating FIDO is illustrated in Figure 10. Like before, a dashed line indicates that a message is included as an extension of the message above.

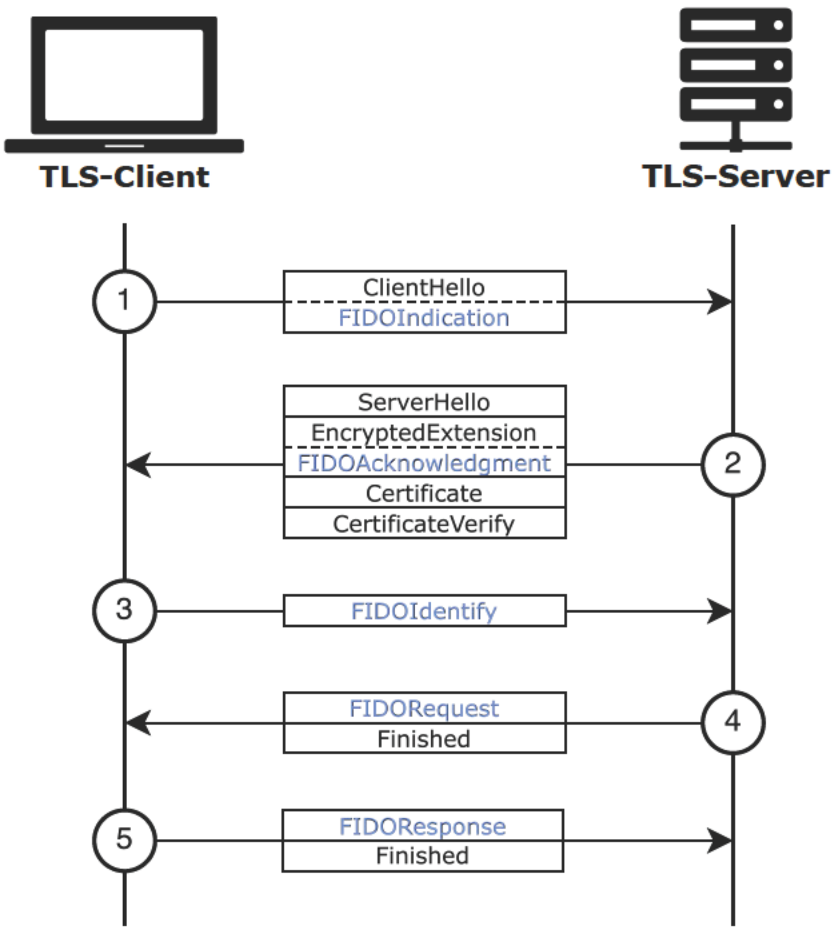

Figure 10: Hypothetical FIDO2 Integration into TLS 1.3. Graphic created by author.

1. The TLS-client sends a *FIDOIndication* extension with the *ClientHello* message. This indication specifies whether the client wishes to register or authenticate using discoverable or non-discoverable credentials. Although this message is not encrypted, it does not compromise security as it contains no sensitive information.

2. The server responds with a *FIDOAcknowledgment* within the *EncryptedExtensions*. This message confirms whether the server is capable and willing to use FIDO. As discussed in Section 4.4.2, this message is protected but could be intercepted by

a MITM attack. However, it carries no sensitive user information and therefore requires no additional protection.

3. For non-discoverable credentials or key registration, the client sends a newly introduced TLS handshake message, the *FIDOIdentity*. As this transmission occurs after a shared secret has been established between peers, the identity information and all following messages are encrypted and authenticated by TLS.

4. The server replies with a *FIDORequest* message, which includes either *PublicKeyCredentialCreationOptions* or *PublicKeyCredentialRequestOptions* depending on the type of FIDO operation. This message is sent along with the traditional TLS *Finished* message, which includes a hash of the entire handshake up to this point, including the newly introduced FIDO messages. This hash is then sent through the authenticated TLS channel.

5. The client then sends a *FIDOResponse*, according to the traditional FIDO protocol. This response is accompanied by the client's *Finished* message, which hashes the entire handshake from the client's perspective, including the new FIDO messages. In line with the TLS 1.3 protocol, encrypted application data may also be transmitted along with this packet.

The illustrated handshake effectively integrates the entire FIDO ceremony into TLS, eliminating the need for a double handshake. However, it also introduces an additional round-trip to the protocol for FIDO key registration and authentication with non-discoverable credentials. In the case of discoverable credentials, the server can combine the packets from steps 2 and 4, while the client omits sending the packet from step 3. This eliminates the additional round-trip, reducing it to the original TLS 1.3 sequence.

Since new TLS messages are introduced, the client's *Certificate* message is no longer required to carry the *FIDOResponse*. Consequently, the server does not need to send a *CertificateRequest*, eliminating the reliance on placeholder client certificates. The transmission of the client's identity is delayed until both parties have established a shared secret and is therefore protected. An adversary could still attempt a MITM attack; however, the server only discloses user-specific information after receiving the user's identity from the client. Since the adversary can not learn this ephemeral identity from a previous handshake, additional symmetric encryption as detailed in Section 4.4.3 is not required.

This integration of FIDO into TLS represents more than just an extension; it significantly amends the protocol by modifying the handshake process itself. While this modification extends beyond typical TLS extensions, it retains interoperability with TLS 1.3. The *FIDOIndication* is transmitted as an extension during the initial *ClientHello* message. If the server does not recognize this extension, it simply ignores it, allowing the client to decide whether to continue the TLS handshake without engaging in the new FIDO-specific message exchanges. Conversely, if the server does not receive a *FIDOIndication*, it may also choose to proceed with the handshake without involving FIDO functionalities. If either the client or server expects FIDO usage but does not receive the appropriate *FIDOIndication* or *FIDOAcknowledgment*, the peer in question can abort the handshake by issuing a *Missing Extension* alert (alert 109). This setup enables FIDO operations to be negotiated at the start of the handshake, similar to how cipher suites are negotiated.

While it is true that TLS 1.3 was designed to minimize handshake times, reducing them to a single RTT to improve performance in long-latency networks such as those encountered with mobile devices, the integration of FIDO into the TLS handshake appears at first to counter these advancements by potentially increasing the number of RTTs. Critics might view this as a step backward, given the potential hidden complexities and implications of modifying the handshake structure. However, it is important to consider the broader context of the whole process. Typically, when FIDO is used in conjunction with HTTPS, the process involves an initial TLS handshake RTT followed by an additional two RTTs for FIDO authentication conducted over HTTP. Therefore, by integrating FIDO directly into the TLS handshake, although we reintroduce an additional RTT into the TLS process, we potentially reduce the overall number of RTTs required for authentication. In other words, this approach adds complexity to the TLS protocol but reduces complexity at the application layer.

6.4. FIDO Authentication in Wi-FI Networks

Section 5.2 demonstrated how the FIDO2 TLS 1.3 extension could be used in actual applications such as Wi-Fi networks. However, this showcase is far from being truly practical. In a wireless environment like *eduroam*, the emphasis is often on high usability rather than high security requirements. The current PoC, however, requires users to physically connect an external FIDO USB token and actively participate in a user presence check. This level of interaction can detract from the user experience typically desired in such settings. To address this, integrating platform authenticators like TPMs or Apple's secure enclave could significantly enhance usability. These built-in authenticators would allow authentication to proceed without the need for external tokens. Additionally, adopting silent authentication, which eliminates the need for user interaction, would align with current Wi-Fi practices. This would make the authentication process completely automatic, closely mirroring the convenience of traditional authentication methods in Wi-Fi.

The EAP-FIDO draft, mentioned in Section 3.3, successfully integrated FIDO into Wi-Fi using a slightly different approach. Instead of performing the FIDO authentication ceremony during the TLS handshake, the ceremony was tunneled through an established TLS connection as "inner authentication". This method allows for a variable number of round-trips between the client and the RP, depending on the outcome of the FIDO authentication, the user presence or user verification assertions, and the policy for a specific FIDO credential. The RP may then choose to initiate a second FIDO authentication with a different set of authentication requirements. The PoC of this work, however, does not have this flexibility, as the number of round-trips is limited to one ceremony at a time. Nevertheless, the RP could abort the handshake and signal the client to start a new one with a new TLS connection.

7. Conclusion

The FIDO protocol provides a method for strong client authentication with high entropy, phishing resistance, high usability, and hardware security through external tokens. However, FIDO was developed for the web and is therefore rarely used in applications that do not build on HTTP. This thesis proposed a TLS 1.3 extension that integrates FIDO authentication and key registration into TLS at the transport layer, thereby decoupling it from its web-based constraints and extending the method's applicability to non-web-based applications.

TLS has proven to be an effective protocol for integrating FIDO. It protects FIDO ceremonies with strong encryption, ensuring the confidentiality and integrity of the data exchanged. Additionally, server authentication using X.509 certificates provides a robust mechanism for validating the identity of the RP. It was demonstrated that authentication with discoverable credentials can be integrated into the TLS handshake without increasing the RTT. Authentication with non-discoverable credentials and key registration can be integrated by performing two consecutive TLS handshakes over the same TCP connection. A ticket-based solution has been proposed for controlling FIDO key registration without preliminary web-based user authentication. The evaluation showed how tickets could be improved by adding semantic meaning and incorporating signatures or MACs for better scalability.

The thesis included an extensive set of FIDO message type definitions. It was shown that CBOR offers an efficient encoding scheme for messages exchanged between client and RP, reducing the message size compared to web-based FIDO ceremonies that use JSON serialization.

The FIDO2-TLS extension has been implemented for the `OpenSSL` library, demonstrating the practical feasibility of the methodology. In addition, the extension was showcased in a practical application, specifically in a Wi-Fi environment using 802.11X EAP-TLS. It was shown that FIDO key registration within TLS is possible but not always desirable. In IoT environments, this method could prove advantageous, while for scenarios involving user interaction, such as with eduroam, out-of-band key enrollment through traditional web interfaces may be more suitable.

The specification of the extension does not mandate how the backend of the FIDO server is implemented. Depending on the use case, the backend can be a dedicated FIDO authentication server, or it could utilize an internal or external public key database. The provided PoC uses an internal `SQLite` database to store public key material and the corresponding metadata.

The methodology of this thesis mandated that FIDO be integrated only using the official TLS extension mechanism. This approach allows for quicker adoption, easier standardization, and thorough testing of new features without the lengthy audit processes associated with core library changes. Even if the extension isn't formally standardized, it can still function alongside the core TLS library and be deployed in production as a non-standard extension. Should it prove successful, this extension might eventually be incorporated into the core protocol. The evaluation further illustrated how such an integration into the core library might be structured. It presented an alternative TLS handshake that incorporates FIDO directly, eliminating the need for a double handshake. This approach increases the RTT of the handshake for non-discoverable credentials and key registration. However, the overall RTTs of the whole authentication process does not increase.

Integrating FIDO within the transport layer of the OSI model has proven advantageous because it allows applications to utilize this authentication method without needing to implement FIDO logic within their own protocols. Given the successful integration of FIDO authentication into EAP-TLS, it is reasonable to suggest that other TLS-based protocols such as OpenVPN, IMAPS, SMTPS, FTPS, and NNTPS might similarly benefit from this extension, requiring only minor modifications to their source code.

References

[1] Openssh git repository. `https://anongit.mindrot.org/openssh.git`.

[2] Openssh release notes. `https://www.openssh.com/releasenotes.html`.

[3] D. E. E. 3rd. Transport Layer Security (TLS) Extensions: Extension Definitions. RFC 6066, Jan. 2011.

[4] Anna Sinitsyna. FIDO2 and WebAuthn in Practice. `https://www.inovex.de/de/blog/fido2-webauthn-in-practice/`, 2019. Accessed on 2024-05-07.

[5] C. Bormann and P. E. Hoffman. Concise Binary Object Representation (CBOR). RFC 7049, Oct. 2013.

[6] T.-L. J. Breitkopf. FIDO2 als TLS-1.3-Erweiterung, 2020.

[7] M. Fischlin and F. Gunther. Replay attacks on zero round-trip time: The case of the tls 1.3 handshake candidates. In *2017 IEEE European Symposium on Security and Privacy*. IEEE, Apr. 2017.

[8] M. Freund. FIDO2-Erweiterung von TLS 1.3 in einer gängigen Kryptobibliothek, 2021.

[9] M. S. Gast. *802.11 wireless networks*. O'Reilly, Beijing [u.a.], 2. ed. edition, 2013.

[10] E. Huseynov. Passwordless vpn using fido2 security keys: Modern authentication security for legacy vpn systems. In *2022 4th International Conference on Data Intelligence and Security (ICDIS)*. IEEE, Aug. 2022.

[11] International Telecommunication Union. ITU-T X.680. ITU-T Recommendation X.680, Telecommunication Standardization Sector of ITU, July 2002. Series X: Data Networks and Open System Communications.

[12] Internet Assigned Numbers Authority (IANA). TLS ExtensionType Values. `https://www.iana.org/assignments/tls-extensiontype-values/tls-extensiontype-values.xhtml`, 2024. Accessed on 2024-04-25.

[13] J. F. Kurose. *Computer networking*. Pearson Education, Boston, seventh edition, global edition edition, 2017.

[14] J. Malinen. Hostapd - IEEE 802.11 AP, IEEE 802.1X/WPA/WPA2/EAP/RADIUS Authenticator. `https://w1.fi/hostapd/`, 2024. Accessed on 2024-04-09.

[15] J. Malinen. WPA Supplicant - IEEE 802.1X/WPA/WPA2/EAP/RADIUS Authenticator and Supplicant. `https://w1.fi/wpa_supplicant/`, 2024. Accessed on 2024-04-09.

[16] S. K. Matharu. Exploiting ssl/tls vulnerabilities in modern technologies. pages 30–32, 2020.

[17] OpenSSH Project. OpenSSH Protocol Specification for U2F. `https://anongit.mindrot.org/openssh.git`, 2019. Commit: 57ecc10628b04c384cbba2fbc87d38b74cd1199d, File: PROTOCOL.u2f.

[18] OpenSSL. OpenSSL Documentation - `SSL_CTX_add_custom_ext()`. `https://www.openssl.org/docs/man3.2/man3/SSL_CTX_add_custom_ext.html`, 2024. Accessed on 2024-04-09.

[19] A. Parsovs. Practical issues with tls client certificate authentication. Cryptology ePrint Archive, Paper 2013/538, 2013. `https://eprint.iacr.org/2013/538`.

[20] E. Rescorla. HTTP Over TLS. RFC 2818, May 2000.

[21] E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.3. RFC 8446, Aug. 2018.

[22] J.-F. Rieckers and S. Winter. Eap method for fido. Internet Engineering Task Force (IETF) Internet-Draft. Work in progress, draft version.

[23] J. Schaad. CBOR Object Signing and Encryption (COSE). RFC 8152, July 2017.

[24] D. Simon, R. Hurst, and D. B. D. Aboba. The EAP-TLS Authentication Protocol. RFC 5216, Mar. 2008.

[25] W. Stallings. *Cryptography and network security.* Pearson/Prentice Hall, Upper Saddle River, NJ, 4. ed. edition, 2006.

[26] S. Turner. Transport layer security. *IEEE Internet Computing*, Nov. 2014.

[27] J. C. Viotti and M. Kinderkhedia. A survey of json-compatible binary serialization specifications, 2022.

[28] Web Hypertext Application Technology Working Group (WHATWG). HTML Living Standard. `https://html.spec.whatwg.org/multipage/browsers.html#concept-origin`, 2022. Accessed: 2024-04-09.

[29] Web Hypertext Application Technology Working Group (WHATWG). Web IDL Specification. `https://webidl.spec.whatwg.org/#idl-DOMException-error-names`, 2022. Accessed: 2024-04-18.

[30] World Wide Web Consortium. Attestation Object Structure. `https://www.w3.org/TR/webauthn-2/images/fido-attestation-structures.svg`, 2021. Accessed on 2024-05-08.

[31] World Wide Web Consortium. Web authentication: An api for accessing public key credentials - level 2. https://www.w3.org/TR/webauthn-2/, 2021. Accessed: 2024-04-04.

[32] World Wide Web Consortium. Fido authentication flow. https://www.w3.org/TR/webauthn-2/images/webauthn-authentication-flow-01.svg, 2022.

[33] World Wide Web Consortium. Fido registration flow. https://www.w3.org/TR/webauthn-2/images/webauthn-registration-flow-01.svg, 2022.

[34] Yubico. FIDO U2F Overview. https://docs.yubico.com/yesdk/users-manual/application-u2f/fido-u2f-overview.html, 2021. Accessed on 2024-05-16.

[35] Yubico Support. How many accounts can I register my YubiKey with? https://support.yubico.com/hc/en-us/articles/360013790319-How-many-accounts-can-I-register-my-YubiKey-with, 2020. Accessed on 2024-05-04.

List of Figures

A. Message Specifications

This appendix details the message types and structures used within the proposed protocol, which exclusively uses CBOR for encoding all data elements. CBOR is a binary data serialization format that allows for the encoding of data structures including integers, floating-point numbers, strings, arrays, and maps. CBOR inherently embeds the size and data type of each element within its encoding, thus eliminating the need for explicit length fields. All messages utilize a CBOR array as the top-level container, encapsulating required data fields. For messages that contain optional data, these fields are encoded within a CBOR map. This structure enables simple key-based lookups, allowing for quick verification of the existence and retrieval of optional values. Some optional data elements are grouped into tuples. If present, these tuples are stored in a sub-array, preserving their relational structure. Furthermore, certain optional tuples may exhibit zero or multiple occurrences, in which case they are organized into nested CBOR arrays.

Enums are frequently used within the specification to represent specific choices or states. As CBOR does not natively support enum types, these are encoded as integers. CBOR automatically uses tiny integers for integer values between 0 and 23, encoding them in a single byte to minimize the data size.

A.1. Pre Registration Indication

- **MESSAGE TYPE**:
  Type: Integer
  Value: 1
  Length: 1 Byte
  Description: Specifies the type of message, indicating a Pre-Registration Indication.

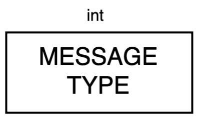

Figure 11: Message Type Definition: Pre-Registration Indication. Graphic created by author.

A.2. Pre-Registration Request

- **MESSAGE TYPE**:
  Type: Integer
  Value: 2
  Length: 1 Byte
  Description: Specifies the type of message, indicating a Pre-Registration Request.
- **EPHEMERAL USER ID**:
  Type: Byte String
  Length: Variable, up to 256 Bytes
  Description: A ephemeral reference that links this handshake to the following one.
- **GCM KEY**:
  Type: Byte String
  Length: $28 - 44$ Bytes
  Description: A key used for symmetric encryption, utilizing the AES algorithm in conjunction with GCM.

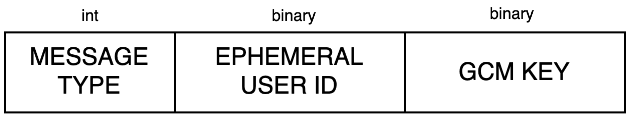

Figure 12: Message Type Definition: Pre-Registration Request. Graphic created by author.

A.3. Pre-Registration Response

- **MESSAGE TYPE**:
  Type: Integer
  Value: 3
  Length: 1 Byte
  Description: Specifies the type of message, indicating a Pre-Registration Response.
- **USER NAME**:
  Type: UTF-8 String
  Length: Variable, up to 255 Bytes
  Description: The username associated with the end user's account.
- **USER DISPLAY NAME**:
  Type: UTF-8 String
  Length: Variable, up to 255 Bytes
  Description: The full name of the user, intended for display purposes within user.
- **TICKET**:
  Type: Byte String
  Length: 128 – 512 Bytes
  Description: A unique ticket that enabled the user to register new FIDO keys with the RP.

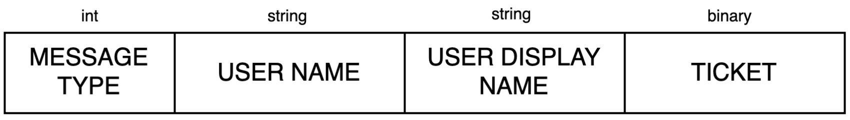

Figure 13: Message Type Definition: Pre-Registration Response. Graphic created by author.

A.4. Registration Indication

- **MESSAGE TYPE**:
  Type: Integer
  Value: 4
  Length: 1 Byte
  Description: Specifies the type of message, indicating a Registration Indication.
- **EPHEMERAL USER ID**:
  Type: Byte String
  Length: Variable, up to 256 Bytes
  Description: A ephemeral reference to a previous handshake.

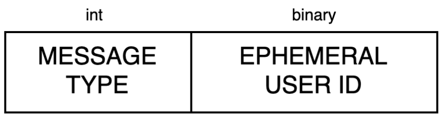

Figure 14: Message Type Definition: Registration Indication. Graphic created by author.

A.5. Registration Request

- **MESSAGE TYPE**:
  Type: Integer
  Value: 5
  Length: 1 Byte
  Description: Specifies the type of message, indicating a Registration Request.

- **CHALLENGE**:
  Type: Byte String
  Length: $16 - 64$ Bytes
  Description: Random challenge from the RP.

- **RP ID**:
  Type: UTF-8 String
  Length: Variable, up to 256 Bytes
  Description: Relying Party identifier.

- **RP NAME**:
  Type: UTF-8 String
  Length: Variable, up to 255 Bytes
  Description: A human-readable name for the Relying Party, used mainly for display.

- **USER NAME**:
  Type: UTF-8 String
  Length: Variable, up to 255 Bytes
  Description: The username associated with the end user's account. This value is encrypted with the GCM Key.

- **USER DISPLAY NAME**:
  Type: UTF-8 String
  Length: Variable, up to 255 Bytes
  Description: The full name of the user, intended for display purposes within user. This value is encrypted with the GCM Key.

- **USER ID**:
  Type: Byte String
  Length: Variable, up to 64 Bytes
  Description: A unique identifier for the user in the context of the Relying Party. This value is encrypted with the GCM Key.

- **PUBKEY CRED PARAMS**:
  Type: Array of Integers
  Length: $1 - 6$ Bytes
  Description: List of possible public key algorithms, in descending preference. Each

Integer represents a enum value, mapped to following public key algorithms:
0: COSE_ES256
1: COSE_ES384
2: COSE_EDDSA
3: COSE_ECDH_ES256
4: COSE_RS256
5: COSE_RS1

- **OPTIONALS**:
  Type: CBOR map
  Description: Contains optional values that can be probed by their keys.
  - **TIMEOUT**:
    Type: Integer
    Key: 1
    Length: 4 Bytes
    Description: Maximum time, in milliseconds, that the client should wait for the user to complete the action.
  - **AUTHENTICATOR SELECTION**:
    Type: CBOR array
    Key: 2
    Description: Criteria the RP wants to impose regarding the authenticators to be used. All values of the array are encrypted with the GCM key.
    * **ATTACHMENT**:
      Type: Integer
      Length: 1 Byte
      Description: Criteria the RP wants to impose regarding authenticator attachment. Enum value that maps to:
      0: PLATFORM
      1: CROSS-PLATFORM
    * **RESIDENT KEY**:
      Type: Integer
      Length: 1 Byte
      Description: Criteria the RP wants to impose regarding discoverable credentials. Enum value that maps to:
      0: REQUIRED
      1: PREFERRED
      2: DISCOURAGED
    * **USER VERIFICATION**:
      Type: Integer
      Length: 1 Byte
      Description: Criteria the RP wants to impose regarding user verification. Enum value that maps to:

    0: REQUIRED
    1: PREFERRED
    2: DISCOURAGED

- **EXCLUDED CREDENTIALS**:
  Type: Nested N $\times$ 3 CBOR array
  Key: 3
  Description: This array contains credentials already registered with the user. If the authenticator detects any of these credentials as existing on the device, it must return an error to prevent the creation of duplicates. All values of the array are encrypted with the GCM key.
  - **CREDENTIAL TYPE**: *Type*: Integer
    Length: 1 Byte
    Description: Specifies the type of credential. Currently, only one type is supported by Webauthn, but the design is structured to allow for future expansions. Enum value that maps to:
    0: PUBLIC_KEY
  - **CREDENTIAL ID**:
    Type: Byte String
    Length: Variable, up to 256 Bytes
    Description: Unique identifier for a FIDO credential
  - **TRANSPORTS**:
    Type: Integer
    Length: 1 Byte
    Description: Mode of transportation for this credential. Enum value that maps to:
    0: USB
    1: NFC
    2: BLE
    3: INTERNAL
- **ATTESTATION**:
  Type: Integer
  Key: 4
  Length: 1 Byte
  Description: Specifies the desired attestation conveyance preference of the RP. This values is encrypted with the GCM key. Enum value that maps to:
  0: NONE
  1: INDIRECT
  2: DIRECT
  3: ENTERPRISE
- **EXTENSIONS**:
  Type: Nested M $\times$ 2 CBOR array
  Key: 5

Description: Custom extension data. All values of the array are encrypted with the GCM key.

∗ **EXTENSION ID**:
  Type: UTF-8 String
  Length: Variable, up to 256 Bytes
  Description: Unique identifier of the extension

∗ **EXTENSION DATA**:
  Type: Byte String
  Length: Variable, up to 2^{12} Bytes
  Description: Data corresponding to the extension, which may include any necessary parameters or configuration settings to support the extension's function. The structure and content of this data are specific to each extension and define how the extension modifies or enhances the base protocol operation.

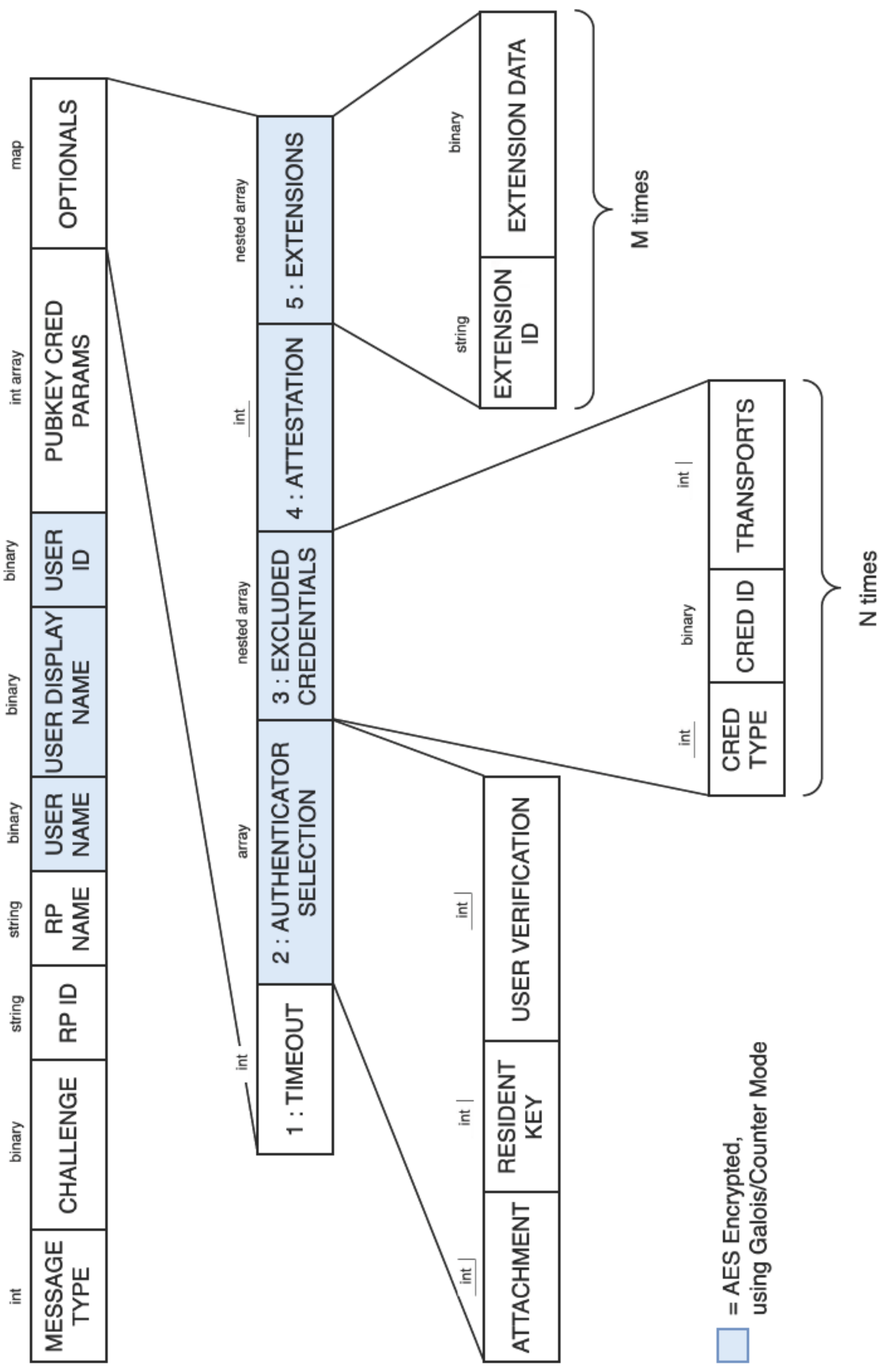

Figure 15: Message Type Definition: Registration Request. Graphic created by author.

A.6. Registration Response

- **MESSAGE TYPE**:
  Type: Integer
  Value: 6
  Length: 1 Byte
  Description: Specifies the type of message, indicating a Registration Response.
- **ATTESTATION OBJECT**:
  Type: Byte Array
  Length: Variable, up to 2^{13} Bytes
  Description: A binary representation of the attestation object, which contains cryptographic proof of the newly created credential. This object is used by the RP to verify the integrity and origin of the new public key credential. The attestation object is defined by Webauthn, as shown in Figure 17
- **CLIENTDATA JSON**:
  Type: UTF-8 String
  Length: Variable, up to 2^{11} Bytes
  Description: A JSON-serialized string containing the client-side data used during the credential creation process. This includes type, challenge and origin.

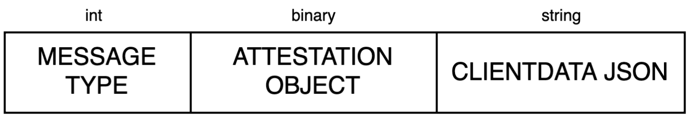

Figure 16: Message Type Definition: Registration Response. Graphic created by author.

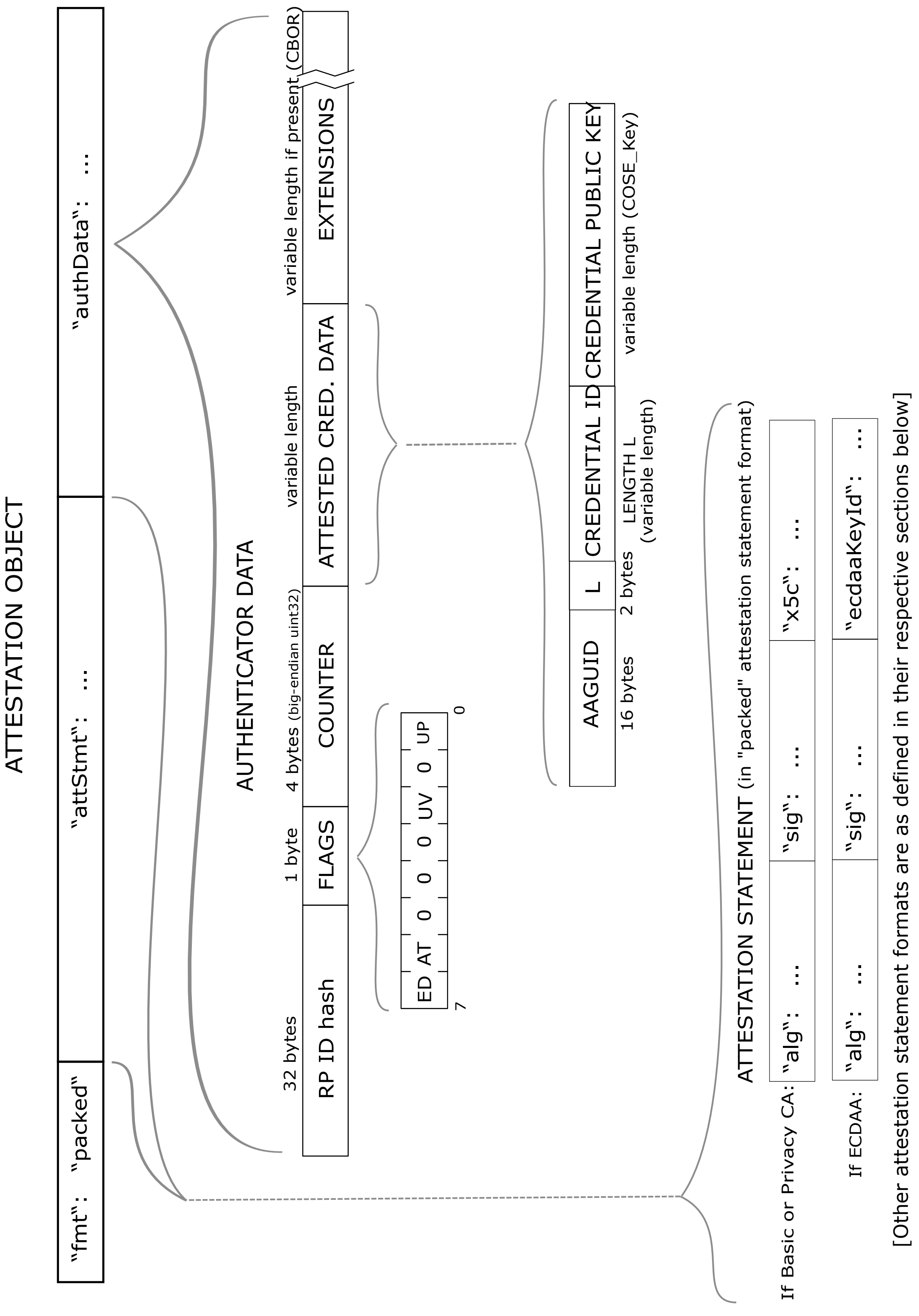

Figure 17: FIDO Attestation Object Definition (Source: W3C [30])

A.7. Pre-Authentication Indication

- **MESSAGE TYPE**:
  Type: Integer
  Value: 7
  Length: 1 Byte
  Description: Specifies the type of message, indicating a Pre-Authentication Indication.

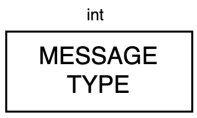

Figure 18: Message Type Definition: Pre-Authentication Indication. Graphic created by author.

A.8. Pre-Authentication Request

- **MESSAGE TYPE**:
  Type: Integer
  Value: 8
  Length: 1 Byte
  Description: Specifies the type of message, indicating a Pre-Authentication Request.
- **EPHEMERAL USER ID**:
  Type: Byte String
  Length: Variable, up to 256 Bytes
  Description: A ephemeral reference that links this handshake to the following one.
- **GCM KEY**:
  Type: Byte String
  Length: $28 - 44$ Bytes
  Description: A key used for symmetric encryption, utilizing the AES algorithm in conjunction with GCM.

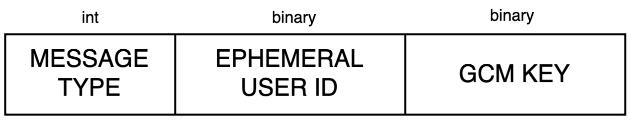

Figure 19: Message Type Definition: Pre-Authentication Request. Graphic created by author.

A.9. Pre-Authentication Response

- **MESSAGE TYPE**:
  Type: Integer
  Value: 9
  Length: 1 Byte
  Description: Specifies the type of message, indicating a Pre-Authentication Response.

- **USER NAME**:
  Type: UTF-8 String
  Length: Variable, up to 255 Bytes
  Description: The username associated with the end user's account.

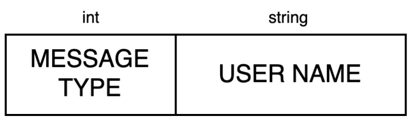

Figure 20: Message Type Definition: Pre-Authentication Response. Graphic created by author.

A.10. Authentication Indication

- **MESSAGE TYPE**:
  Type: Integer
  Value: 10
  Length: 1 Byte
  Description: Specifies the type of message, indicating an Authentication Indication.
- **EPHEMERAL USER ID**:
  Type: Byte String
  Length: Variable, up to 256 Bytes
  Description: A ephemeral reference to a previous handshake.

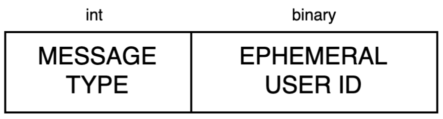

Figure 21: Message Type Definition: Authentication Indication. Graphic created by author.

A.11. Authentication Request

- **MESSAGE TYPE**:
  Type: Integer
  Value: 11
  Length: 1 Byte
  Description: Specifies the type of message, indicating an Authentication Request.
- **CHALLENGE**:
  Type: Byte String
  Length: $16 - 64$ Bytes
  Description: Random challenge from the RP.
- **OPTIONALS**:
  Type: CBOR map
  Description: Contains optional values that can be probed by their keys.
  - **TIMEOUT**:
    Type: Integer
    Key: 1
    Length: 4 Bytes
    Description: Maximum time, in milliseconds, that the client should wait for the user to complete the authentication process.
  - **RP ID**:
    Type: UTF-8 String
    Key: 2
    Length: Variable, up to 256 Bytes
    Description: Relying Party identifier.
  - **USER VERIFICATION**:
    Type: Integer
    Key: 3
    Length: 1 Byte
    Description: Specifies the level of user verification required for authentication. Enum value that maps to:
    0: REQUIRED
    1: PREFERRED
    2: DISCOURAGED
  - **ALLOW CREDENTIALS**:
    Type: Nested N $\times$ 3 CBOR array
    Key: 4
    Description: List of credentials permitted for use in the authentication.

    * **CREDENTIAL TYPE**:
      Type: Integer
      Length: 1 Byte
      Description: Specifies the type of credential. Currently, only one type is supported by Webauthn, but the design is structured to allow for future expansions. Enum value that maps to:
      0: PUBLIC_KEY
    * **CREDENTIAL ID**:
      Type: Byte String
      Length: Variable, up to 256 Bytes
      Description: Unique identifier for the credential.
    * **TRANSPORTS**:
      Type: Integer
      Length: 1 Byte
      Description: Mode of transportation for this credential. Enum value that maps to:
      0: USB
      1: NFC
      2: BLE
      3: INTERNAL
- **EXTENSIONS**:
  Type: Nested M $\times$ 2 CBOR array
  Key: 5
  Description: Custom extension data. All values of the array are encrypted with the GCM key.
    * **EXTENSION ID**:
      Type: UTF-8 String
      Length: Variable, up to 256 Bytes
      Description: Unique identifier of the extension.
    * **EXTENSION DATA**:
      Type: Byte String
      Length: Variable, up to 2^{12} Bytes
      Description: Data corresponding to the extension, which may include any necessary parameters or configuration settings to support the extension's function.

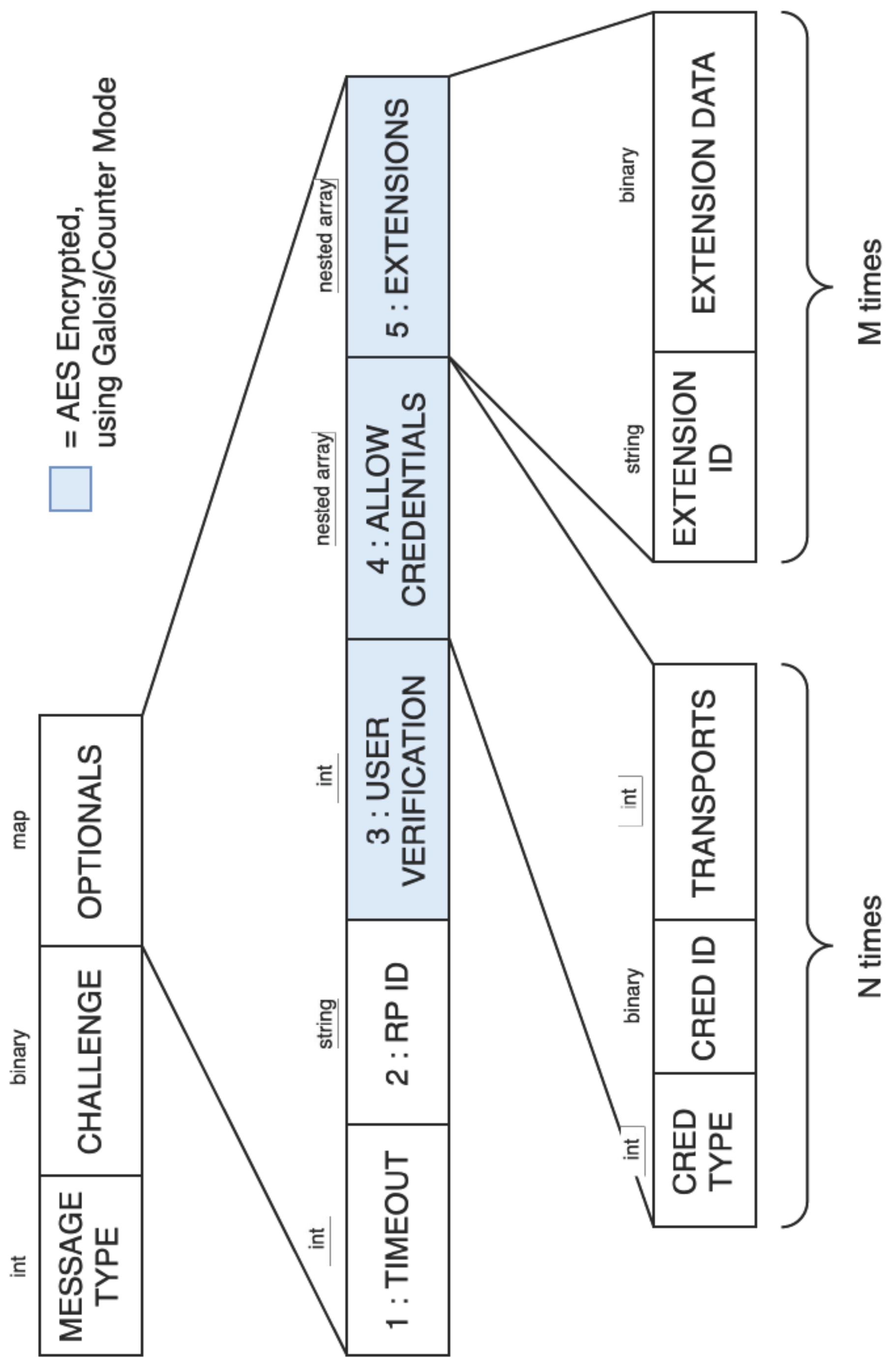

Figure 22: Message Type Definition: Authentication Request. Graphic created by author.

A.12. Authentication Response

- **MESSAGE TYPE**:
  Type: Integer
  Value: 12
  Length: 1 Byte
  Description: Specifies the type of message, indicating an Authentication Response.
- **CLIENTDATA JSON**:
  Type: UTF-8 String
  Length: Variable, up to 2^{11} Bytes
  Description: A JSON-serialized string containing the client-side data used during the authentication process. This includes type, challenge, and origin.
- **AUTHENTICATOR DATA**:
  Type: Byte Array
  Length: Variable, up to 512 Bytes
  Description: A binary representation of the authenticator data, which contains data from the authenticator, including flags and counters.
- **SIGNATURE**:
  Type: Byte Array
  Length: Variable, up to 256 Bytes
  Description: A digital signature over the concatenation of the authenticator data and the clientDataHash.
- **OPTIONALS**:
  Type: CBOR map
  Description: Contains optional values that can be probed by their keys.
  - **USER HANDLE**:
    Type: Byte String
    Key: 1
    Length: Variable, up to 64 Bytes
    Description: The user ID, that uniquely identifies the user. If discoverable credentials were used, this field is mandatory.
  - **SELECTED CREDENTIAL ID**:
    Type: Byte String
    Key: 2
    Length: Variable, up to 256 Bytes
    Description: The identifier of the credential that was used in the authentication.
  - **CLIENT EXTENSION OUTPUT**:
    Type: Nested N $\times$ 2 CBOR array
    Key: 3
    Description: Custom extension data returned by the client.

- **EXTENSION ID**:
  Type: UTF-8 String
  Length: Variable, up to 256 Bytes
  Description: Unique identifier of the extension.
- **EXTENSION DATA**:
  Type: Byte String
  Length: Variable, up to 2^{12} Bytes
  Description: Data corresponding to the extension, which may include any necessary parameters or configuration settings to support the extension's function.

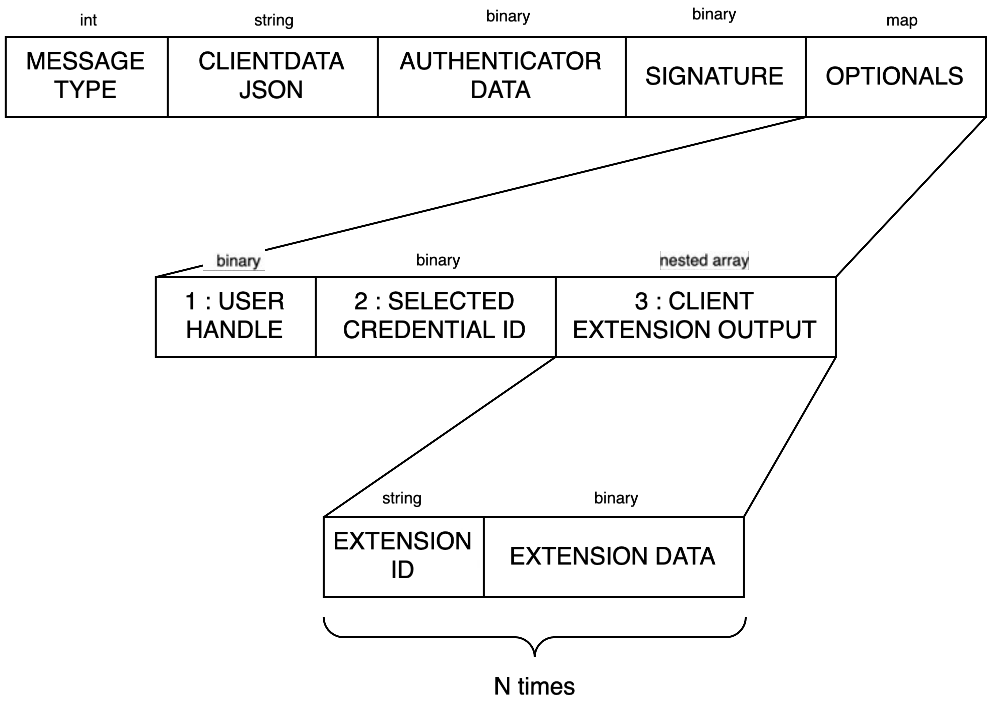

Figure 23: Message Type Definition: Authentication Response. Graphic created by author.

B. Test PKI and Certificate Creation Guide

This appendix provides commands necessary to establish a local PKI and generate certificates using `OpenSSL`. These resources are intended to assist developers in creating a controlled testing environment for the proposed client-server application.

B.1. Certificate Authority

```
# Generate an EC Private Key for the CA
openssl ecparam -genkey -name secp384r1 -out ca.key

# Generate a CA Certificate
openssl req -x509 -new -nodes -key ca.key -sha384 -days 1024 -out ca.pem
```

B.2. Client

```
# Generate an EC Private Key for the Client
openssl ecparam -genkey -name secp384r1 -out client.key

# Generate a Certificate Signing Request (CSR) for the Client
openssl req -new -sha384 -key client.key -out client.csr

# Sign the Client CSR with the CA Certificate
openssl x509 -req -in client.csr -CA ca.pem -CAkey ca.key -CAcreateserial -out \
    client.pem -days 1024 -sha384

# Verify the Client Certificate
openssl verify -CAfile ca.pem client.pem
```

B.3. Server

A configuration file named `cert.cnf` is initially generated for the server, incorporating the SAN field of the certificate:

```
[ req ]
default_bits       = 2048
default_keyfile    = server.key
distinguished_name = req_distinguished_name
req_extensions     = req_ext
x509_extensions    = v3_ca
prompt             = no

[ req_distinguished_name ]
C                   = DE
ST                  = Berlin
L                   = Berlin
O                   = Test Company GmbH
OU                  = Test Division
CN                  = fido.tls.extension

[ req_ext ]
subjectAltName      = @alt_names

[ v3_ca ]
subjectAltName      = @alt_names

[ alt_names ]
DNS.1               = fido.tls.extension
DNS.2               = localhost
IP.1                = 127.0.0.1
```

Next up, this configuration is used to generate the certificate:

```
# Generate an EC Private Key for the Server
openssl ecparam -genkey -name secp384r1 -out server.key

# Generate a Certificate Signing Request (CSR) for the Server
openssl req -new -sha384 -key server.key -out server.csr -config cert.cnf

# Create and Sign the Server Certificate
openssl x509 -req -in server.csr -CA ca.pem -CAkey ca.key -CAcreateserial \
    -out server.pem -days 1024 -sha384 -extfile cert.cnf -extensions v3_ca

# Verify the Server Certificate
openssl verify -CAfile ca.pem server.pem
```

C. Configuration Files for EAP-TLS

This appendix provides the actual configuration files used in the implementation of the FIDO2 TLS 1.3 Extension within an EAP-TLS framework as discussed in the main body of this thesis. These files are essential for setting up and operating the wireless network environment that utilizes enhanced FIDO-based authentication. The configuration examples include the `eap_user_file` for defining EAP users, the `hostapd.conf` for configuring the access point, and the `wpa_supplicant.conf` for setting up the client-side network parameters.

C.1. eap_user_file

```
# EAP User File for hostapd
# Each line defines an EAP user

"Alice" TLS
```

C.2. hostapd.conf

```
## Basic network configuration settings:

# Interface used by the access point
interface=wlan0

# SSID of the Wi-Fi network
ssid=DemoNetwork

# Wireless mode, g for 2.4 GHz
hw_mode=g

# Wi-Fi channel to use
channel=6

# Authentication algorithm; 1 for open system
auth_algs=1

# Broadcast SSID; 0 to enable SSID broadcasting
ignore_broadcast_ssid=0

# WPA2 only
wpa=2

# Key management set to WPA-EAP for enterprise networks
wpa_key_mgmt=WPA-EAP

# WPA pairwise encryption method
wpa_pairwise=CCMP
```

```
# WPA2 pairwise encryption method
rsn_pairwise=CCMP

# Enables 802.1X authentication
ieee8021x=1

# Internal EAP server enabled
eap_server=1

# Path to EAP user database file
eap_user_file=/path/to/eap_user_file

# Path to CA certificate file
ca_cert=/path/to/ca.crt

# Path to server's certificate
server_cert=/path/to/server.crt

# Path to server's private key
private_key=/path/to/server.key

# Enables TLS version 1.3
tls_flags=[ENABLE-TLSv1.3]

## FIDO2-specific configurations:

# Relying Party identifier for FIDO
fido_rp_id=demo.fido2.tls.edu

# Descriptive name for the Relying Party
fido_rp_name=Demo_FIDO2_TLS

# User verification policy (required, preferred, discouraged)
fido_user_verification=required

# Policy for device-resident keys (required, preferred, discouraged)
fido_resident_key=required

# Type of authenticator attachment (cross-platform or platform)
fido_auth_attach=cross-platform

# Transport protocol for the FIDO device (usb, nfc, ble, internal)
fido_transport=usb

# Timeout for FIDO operations in milliseconds
fido_timeout=30000

# Debug level for FIDO operations: 1 for errors, 2 for verbose, 3 for very verbose
fido_debug_level=2
```

C.3. wpa_supplicant.conf

```
ctrl_interface=/var/run/wpa_supplicant

network={
    ssid="DemoNetwork"
    scan_ssid=1
    key_mgmt=WPA-EAP
    eap=TLS
    identity="Alice"
    ca_cert="/path/to/ca.crt"
    client_cert="/path/to/client.crt"
    private_key="/path/to/client.key"
}
```

D. Scripting Simultaneous Startup and Shutdown of Client-Server Applications

This appendix provides a bash script that uses the terminal multiplexer `tmux` to facilitate the launching and restarting of client and server instances simultaneously. This automation is particularly beneficial in development environments where frequent restarts of client and server software are necessary to incorporate new code changes and test functionality. Prior to utilizing this script, ensure the following environment variables are properly set:

```bash
# Path to the client application
export TMUX_PATH1="$HOME/MyApplication/Client"

# Path to the server application
export TMUX_PATH2="$HOME/MyApplication/Server"

# Command to launch the client
export TMUX_CMD1='make && ./build/client'

# Command to launch the server
export TMUX_CMD2='make && ./build/server'
```

With these environment variables configured, the script below can be executed. It creates a new `tmux` session with two vertically split panes. The upper pane navigates to `$TMUX_PATH1`, and the lower pane to `$TMUX_PATH2`. Subsequently, `$TMUX_CMD1` executes in the upper pane and `$TMUX_CMD2` in the lower. These commands can include compiling and launching the applications, depending on how they are configured. If the script is executed while the session is already active, it selects the existing `tmux` window, sends a SIGINT signal to terminate running instances, and re-executes the specified commands for both the client and server. Creating a terminal shortcut to execute this script is advised, as it facilitates the quick launching and restarting of client and server applications, enhancing the efficiency of the development cycle.

```bash
#!/usr/bin/env bash

win=split-run

# Create or select window
if tmux list-windows | grep -q "$win"; then
    pane_pids=$(tmux list-panes -t "$win" -F '#{pane_pid}' | tr '\n' ' ')
    read -r pid1 pid2 <<< "$pane_pids"

    if pgrep -P "$pid1" > /dev/null; then
        tmux send-keys -t "$win.1" C-c
    fi
    if pgrep -P "$pid2" > /dev/null; then
        tmux send-keys -t "$win.2" C-c
    fi
    while pgrep -P "$pid1" > /dev/null || pgrep -P "$pid2" > /dev/null; do
        sleep 0.1
    done
    unset pid1 pid2 pane_pids
else
    tmux new-window -n "$win" -c "$TMUX_PATH1"
    tmux split-window -v -c "$TMUX_PATH2"
fi

# Execute commands
tmux send-keys -t "$win.1" "$TMUX_CMD1" Enter
tmux send-keys -t "$win.2" "$TMUX_CMD2" Enter
tmux select-window -t "$win"

unset win
```

E. Maximum Size Analysis of Message A5

MESSAGE TYPE:
Type Field: 1 byte (CBOR small integer)
Length Field: N/A (length is implicit in the type for small integers)
Value: N/A (value is encoded directly in the type field)
Total: 1 byte

CHALLENGE:
Type Field: 1 byte (CBOR byte string type indicator)
Length Field: 1 byte (64 bytes length can be represented in 1 byte)
Value: 64 bytes
Total: 66 bytes

RP ID:
Type Field: 1 byte (CBOR text string type indicator)
Length Field: 1 byte (256 bytes length can be represented in 1 byte)
Value: 256 bytes
Total: 258 bytes

RP NAME:
Type Field: 1 byte (CBOR text string type indicator)
Length Field: 1 byte (255 bytes length can be represented in 1 byte)
Value: 255 bytes
Total: 257 bytes

USER NAME:
Type Field: 1 byte (CBOR text string type indicator)
Length Field: 1 byte (255 bytes length can be represented in 1 byte)
Value: 255 bytes
Total: 257 bytes

USER DISPLAY NAME:
Type Field: 1 byte (CBOR text string type indicator)
Length Field: 1 byte (255 bytes length can be represented in 1 byte)
Value: 255 bytes
Total: 257 bytes

USER ID:
Type Field: 1 byte (CBOR byte string type indicator)
Length Field: 1 byte (64 bytes length can be represented in 1 byte)
Value: 64 bytes
Total: 66 bytes

PUBKEY CRED PARAMS:
Type Field: 1 byte (CBOR array type indicator, size included for arrays < 24 elements)
Length Field: N/A (length is implicitly defined within the type for small arrays)
Value: 6 bytes (array of integers, each less than 24, encoded directly in one byte each)
Total: 7 bytes

OPTIONALS:
Type Field: 1 byte (CBOR map type indicator, size included for maps with < 24 pairs)
*Length Fiel*d: N/A (count is implicit in the type for small maps)
Values: All the following items
Total: 1 byte + all following items

TIMEOUT:
Key Type & Length Field: 1 byte (CBOR small integer)
Value Type Field: 1 byte (indicating 4-byte unsigned integer)
Value Length Field: N/A (not needed since type implicitly denotes the length)
Value: 4 bytes
Total: 6 bytes

AUTHENTICATOR SELECTION:
Key Type & Length Field: 1 byte (CBOR small integer)
Value Type Field: 1 byte (CBOR array type with 3 items, size included)
Value Length Field: N/A (not needed since type implicitly denotes the length)
Value: 3 bytes (each element is a CBOR small integer)
Total: 5 bytes

EXCLUDED CREDENTIALS:
Key Type & Length Field: 1 byte (CBOR small integer)
Value Type Field: 1 byte (CBOR array type)
Value Length Field: 2 bytes (length of array allowing for up to 2^{16} elements)
Value: $N \times 261$ bytes (each credential has 260 bytes + 1 byte for the array)
Total: 4 bytes + $N \times 261$ bytes

ATTESTATION:
Key Type & Length Field: 1 byte (CBOR small integer)
Value Type Field: 1 byte (CBOR small integer)
Value Length Field: N/A (not needed since type implicitly denotes the length)
Value: N/A (value is encoded directly in the type field)
Total: 2 bytes

EXTENSIONS:
Key Type & Length Field: 1 byte (CBOR small integer)
Value Type Field: 1 byte (CBOR array type)
Value Length Field: 2 bytes (length of array allowing for up to 2^{16} elements)

Value: $M \times 4358$ bytes (258 bytes for EXTENSION ID, $3 + 2^{12}$ bytes for EXTENSION DATA, 1 byte for array)
Total: 4 bytes + $M \times 4358$ bytes

Maximum message size

Given the individual totals for each part, we can sum these to find the total byte size of the message.

$$\begin{aligned}\text{Total bytes} &= 1 + 66 + 258 + 257 + 257 + 257 + 66 + 7 + 1 \\ &\quad + 6 + 5 + (4 + 261N) + 2 + (4 + 4358M) \\ &= 1191 + 261N + 4358M\end{aligned}$$

where:

- N is the number of `excludedCredentials`.
- M is the number of `extensions`.