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

FIDO2-TLS-1.3-Erweiterung in JSSE

Masterarbeit

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

eingereicht von: Gloria Geise
geboren am:
geboren in:

Gutachter/innen: Prof. Dr. rer. nat. Jens-Peter Redlich
Prof. Dr. rer. nat. Ernst-Günter Giessmann

eingereicht am: .... verteidigt am: ....

Danksagung

An dieser Stelle möchte ich mich bei denjenigen bedanken, die mich während der Anfertigung dieser Masterarbeit unterstützt und motiviert haben.

Zuerst gebührt mein Dank Dr. rer. nat. Wolf Müller, welcher meine Masterarbeit betreut hat und mir bei allen Fragestellungen zur Seite stand. Für die hilfreichen Anregungen, die aufgebrachte Zeit und die konstruktive Kritik bei der Erstellung dieser Arbeit möchte ich mich herzlich bedanken.

Ein besonderer Dank gilt meinem Lebenspartner Dustin Gerd Teschner, der mit viel Geduld stets an meiner Seite war und regelmäßig diese Arbeit korrekturgelesen hat.

Zusammenfassung

Heutzutage ist das Passwort weiterhin die weit verbreitetste Methode, um sich bei Online-Diensten zu authentifizieren. [48] Durch die Aneignung einer unsicheren Praxis im Umgang mit Passwörtern wird ein breiter Spielraum für mögliche Angriffe geschaffen. [36] Um potenziellen Angreifern diese Möglichkeit zu verwehren und gleichzeitig eine Authentifizierung ohne Passwörter zu ermöglichen, wurde hierfür ein Standard entwickelt. FIDO2 bietet die Möglichkeit unter Anwendung kryptografischer Verfahren und Verwendung eines Authentifikators, eine sichere Ende-zu-Ende-Authentifizierung durchzuführen. [15] [16] Um FIDO2 breiter aufzustellen, wurde in dieser Arbeit eine TLS-Erweiterung erarbeitet, die die Registrierung eines solchen Authentifikators vornimmt sowie auch die Authentifizierung anbietet. TLS wird bereits standardmäßig bei vielen Online-Diensten eingesetzt und bietet eine verschlüsselte und wahlweise auch authentifizierte Kommunikation an. [40] Zu diesem Thema wurde bereits eine Abschlussarbeit verfasst, in der die FIDO2-Authentifizierung in einer *hardware*-nahen Programmiersprache eingebettet wurde. [24] Im Zuge dieser Arbeit wurde dieses Konzept aufgenommen, modifiziert und um eine Registrierung erweitert. Die dazugehörige PoC Implementierung fand mittels der Programmierung in *Java* statt und konnte erfolgreich dieses Konzept umsetzen. Das Konzept sowie die PoC Implementation weisen Verbesserungspotenzial in Bezug auf Sicherheit und Rechenzeit auf.

Inhaltsverzeichnis

1. Einleitung und Motivation **1**

2. Technische Grundlagen **3**
2.1. Transport Layer Security Protokoll . . . 3
2.1.1. Aufbau . . . 3
2.1.2. Der TLS Handshake . . . 5
2.1.3. TLS Erweiterungen . . . 7
2.2. Fast Identity Online . . . 9
2.2.1. Vor- und Nachteile von FIDO . . . 13
2.2.2. Web Authentication . . . 14
2.2.3. Client-to-Authenticator Protokoll . . . 17
2.2.4. Sicherheitschätzung von FIDO2 . . . 19

3. TLS1.3-Erweiterung mit FIDO2 **21**
3.1. Theoretische Erwägung . . . 22
3.1.1. Vorteile . . . 22
3.1.2. Nachteile . . . 23
3.1.3. Einbettung in TLS . . . 23
3.2. TFE-Handshake . . . 26
3.2.1. Schematischer Aufbau . . . 27
3.2.2. Authentifizierung mit doppeltem Handshake . . . 28
3.2.3. Registrierung . . . 33
3.3. Diskussion . . . 34

4. PoC Implementation **36**
4.1. Verwendete Bibliotheken . . . 36
4.2. Implementierung . . . 38
4.3. Einschränkungen . . . 40
4.4. Diskussion . . . 41

5. Fazit **43**

Literaturverzeichnis **i**

A. Registrierungsnachrichten **v**
A.1. preFIDO . . . v
A.2. preFIDORequest . . . v
A.3. preFIDOResponse . . . vi
A.4. FIDO . . . vii
A.5. FIDORequest . . . vii
A.6. FIDOResponse . . . xii

B. Authentifizierungsnachrichten **xiv**
B.1. preFIDO xiv
B.2. preFIDORequest xiv
B.3. preFIDOResponse xv
B.4. FIDO xvi
B.5. FIDORequest xvi
B.6. FIDOResponse xix

C. Bauen einer JDK **xxi**

D. Zertifikate **xxii**

E. Ausführung **xxiv**

Abkürzungsverzeichnis

AES Advanced Encryption Standard

BSI Bundesamt für Sicherheit und Informationstechnik

CBOR Concise Binary Object Representation

CTAP Client-to-Authenticator Protocol

DoC Denial of Service

ECDH Elliptic Curve Diffie-Hellman

FIDO Fast Identity Online

FS Forward Secrecy

GCM Galois/Counter Mode

HTTP Hypertext Transfer Protocol

HTTPS Hypertext Transfer Protocol Secure

JDK Java Development Kit

JRE Java Runtime Environment

JSSE Java Secure Socket Extension

JSON JavaScript Object Notation

MITM Man-in-the-Middle

MFA Multi-Faktor-Authentifizierung

PoC Proof of Concept

PSK Pre-Shared Key

RFC Request for Comments

RP Relying Party

SFA Single-Faktor-Authentifizierung

TFE TLS1.3 mit FIDO2-Erweiterung

TLS Transport Layer Security

U2F Universal Second Factor

UAF Universal Authentication Framework

VPN Virtual Private Network

W3C World Wide Web Consortium

WebAuthn Web Authentication

Abbildungsverzeichnis

1. TLS *Handshake* mit Hervorhebung erweiterbarer Nachrichten . . . 5
2. Diffie-Hellman-Schlüsselaustausch 6
3. *ClientHello*-Erweiterungen beim Aufruf von `Youtube` 8
4. Kommunikationskanäle 12
5. FIDO Ablauf 12
6. WebAuthn Protokoll 15
7. Zusammenarbeit zwischen CTAP2 und WebAuthn 20
8. Aufbau der Client-Server-Anwendung 27
9. Doppelter *Handshake* mit Übergabe der FIDO2-Nachrichten 29
10. Aufbau der *Java*-Implementation 38

Tabellenverzeichnis

1. Kandidaten für eine geeignete *Java*-Bibliothek 37

1. Einleitung und Motivation

Versicherungen abschließen, mit Menschen in Kontakt bleiben, wichtige Überweisungen tätigen, Artikel online bestellen und vieles mehr. Fast alle Dienstleistungen werden heutzutage online angeboten und damit einhergehend steigt auch die Anzahl an Nutzern dieser Online-Dienste. Gut 95 % der deutschen Bevölkerung ist online aktiv und weltweit gesehen sind es rund zwei Drittel. [27] [38] Die vielen Vorteile, die diese Dienste mit sich bringen, liegen auf der Hand. Sie sind fast immer und von überall aus erreichbar, das Angebot ist groß und es verbindet Menschen. Allerdings weisen sie auch Schattenseiten auf. Menschen werden auf eine ganz andere Art angreifbar, indem böswillige Personen versuchen, das Online-Nutzungsverhalten von anderen Menschen auszunutzen. Ein häufig zu beobachtender Angriffsvektor besteht in dem Versuch, sich Accountdaten zu verschaffen, um im Anschluss unter Nutzung dieser Daten im Namen der betroffenen Person Aktivitäten durchzuführen. Um das zu schaffen, gibt es verschiedene mögliche Wege.

Klassischerweise bestehen Accountdaten aus einem Nutzernamen oder E-Mail-Account und einem Passwort. Nutzernamen und E-Mail-Accounts sind teilweise öffentlich einsehbar oder können mittels *Social Engineering* ermittelt werden. Bei Passwörtern ist ein beliebter Angriffsvektor die *Phishing*-Mail, welche einen vermeintlich seriösen Inhalt hat, jedoch den Hintergrund des Datendiebstahls versendet wird. Ein weiterer, aber weniger effektiver Weg, ist das sogenannte *Brute-Force*-Verfahren. Dabei wird versucht, die Accountdaten zu erraten z. B. mithilfe des Tools `metasploit`. Eine alternative Vorgehensweise besteht darin, den Passwort-Hash beim Versenden an den Server abzufangen, was als *Man-in-the-Middle*-Angriff (MITM) bezeichnet wird. Dabei kann der Hashwert entweder bereits ausreichen, um sich bei einem Online-Dienst zu authentifizieren oder dieser muss erst entschlüsselt werden. Entschlüsseln steht hierbei für das Umwandeln von einem Hash-Wert in einen Klartext. Hierbei darf das Passwort allerdings nicht zu komplex oder lang sein, da Hashwerte ab einer gewissen Passwortkomplexität mit aktuellen Technologien nicht zu brechen sind. Für das sogenannte Brechen existieren eine Reihe an Tools, wie z. B. `AirCrack`, `Hashcat` und `THC Hydra`. Gerade bei einfach vergebenen Passwörtern, wie ‚passwort' oder ‚123456789', dauert das Erraten meist nicht lange. Statistisch gesehen war 2023 in Deutschland ‚12345678' das am häufigsten verwendete Passwort, obwohl bereits vom Bundesamt für Sicherheit und Informationstechnik (BSI) Vorschläge

zum Erstellen eines sicheren Passworts existieren. [1] [44] Würden alle Menschen diesen Bestimmungen folgen, so würde die Wahrscheinlichkeit drastisch sinken, dass Angreifer an die Passwörter kommen, da mit steigenden Sicherheitsbestimmungen die Komplexität der Passwörter steigt. Allerdings ist dies ein unrealistisches Szenario und der technologische Fortschritt schreitet ebenfalls voran und die damit verwendeten Technologien zum Stehlen oder Brechen von Passwörtern. Ebenfalls ist der Einsatz von *Phishing*-Mails gängig und macht jedes noch so komplexe Passwort bei einem erfolgreichen Angriff nichtig. [2] Um diese Art von Datendiebstahl zu verhindern, wurde der Standard *Fast Identity Online* (FIDO) entwickelt.

Wie der Name bereits verrät, geht es bei diesem Standard um das Erstellen einer Online-Identität eines Nutzers für einen Dienst. Jede Identität stellt ein eindeutiges Schlüsselpaar dar, das zum Authentifizieren bei einem bestimmten Online-Service -oder Produkt eingesetzt wird und soll langfristig Passwörter abschaffen oder ergänzen. Nach einer anfänglichen Registrierung können sich Nutzer bei einem FIDO-kompatiblen Anbieter authentifizieren, in Kombination mit einer PIN, einem Fingerabdruck oder einer Gesichtserkennung - abhängig von der Technologie des verwendeten Geräts und der des angesprochenen Servers. Die gesamte Kommunikation findet verschlüsselt statt, basierend auf dem *Public-Key*-Verfahren. [15] Das bedeutet, zum einen findet die Kommunikation zwischen Client und Server auf einem verschlüsselten Kanal statt. Zusätzlich wird beim Registrieren des Clients ein Schlüsselpaar erzeugt, welches für alle zukünftigen Authentifizierungen mithilfe von FIDO verwendet wird. Zum Speichern des privaten Schlüssels gibt es z. B. externe Sicherheitsschlüssel, wie dem *YubiKey*. [13] Es ist ebenso denkbar, sich einen digitalen Speicherort auf dem jeweiligen verwendeten Gerät anzulegen.

Diese Arbeit beschäftigt sich mit der Integration von FIDO in das *Transport Layer Security* (TLS) Protokoll. Die Aufgabe von TLS ist es, verschlüsselte Nachrichten, und teils auch authentifiziert, auf einem Kanal versenden zu können. Dieser Ansatz wird aus dem Grund heraus verfolgt, um Browser-unabhängig FIDO als Authentifizierungsmethode einsetzen zu können. Dafür wird ein Konzept vorgestellt und anschließend folgt darauf eine *Proof-of-Concept* (PoC) Implementation. Diese wurde vollständig in *Java* geschrieben und in die *Java Secure Socket Extension* (JSSE) integriert, welche Bestandteil der Standard-*Java*-Bibliothek ist.

2. Technische Grundlagen

FIDO ist ein Standard zur sicheren Authentifizierung – entweder als passwortlose Alternative zum herkömmlichen Passwort oder für den Einsatz eines zweiten Faktors. [18] [3] Dieses Kapitel wird den Aufbau und die Funktionsweise von FIDO beleuchten, welche Vor- und Nachteile er mit sich bringt und inwiefern FIDO das Authentifizieren verglichen zu herkömmlichen Methoden sicherer macht. Da FIDO in TLS integriert werden soll, wird zuerst darauf eingegangen, was TLS ist und wie genau dort Erweiterungen eingebaut werden können.

2.1. Transport Layer Security Protokoll

Das *Transport Layer Security* (TLS) Protokoll stellt einen sicheren Kanal zur Kommunikation zwischen zwei Entitäten her. [41] Dabei kann es sich um beliebige Arten von Kommunikationsmodellen drehen, beispielsweise um einen Client-Server- oder Peer-to-Peer-Kanal. Es gibt viele Anwendungsmöglichkeiten, in denen eine sichere Verbindung unter Einsatz von TLS hergestellt wird. Dazu zählen: HTTPS, VPN-Clients, POP3S, SMTPS und viele mehr. TLS ist das Nachfolgeprotokoll von *Secure Socket Layer* (SSL) und ist im ISO/OSI-Schichtenmodell der Schicht 5, der Sitzungsschicht, zuzuordnen. Nach dem TCP/IP-Modell befindet sich das Protokoll zwischen der Anwendungs- und Transportschicht. Seither wurde TLS stetig verbessert und gehärtet. Um dem neusten Standard zu entsprechen, wird in dieser Arbeit ausschließlich TLSv1.3 betrachtet. [40]

2.1.1. Aufbau

TLS baut auf einem darunterliegenden zuverlässigen und geordneten Datenstrom auf. [40] Dabei sollte TLS folgende Eigenschaften aufweisen:

- **Authentifizierung**: Die Serverseite ist immer authentifiziert, wohingegen die Clientseite authentifiziert sein kann. Bei der Authentifizierung kann es sich um eine asymmetrisches Verfahren handeln oder es baut auf einem symmetrischen *Pre-Shared Key* (PSK) auf.
- **Vertraulichkeit**: Die ausgetauschten Datenpakete sind nur den beteiligten Kommunikationspartner zugänglich.
- **Integrität**: Daten, welche über den Kommunikationskanal gesendet werden,

können nicht von Angreifern modifiziert werden, ohne dass diese Modifikation gemeldet wird.

Diese Eigenschaften sollte auch in dem Fall gelten, in welchem das gesamte Netzwerk kompromittiert wurde. TLS besteht aus zwei Hauptkomponenten:

- **Handshake Protokoll**: Innerhalb dieses Protokolls werden die Kommunikationspartner authentifiziert, kryptografische Modalitäten und Parameter werden ausgehandelt und es wird das gemeinsame Schlüsselmaterial festgelegt.
- **Record Protokoll**: Hierbei wird der Datenverkehr zwischen den Kommunikationspartner geschützt, durch Nutzung der Parameter, welche mit dem Handshake Protokoll ausgemacht wurden. *Record* steht in diesem Fall für einen Eintrag, welcher bestimmte Informationen über den Datenverkehr beinhaltet. Diese Einträge werden mithilfe der zuvor generierten Schlüssel gesichert.

Andere übergeordnete Protokolle können auf TLS aufbauen, jedoch gibt TLS nicht vor, wie ein TLS *Handshake* initiiert und wie die Authentifizierung interpretiert werden soll. Dies kann je nach Implementierung selbst designt werden.

2.1.2. Der TLS Handshake

Nachfolgend wird das Grundprinzip des TLS *Handshakes* näher betrachtet. Abbildung 1 soll einleitend dessen Ablauf skizzieren. Die Bedeutung der farblich

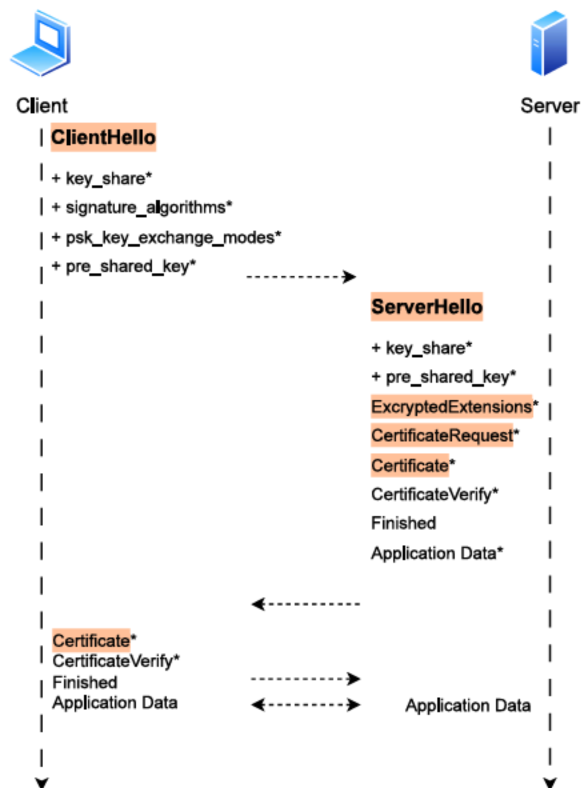

Abbildung 1: TLS *Handshake* mit Hervorhebung erweiterbarer Nachrichten
Quelle: In Anlehnung an [40]

markierten Teile des *Handshakes* werden in Unterunterabschnitt 2.1.3 näher erläutert. Die mit einem "+" gekennzeichneten Nachrichten sind speziell hervorgehobene Erweiterungen und bei den mit einem "*" gekennzeichneten Nachrichten handelt es sich um optionale Erweiterungstypen. Zu Beginn soll das Schlüsselmaterial für die *Shared Keys* zwischen Client und Server ausgetauscht, sowie kryptografische Parameter ausgewählt werden. Das bedeutet vom Client ausgehend wird eine *ClientHello*-Nachricht gesendet. Diese Nachricht beinhaltet unter anderem eine Liste von symmetrischen Chiffren und ein Set von Diffie-

Hellmann *Key Shares* und/oder ein Set von PSK Labeln. Durch die Einführung von *Key Shares*, was erst ab Version 1.3 möglich wurde, werden nun viele Teile des Handshakes verschlüsselt. Der Austausch der *Key Shares* zwischen Client und Server wird in diesem Fall auch als ein Diffie-Hellman-Schlüsselaustausch bezeichnet (siehe Abbildung 2). Das bedeutet, beide Parteien generieren jeweils

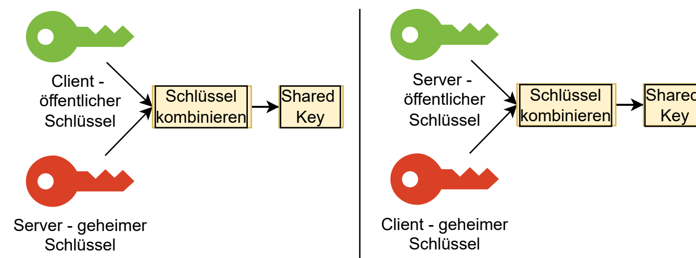

Abbildung 2: Diffie-Hellman-Schlüsselaustausch
Quelle: Eigene Darstellung

eine Liste von *Key Shares*, wobei ein *Key Share* ein Schlüsselpaar ist, bestehend aus einem öffentlichen und privaten Schlüssel. Beide Parteien tauschen dann ihren jeweiligen öffentlichen Teil des *Key Shares* aus um mithilfe dieser das gemeinsame Geheimnis zu erzeugen. Im Anschluss können mit dem Geheimnis Schlüssel für die Ver- und Entschlüsselung von *Handshake*-Nachrichten erzeugt werden. [5] [39] [43]

Der Server verarbeitet die *ClientHello*-Nachricht und bestimmt die geeigneten kryptografischen Parameter für die Verbindung. Darauf aufbauend antwortet der Server mit einer *ServerHello*-Nachricht, welche die ausgehandelten Verbindungsparameter angibt. Das bedeutet, dass ebenso das serverseitige Schlüsselmaterial erzeugt und damit die gemeinsamen Schlüsselpaare bestimmt werden können. Sobald jeweils die *ClientHello*- und *ServerHello*-Nachrichten ausgetauscht wurden, sind alle darauffolgenden Nachrichten verschlüsselt. Das gilt nicht nur für die Nachrichten nach Abschluss des *Handshakes*, sondern für die darauffolgenden *Handshake*-Nachrichten. Falls der Server auf *Elliptic Curve Diffie-Hellman* (ECDH) basiert, dann besitzt dieser einen ephemerischen Diffie-Hellmann-*Share*. Ephemerisch bedeutet, dass bei jedem neuen Verbindungsaufbau neues Schlüsselmaterial erzeugt und damit niemals zweimal das selbe verwendet wird. Durch diese Eigenschaft wird *Forward Secrecy* (FS) geschaffen. [39] [28] FS bedeutet, dass falls ein Angreifer in den Besitz eines Sitzungsschlüssels kommt, kann mit diesem Schlüssel nicht auf zukünftige Sitzungsschlüssel geschlossen werden. Dadurch kann die nachfolgende Kommunikation nicht entschlüsselt werden. Falls

der Server eine zertifikatbasierte Authentifizierung fordert, wird der Parameter *CertificateRequest* in der *ServerHello*-Nachricht mitgeschickt. Ist dies der Fall, werden folgende Parameter mitgesendet:

- **Certificate**: Das Zertifikat vom Endpunkt (Server oder Client).
- **CertificateVerify**: Eine Signatur unter Verwendung des privaten Schlüssels, der dem öffentlichen Schlüssel in dem *Certificate*-Parameter entspricht.
- **Finished**: Hier wird ein MAC mitgesendet, wobei diese Nachricht eine Schlüssel-Bestätigung bereitstellt sowie den jeweiligen Endpunkt an die ausgetauschten Schlüssel bindet.

Sobald der Client die Nachricht des Servers empfängt, sendet dieser eine Authentifizierungsnachricht, worin ebenfalls ein *Certificate*-, *CertificateVerify*- sowie *Finished*-Parameter enthalten sind. Damit ist der TLS-*Handshake* abgeschlossen und der Server und Client können authentifiziert, verschlüsselte Nachrichten auf der Anwendungsschicht miteinander austauschen.

2.1.3. TLS Erweiterungen

Handshake-Nachrichten können erweitert werden. [40] TLS-Erweiterungen dienen dazu, zusätzliche Nachrichten verschicken zu können, um so weitere Funktionen in TLS einzubauen. Es sei darauf verwiesen, dass einige Erweiterungen ab der Version 1.3 obligatorisch sind, mindestens jedoch *supported_versions*. Der einzige Grund, weshalb diese nicht in das Kern-TLS aufgenommen wurde, ist, dass die jeweiligen TLS-Nachrichten auch weiterhin mit Versionen unter 1.3 kompatibel sein sollen. Beispiele für Erweiterungen sind auch *server_name*, *client_certificate_type* und *pre_shared_key*. Ein Anwendungsbeispiel für den Gebrauch einer Erweiterung ist folgender: *Session Resumption*. Wurde bereits ein zertifikatsbasierter Handshake ausgeführt und der Client möchte sich über die gleiche *Session* nochmals mit dem Server verbinden, so kommt die Erweiterung *pre_shared_key* zum Einsatz, welche die Funktion *Session Resumption* ermöglicht. Unter Verwendung dieser Erweiterung, müssen keine Zertifikate mehr untereinander ausgetauscht werden. Sondern es wird ein im ersten *Handshake* vereinbartes Geheimnis zum Authentifizieren des jeweiligen Kommunikationspartners verwendet. So kann Rechenleistung eingespart werden und trotzdem können Client und Server verschlüsselt miteinander kommunizieren. Zu beachten ist, dass in diesem Beispiel TLS 1.3 vorausgesetzt wurde.

Beim Hinzufügen von Erweiterungen werden bereits vorhandene Nachrichten um zusätzliche Nachrichten erweitert. Zu den erweiterbaren Nachrichten gehören: *ClientHello*, *ServerHello*, *EncryptedExtensions*, *Certificate*, *CertificateRequest*, *NewSessionTicket* und *HellyRetryRequest* (siehe Abbildung 1 die farblich markierten Nachrichten). Bei der Verwendung von mehreren Erweiterungen unterliegen diese keiner bestimmten Reihenfolge bis auf eine Ausnahme. Findet die Erweiterung *pre_shared_key* Anwendung in einer Implementierung, so muss diese die letzte sein in der *ClientHello*-Nachricht. Außerdem müssen Erweiterungen zwischen Client und Server verhandelt werden, sodass im Falle einer Erweiterungsanfrage von einen der beiden Entitäten, diese ignoriert werden kann. Allgemein sind Erweiterungen in einer Art Anfrage-Antwort-Form aufgebaut, wobei es auch Anfragen gibt, die keine Antwort erwarten und eher als *Indication* gesehen werden können. Fragt der Client eine Erweiterung an, so schickt er

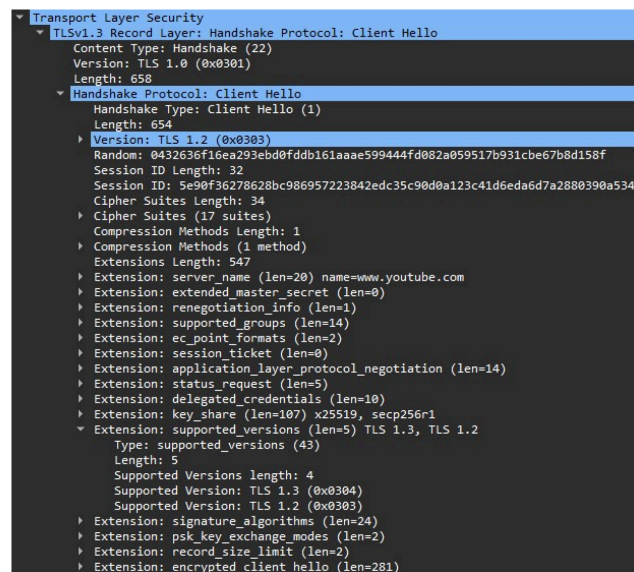

Abbildung 3: *ClientHello*-Erweiterungen beim Aufruf von `Youtube`
Quelle: Bildschirmfoto

diese in dessen *ClientHello*-Nachricht mit. Daraufhin antwortet der Server mit der gleichen Erweiterung entweder in der *ServerHello-*, *EncryptedExtensions-*, *HelloRetryRequest-*, oder *Certificate*-Nachricht. Fragt der Server eine bestimmte Erweiterung an, schickt er die Erweiterung mit der *CertificateRequest*-Nachricht mit. Auf diese Anfrage kann der Client in der *Certificate*-Nachricht antworten. Eine Erweiterung und damit eine Anfrage dieser Erweiterung ist folgendermaßen aufgebaut:

```
struct {
   ExtensionType extension_type;
   opaque extension_data<0..2^16-1>;
} Extension;
```

Der `extension_type` ist ein eindeutiger und einmalig vergebener Zahlenwert und in `extension_data` sind die zu versendenden Pakete als Bytes enthalten. [40] In der nachfolgenden Abbildung ist ein Ausschnitt aus `Wireshark` zu sehen. `Wireshark` ist ein Programm, welches Einblicke in den Netzwerkverkehr gibt und die darin enthaltenen Pakete granular darstellt. Beim Aufruf einer TLS-fähigen Webseite, wie `Youtube`, wird zu Beginn eine *ClientHello*-Nachricht verschickt. Dazu werden mehrere *Extensions* mitgeschickt, die in Abbildung 3 aufgelistet zu erkennen sind. In diesem dargestellten Format werden Erweiterungen auch bei anderen TLS-Nachrichten mitgeschickt. Dabei ist zu erwähnen, dass `Wireshark` nur bereits fest integrierte Erweiterungen auch namentlich mit anzeigt.

2.2. Fast Identity Online

Entwickelt wurde FIDO von der *FIDO Alliance* und formalisiert folgende Technologien: *Universal Authentication Framework* (UAF), *Universal Second Factor* (U2F) und FIDO2. [15] [16] Das Ziel der *FIDO Alliance* ist die Einführung einer sicheren und starken Authentifizierung, die folgende Aspekte umsetzen soll:

- **Passwortfreie Authentifizierung:** Entwicklung offener, skalierbarer Mechanismen zur passwortfreien Authentifizierung von Online-Nutzern
- **Globale Branchenprogramme:** Durchführung von Branchenprogrammen, um die weltweite Übernahme der Spezifikation zur gewährleisten
- **Standardisierung technischer Spezifikationen** Einbettung ausgereifter technischer Spezifikationen bei anerkannten Organisationen für die Entwicklung von Standards zur formalen Standardisierung.

Hierzu wurden die oben genannten FIDO Technologien entwickelt. UAF und U2F wurden fast zeitgleich im Jahr 2014 veröffentlicht, ein Jahr später gefolgt von FIDO2. Alle drei Technologien verwenden das sogenannte *Challenge-Response*-Verfahren.

Challenge-Response-Verfahren

Die *Challenge-Response*-Authentifizierung ist im Allgemeinen ein Verfahren, bei dem eine Entität einem Verifizierer gegenüber beweist, dass es über ein bestimmtes Geheimnis verfügt. [33] Dazu sendet der Verifizierer eine zufällige und damit einmalige *Challenge* an den Antragsteller, der seinerseits eine *Response* erzeugt, indem er eine geheimnisabhängige Funktion auf die *Challenge* anwendet. Begonnen wird mit der Registrierungsphase. Der Nutzer verlässt sich auf ein vertrauenswürdiges Authentifizierungsgerät sowie einen Client. Das Authentifizierungsgerät ist ein Sicherheitsgerät und der Client ein Browser. Der Server sendet über den Client hinweg eine Zufallsanfrage, also die *Challenge*, an das Gerät. Daraufhin wird ein neues Schlüsselpaar erzeugt und eine *Response* generiert. Die *Response* wird anschließend mit dem geheimen Schlüssel signiert und zusammen mit dem öffentlichen Schlüssel an den Server zurückgeschickt. Für alle weiteren Verifikationen soll der öffentliche Schlüssel verwendet werden. Das bedeutet, jedes Mal, wenn sich der Benutzer neu authentifizieren will, schickt der Server eine *Challenge* und die *Response* wird vom Authentifikator mit dem geheimen Schlüssel signiert. Der Server überprüft die Signatur mit dem öffentlichen Schlüssel und bei erfolgreicher Überprüfung gilt der Nutzer als authentifiziert. [4] [3]

Universal Authentication Framework

Das UAF ist ein Protokoll, welches die Möglichkeit einer passwortlosen oder Multi-Faktor-Authentifizierung (MFA) bietet. Nutzer müssen über ein persönliches Gerät, wie einem Computer oder Smartphone verfügen und dieses bei einem Webportal registrieren. Innerhalb des hier thematisierten Prozesses besteht für den Benutzer die Möglichkeit, eines der vom Server akzeptierten Verfahren auszuwählen. Dies kann beispielsweise die Stimmenerkennung, das Einlesen eines Fingerabdrucks oder die Eingabe einer PIN umfassen. Nach erfolgreicher Registrierung können die Benutzer mithilfe ihrer gewählten Methode, anstelle eines Passwort, anmelden. [14] Für die Registrierung wird der Benutzer dazu aufgerufen, sich für eine Methode zu entscheiden. Anschließend erstellt das verwendete Gerät ein Schlüsselpaar, welches individuell auf dieses Gerät, den aktuellen Webdienst und das Benutzerkonto generiert wird. Wichtig zu erwähnen, ist, dass das Schlüsselpaar keine Rückschlüsse auf die genannten Abhängigkeiten ziehen lässt. Auf dem Gerät wird der private Schlüssel gespeichert und der öffentliche Schlüssel wird serverseitig gespeichert. Es handelt sich um ein asymmetrisches Kryptoverfahren. Bei der Authentifizierung sendet der Server eine *Challenge*

an das Endgerät, woraufhin dieses mit dem privaten Schlüssel die *Challenge* signiert. In die Signatur, also der signierten *Challenge*, geht ebenfalls die ID des Servers mit ein, sodass eine Signatur immer an eine korrekte Server-ID gebunden ist. Somit kann auch sicher gestellt werden, dass sie Signatur zum anfragenden Server passt. Dieser Wert wird zurück an den Webserver geschickt, woraufhin dieser den empfangenen Wert mithilfe des öffentlichen Schlüssels verifiziert. Im Rahmen der Prüfung durch den Server wird zudem sichergestellt, dass die Signatur korrekt an die jeweilige Server-ID gebunden ist. Auf diese Weise wird gewährleistet, dass die Signatur tatsächlich für den Server erstellt wurde. Das macht im Kern den *Phishing*-Schutz aus.

Universal Second Factor

Das U2F-Protokoll ergänzt die herkömmliche passwortbasierte Authentifizierung anstatt sie zu ersetzen. Dabei wird zusätzlich ein Token verwendet, welcher entweder über USB, NFC oder Bluetooth mit dem Client interagiert und anschließend mittels HTTPS mit dem entsprechenden Webserver kommuniziert. Wie auch bei UAF, muss der Token erst einmal registriert werden. Hierfür müssen zuerst die Zugangsdaten eingegeben werden und anschließend wird ein Schlüsselpaar erzeugt. Beim Login wird eine *Challenge* an den Token geschickt, welche wiederum mit dem privaten Schlüssel signiert und an den Webserver zurückgeschickt und mit dem öffentlichen Schlüssel verifiziert wird.

Fast Identity Online v2.0

FIDO2 fasst die neuesten Spezifikationen der FIDO *Alliance* und des *World Wide Web Consortium* (W3C) zusammen: *Client-to-Authenticator* v2.0 Protokoll (CTAP2) und W3C-Standard Web Authentication (WebAuthn). Daraus lässt sich FIDO2 ableiten – die aktuellste Version von FIDO. Wie FIDO2 funktioniert, wird im Folgenden näher betrachtet.

FIDO2 setzt die Verwendung eines Authentifikators voraus. Dieser muss FIDO2 unterstützen. Damit kann sich ein Nutzer auf mehreren Servern registrieren und anschließend authentifizieren. Bei jeder Registrierung wird ein Schlüsselpaar generiert, welches für alle folgenden Authentifizierungen verwendet wird. Anhand dieses Schlüsselpaares und damit einhergehend weiterer Prozesse kann sich die verwendete Komponente bei einer Webseite authentifizieren. Auf dem Gerät wird der private Schlüssel gespeichert und auf dem Webserver der öffentliche Schlüssel.

Dabei besteht der FIDO2 Standard aus zwei Kernkomponenten, welche in Abbildung 4 und Abbildung 5 dargestellt sind. Die klassische Client-Server-Kommunikation

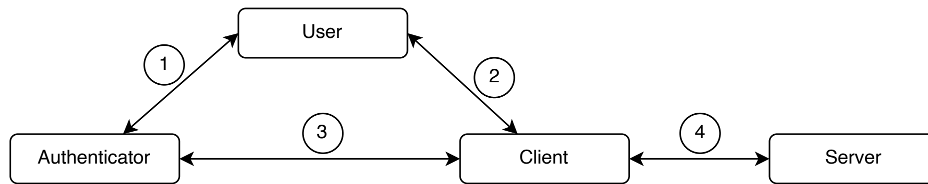

Abbildung 4: Kommunikationskanäle
Quelle: In Anlehnung an [3]

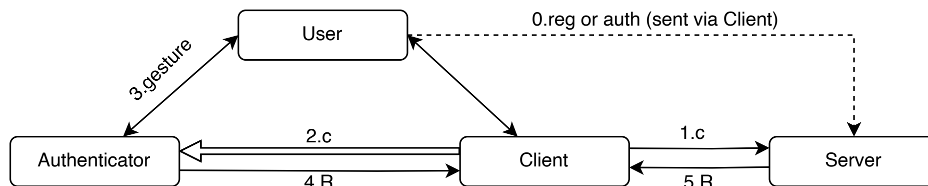

Abbildung 5: FIDO Ablauf
Quelle: In Anlehnung an [3]

wird um einen Authentifikator erweitert, womit insgesamt vier Kommunikationspartner vorhanden sind (siehe Abbildung 4). Hierbei kann es sich beispielsweise um einen externen Sicherheitsschlüssel handeln, welcher den Client beim Server authentifiziert. Es werden auch zusätzliche Nachrichten ausgetauscht zwischen den verschiedenen Entitäten (siehe Abbildung 5). Diese Art der Authentifizierung wird durch WebAuthn ermöglicht. WebAuthn ist eine API, welche in den Browser mit eingebunden wird, damit Client und Server miteinander FIDO2-Nachrichten austauschen können. Diese Nachrichten lassen sich grob in zwei Kategorien einteilen: *Challenge* und *Response*. Unter Anwendung des *Challenge-Response*-Verfahrens kann WebAuthn passwortlos oder unter Nutzung von zwei Faktoren einen Nutzer bei einem Server authentifizieren. In Unterunterabschnitt 2.2.2 wird nochmal näher auf den WebAuthn Standard eingegangen. Damit der Authentifikator und der Client, z. B. ein Browser, miteinander kommunizieren können, wird das Protokoll CTAP2 benötigt. Ziel dieses Protokolls ist es, zu gewährleisten und somit zu beschränken, dass der Client den Sicherheitsschlüssel nur mit der Erlaubnis des Benutzers verwenden kann. Damit der Zugang gestattet wird, fragt der Client den Nutzer nach einer Authentifikator-PIN. Alternativ zum Authentifikator-PIN kann es sich auch um eine Art *Proof-of-Presence*, einem biometrischen Verfahren handeln oder einer Kombination mehrerer Verfahren

handeln. Folglich muss der Benutzer die Authentifikatorschnittstelle verwenden, um Registrierungs- oder Authentifikationsschritte zu autorisieren. CTAP2 legt fest, wie ein Authentifikator mithilfe eines Authentifizierungsverfahren eines Benutzers zu konfigurieren ist. Das Sicherheitsziel besteht darin, einen vertrauenswürdigen Client an den eingerichteten Authentifikator zu binden. Dies geschieht, indem der Benutzer aufgefordert wird, die korrekte PIN (oder ein anderes Verfahren) anzugeben. Nun kann der Authentifikator nur Nachrichten akzeptieren, die von einem bereits gebundenen Client gesendet werden. Es ist außerdem erwähnenswert, dass CTAP auf dem Diffie-Hellman-Schlüsselaustausch beruht, welcher nicht authentifiziert ist. Unterunterabschnitt 2.2.3 verschafft einen tieferen Einblick in den Ablauf von CTAP.

2.2.1. Vor- und Nachteile von FIDO

Die noch heute dominierende Methode, um sich online bei Diensten anzumelden, ist die Nutzung eines Passworts. [48] Das Passwort wird hierbei als eine 1-Faktor-Authentifizierung (SFA, Single-Faktor-Authentifizierung) eingestuft. Dennoch wird mindestens eine MFA empfohlen. Dafür könnte beispielsweise ein Sicherheitsschlüssel oder ein TAN-Generator verwendet werden. Eine neuere Methode stellt dahingegen FIDO2 dar, dessen Vor- und Nachteile im Vergleich zu anderen Authentifizierungsverfahren nachstehend herausgearbeitet werden sollen.

Vorteile
Mit der steigenden Anzahl an Online-Diensten, steigt ebenso die Anzahl der Passwörter. Auf diese Weise tendieren Endnutzer dazu, sich ein unsicheres Passwortverhalten anzueignen, indem sie bspw. das gleiche Passwort bei verschiedenen Diensten verwenden. [36] Im Kontrast dazu steht FIDO2. Es müssen keine Passwörter mehr eingegeben oder auf einen Zettel geschrieben werden. Der gesamte Prozess wird vielmehr durch einen Standard abgewickelt, welcher den Authentifizierungsprozess mithilfe einer kryptografischen Methode löst. Dies erhöht gleichzeitig das Sicherheitsniveau verglichen zur Passwort-Authentifizierung, insbesondere mit Blick auf *Brute-Force*-Attacken oder *Social Engineering*. Wohingegen ein Angreifer für den unerlaubten Zugriff auf ein FIDO2-geschütztes Konto physischen Zugriff auf den Sicherheitstoken benötigt. Ebenso erhöht sich die Benutzerfreundlichkeit, falls FIDO2 nur als Ein-Faktor eingesetzt wird, da kein Passwort mehr eingegeben werden muss. Ebenso gibt es durch den Einsatz kryptografischer Verfahren keine Verbindung zwischen den Nutzerattributen und

dem Authentifizierungsprozess, was einen höheren Schutz der Privatsphäre der Nutzer bewirken kann. Zusätzlich entsteht durch den Einsatz von TLS und die Bindung der $rpID$ an die Signatur *Phishing*-Schutz.

Nachteile
Die gleichen Merkmale, die FIDO2 vorteilhaft machen, bringen verschiedene Nachteile mit sich. Die fehlende Verbindung zwischen den Nutzerattributen und dem Authentifizierungsprozess kann zur Einschränkung des potenziellen Nutzungsbereichs führen. [48] Ein weiterer Nachteil besteht darin, dass es aktuell nur wenige Webservices gibt, die FIDO2 unterstützen. Ebenso muss der Nutzer Kosten für die erstmalige Anschaffung des physischen Sicherheitstoken aufwenden, was die Anwendungsmöglichkeiten faktisch einschränkt. Gerade in Firmen, wo es mehrere hundert Mitarbeitende gibt, von denen jeder einen eigenen Token benötigt, sind die Kosten für die Anschaffung relativ hoch im Vergleich dazu, dass das Anlegen von Passwörtern keine Anschaffungskosten verursacht. Schließlich löst, ausgehend von einer bloßen Passwortnutzung, der Einsatz von FIDO2 als zweiter Faktor bei einer MFA für sich genommen administrativen Zeitaufwand sowie zusätzliche Rechenleistung aus.

2.2.2. Web Authentication

Browser, wie Mozilla Firefox und Google Chrome, haben das *Web Authentication* Protokoll, als wesentlicher Teil von FIDO2, bereits standardmäßig implementiert. 2019 wurde es zum allgemeinen Web Standard und 2021 wurde eine aktualisierte Version des Standards veröffentlicht. [9] Wie der Datenaustausch bei diesem Standard aussieht, ist in Abbildung 6 dargestellt. Zu sehen ist einmal die Registrierungsphase und anschließend die Authentifizierungsphase. [3] [4] [17] Zukünftig kann im Kontext von WebAuthn statt Server auch der Begriff *Relying Party* (RP) verwendet werden. Dieser Begriff bezeichnet die Webanwendung, die die WebAuthn-API zur Registrierung und Authentifizierung von Benutzern verwendet. **Registrierung**:
Die RP besitzt einen eigenen Identifikationsstring id_S, sowie einen öffentlichen Schlüssel vk_T. id_S wurde dem Client bereits über den bestehenden Kommunikationskanal mitgeteilt und ist damit dem Client ebenso bekannt. Diese wird dem Client über die bestehende TLS-Verbindung mitgeteilt. Der Authentifikator besitzt einen Signatur-Schlüssel ak_T sowie einen öffentlichen Schlüssel vk_T. Dieses Schlüsselpaar, welches vom Authentifikator gehalten wird, ist dessen

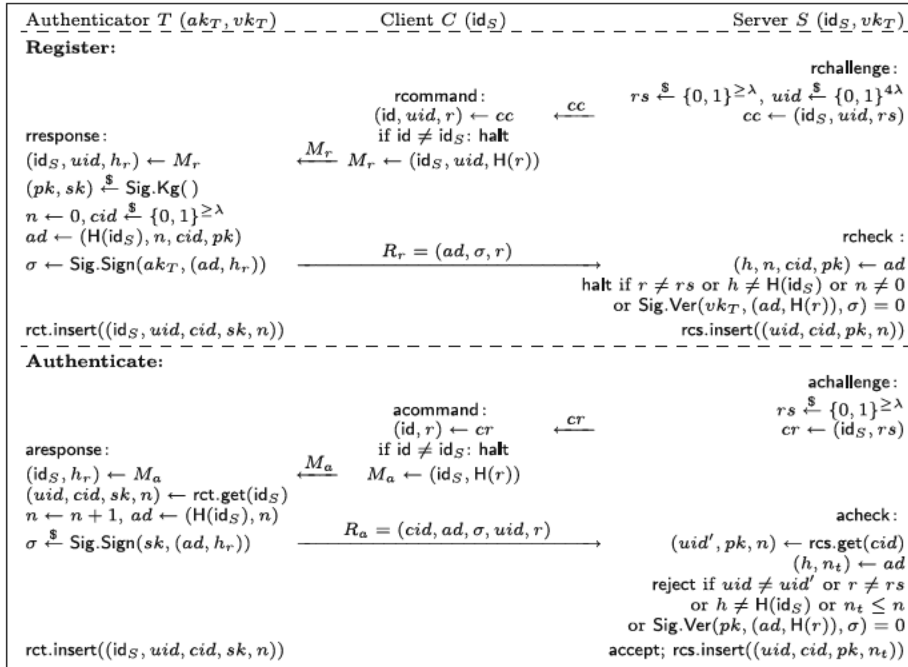

Abbildung 6: WebAuthn Protokoll
Quelle: [3]

einziger Vertrauensanker. Denn anhand dieser Schlüssel ist der Token eindeutig zuordenbar. Zu Beginn schickt die RP eine *Challenge*, eine Nutzer-ID uid und dessen Identifikationsnummer id_S an den Client. Die *Challenge* enthält einen zufällig generierten Wert rs, welcher mindestens 16 Byte groß sein muss. Durch das Einbinden einer *Challenge* können *Replay*-Attacken verhindert werden. Um sicherzugehen, dass es sich um die korrekte RP handelt, greift TLS. Hierbei wird im Rahmen einer sicheren TLS-Verbindung die RP vom Client validiert, unter Einbindung der RP-ID id_S. Das bedeutet, der Client überprüft, ob die empfangene id_S mit der RP-ID übereinstimmt, die dem Client bereits bekannt ist. Anschließend wird die *Challenge* rs, die uid und id_s als Message M_r zum Authentifikator weitergeleitet. Wobei rs unter Nutzung einer Hashfunktion H gehasht wird.

Nach Ankunft der Message M_r muss der Authentifikator mit einer *Response* reagieren. Mithilfe dieser *Response* wird dem Client und damit dann auch der RP mitgeteilt, dass der Authentifikator neue *Credentials* (dt.: Zugangsdaten) generiert hat. In dieser *Response* befinden sich Informationen zu diesen Credentials, die später für das Identifizieren des Authentifikators genutzt werden und um den Authentifikator für zukünftige Authentifizierungen nutzen zu können. Wie genau der Authentifikator die Informationen verarbeitet, ist im Rahmen dieser

Arbeit nicht relevant. Relevant sind aber die Informationen, die am Ende vom Authentifikator zurückgegeben werden, worauf folglich eingegangen wird.

Zuerst wird ein Schlüsselpaar generiert, bestehend aus einem öffentlichen pk und privaten sk Schlüssel. Zusätzlich wird ein Signatur-Zähler n auf 0 initialisiert um möglichen Sicherheitslücken vorzubeugen. Ebenso wird eine *Credential*-ID cid erzeugt. Mit $H(id_S)$, n, cid und pk wird ein 4-Tupel ad gebildet. Erweitert mit dem gehashten *Challenge* $H(rs)$ werden diese Datensätze mit dem Signatur-Schlüssel des Authentifikators signiert, woraus eine eindeutige Signatur σ entsteht. Im Anschluss wird die Signatur σ, ad und rs zurück an den Client gesendet, welcher die *Response* an die RP weiterleitet. Dieser verifiziert die erhaltene *Response* und falls alles korrekt ist, werden die *Credentials* mit bei der RP aufgenommen und für zukünftige Authentifizierungsprozesse gespeichert. Die *Credentials* werden beim Authentifikator ebenfalls aufgenommen.

Authentifizierung:
Bei der Authentifizierung handelt es sich ebenfalls um einen *Challenge-Response*-Nachrichtenaustausch. Ähnlich zur Registrierung, generiert die RP eine zufällig generierte *Challenge* rs und schickt diese zusammen mit der eigenen ID id_S an den Client. Der Client prüft, ob es sich um die korrekten RP handelt und schickt die *Challenge* als Message M_a weiter an den Authentifikator. Der Token inkrementiert den Signatur-Zähler n um eins und generiert mit dem privaten Schlüssel sk eine Signatur über die Werte $H(id_S)$, dem Zähler und $H(rs)$. Diese Signatur wird an den Client gesendet, zusammen mit $H(id_S)$, n, der *Credential*-ID cid sowie der Nutzer-ID uid. Die Nutzer-ID wird in diesem Kontext auch *UserHandle* genannt. Im letzten Schritt leitet der Client die empfangenen Daten an die RP weiter, woraufhin dieser eine Verifikation durchführt. Sobald alle Daten als gültig erklärt worden sind, aktualisiert die RP den Signatur-Zähler für diese cid und der Client hat sich erfolgreich authentifiziert.

WebAuthn bietet dem Nutzer die Möglichkeit, sich entweder mittels eines Nutzernamen zu authentifizieren oder ausschließlich durch die Interaktion mit dem Authentifikator. Unterstützt die RP die Authentifzierung mittels eines Nutzernamen, so handelt es sich um *non-resident Keys*, welche einen Typ von *non-discoverable Credentials* darstellen. Dabei werden die *Credentials* ausschließlich serverseitig gespeichert. Ist dies der Fall, so muss der Nutzer seinen Nutzernamen bei der Authentifizierung mitgeben, sodass die RP die dazu passenden *Credentials* finden kann. Ebenso kann die Authentifizierung ohne die Einbindung eines Nutzerna-

mens ausgeführt werden. Wird dieser Modus von der RP unterstützt, so handelt es sich um *resident Keys*, ein Typ von *discoverable Credentials*. Hierbei muss der Nutzer keinen Nutzernamen mitgeben, sondern schickt ausschließlich dessen *UserHandle* automatisch als Antwort an den Server, woraufhin dieser den *User-Handle* mit dessen gespeicherten uid abgleicht und daraufhin authentifizieren kann.

Sicherheitsabschätzung:
Das WebAuthn Protokoll kann bei erfolgreicher Durchführung Sicherheit gewährleisten. [17] [11] [3] Sobald serverseitig die Registrierung oder Authentifizierung abgeschlossen ist, existiert dazu eine einmalige authentifikatorseitige Sitzung, welche dadurch ebenfalls erfolgreich abgeschlossen wird. So entsteht eine Art Isolierung für die gesamte Kommunikation zwischen RP und Authentifikator. Insbesondere bei der Registrierung, bei der die *Credentials* untereinander zwischen RP und Authentifikator ausgetauscht werden, ist dies ein wichtiger Punkt. Ein weiterer Schutzmechanismus wird durch die Nutzung von TLS integriert, der Client beim Empfang vom Nachrichten der RP die mitgesendete id_S mit dessen bereits bekannten RP-ID vergleichen kann. So können potenzielle *Man-in-the-Middle* (MITM) Angriffe verhindert werden.

Doch WebAuthn weist ein offensichtliches Problem auf. Sobald der Authentifikator nicht mehr erreichbar ist, sind dessen privaten Schlüssel nicht mehr vorhanden. Dieser Fall kann beispielsweise beim Verlust eines Sicherheitsschlüssel auftreten. Dieses Problem könnte gelöst werden, indem ein weiterer Token für die gleiche RP registriert wird als eine Art Backup. Doch auch dieser kann verloren gehen oder missbräuchlich verwendet werden. Dies ist einer der schwerwiegendsten Schwächen von WebAuthn und somit FIDO2 selbst, was die Adoption bei Nutzern stark beeinflusst. [18]

2.2.3. Client-to-Authenticator Protokoll

Das *Client-to-Authenticator* Protokoll, als zweiter wesentlicher Bestand von FIDO2, sorgt für den Datenaustausch zwischen dem Client und dem Authentifikator. In diesem Abschnitt wird näher darauf eingegangen, wie CTAP2 funktioniert und was es für Chancen und Risiken mit sich bringt. Das Client-to-Authenticator Protokoll ist folgendermaßen strukturiert: [12]

- **Authentifikator API:** Jede Operation wird, ähnlich zu einem API Aufruf, definiert. Der Rückgabewert ist entweder eine Ausgabe oder eine Fehlermeldung. Wichtig zu erwähnen ist, dass diese API nur konzeptionell ist. Die tatsächliche API wird von der jeweiligen implementierenden Plattform bereitgestellt.
- **Nachrichten-Kodierung:** Um eine Methode in der Authentifikator-API aufzurufen, muss der Host eine Anfrage erstellen, diese verschlüsseln und anschließend über das ausgewählte Transportprotokoll an den Authentifikator senden. Der Authentifikator verarbeitet dann die Anfrage und gibt eine verschlüsselte *Response* zurück.
- **Transportspezifische Bindung:** Anfragen und Antworten werden über bestimmte Transportmittel (z.B. USB, Bluetooth, NFC) verschickt. Für jedes der Transporttechnologien werden bestimmte Nachrichtenbindungen für dieses Protokoll festgelegt.

Zuerst baut der Client eine Verbindung zu dem Authentifikator auf. Der Client fragt Informationen über den Authentifikator an, um mehr über dessen Leistung, Fähigkeiten etc. zu erfahren. Mithilfe dieser Informationen kann der Client einschätzen, welche Operationen der Authentifikator ausführen kann. Anschließend verschickt der Client einen Befehl zum Ausführen einer Operation über den Nachrichtenkanal. Entweder antwortet der Authentifikator mit einem entsprechenden Rückgabewert oder einem Fehler. [12] [3] Wie bereits in Unterunterabschnitt 2.2.2 erläutert, hängt die Antwort des Authentifikators von der Anfrage der RP ab, d. h. es handelt sich entweder um eine Antwort auf eine Registrierungs- oder Authentifizierungsanfrage. Dabei finden folgende Prozesse statt:

1. Der Benutzer übergibt seine PIN an den Authentifikator.
2. Der Authentifikator sowie der Client speichern sich einen PIN-bezogenen Langzeitzustand. Dabei entsteht ein beidseitiger Bindungsstatus.
3. Der Client kann mithilfe des Bindungsstatus Nachrichten an den Authentifikator schicken.
4. Der Authentifikator verifiziert Nachrichten mit dessen eigenen Bindungsstatus und schickt eine entsprechende Antwort zurück an den Client. Bei den eingehenden Nachrichten handelt es sich entweder um Registrierungs- oder Authentifizierungsanfragen der RP.

Sicherheitsabschätzung:
Um die Sicherheit von CTAP abzuschätzen, genügt es, die genannten Kanäle zur Übertragung von Nachrichten zwischen Client und Authentifikator zu betrachten. Nachdem der Client dessen PIN an den Authentifikator übergibt, generiert der Authentifikator einen dazugehörigen PinToken. Dieser gibt den *PinToken* zurück an den Client. Der Client nutzt anschließend diesen *PinToken*, um zukünftig Antworten vom Authentifikator zu erhalten. Die Problematik von diesem Ablauf besteht darin, dass der *PinToken* für jede Einschaltung einzigartig ist. Das bedeutet, sobald der Client einen *PinToken* erhalten hat und diesen für eine serverseitige Anfrage verwenden möchte, wird bis zur nächsten Anfrage der gleiche *PinToken* verwendet. So können mehrere Clients mehrere Kanäle mit demselben Authentifikator unter Verwendung desselben *PinTokens* aufbauen. Damit ein Kanal zwischen Client und Authentifikator sicher ist, darf keiner der Clients, die während des gleichen Einschaltvorgangs an den selben Authentifikator gebunden sind, gefährdet sein. Sicher bedeutet hier, dass Nachrichten, ausgehend vom Client, nicht gefälscht werden können und damit die Integrität der Nachrichten gewahrt sein muss. Ein Angreifer könnte eine Bindung zu einem Authentifikator initiieren und sich danach als Authentifikator selbst ausgeben, um die PINs, genauer gesagt die Hashwerte dieser PINs, von anderen Clients zu erhalten und diese schließlich verwenden, um den geheimen Authentifikatorseitigen Bindungsstatus zu erhalten. Ein alternativer Angriffsvektor besteht in der Durchführung eines Brute-Force-Angriffs auf alle PINs der Länge x, um den gesuchten PIN zu ermitteln. Hierbei stellt x die Länge der gesuchten PIN dar. Ein weiterer potenzieller Angriffspunkt wäre die Kompromittierung eines Clients, um dadurch den Bindungsstatus offenzulegen.

2.2.4. Sicherheitschätzung von FIDO2

Unter Einschätzung der Sicherheit von WebAuthn und CTAP2, nun die Sicherheit von FIDO2 bewertet werden. Wie zuvor, sind relevante Anknüpfpunkte bei der Sicherheitsanalyse der Authentifikator, der Client, die RP und der Nutzer. Der Nutzer setzt über den Client einen PIN für dessen Authentifikator, wodurch sich der Client an den Authentifikator bindet. Dieser Vorgang wird durch CTAP2 geregelt. Auf diesem Weg wird der Zugangskanal zwischen dem gebundenen Client und dem Authentifikator gestellt. Sobald der Kanal aufgebaut ist, werden unter Verwendung von WebAuthn, die Nachrichten zwischen Authentifikator und

RP ausgetauscht. Der Client dient jeweils als eine Art Brücke zwischen diesen beiden Entitäten (siehe Abbildung 7). Die bisher genannten Sicherheitsaspekte von WebAuthn bleiben erhalten (siehe Unterunterabschnitt 2.2.2), unter der Prämisse, dass der Authentifikator den Client-Zugang zu dessen Credentials über CTAP2 kontrollieren kann. [3]

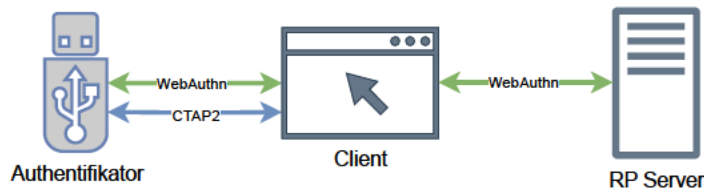

Abbildung 7: Zusammenarbeit zwischen CTAP2 und WebAuthn
Quelle: Eigene Darstellung

Ein weiterer zu betrachtender Punkt ist die Garantie einer Ende-zu-Ende-Registrierung- und Authentifizierung zwischen dem Authentifikator und der RP. Dies ist nur möglich, falls der Client und die RP über einen gesicherten Kanal, wie einer TLS Verbindung, miteinander kommunizieren und sich alle genannten Rollen für eine Authentifizierung den selben Kontext teilen. Da heutzutage TLS standardmäßig verwendet wird, kann hierbei keine Schwachstelle identifiziert werden. Sobald ein Authentifikator bei einer RP registriert wurde, kann die RP zukünftig bei jeder Authentifizierung davon ausgehen, dass es sich um den selben Authentifikator handelt. Diese Aussage kann nur unter dem Umstand getätigt werden, dass zuvor die korrekte PIN eingegeben und der Client nicht kompromittiert wurde. Ein Client gilt als nicht kompromittiert, falls dieser von bösartigem Code isoliert ist, CTAP2 korrekt ausgeführt wurde und ein aktiver Angriff auf den Client über CTAP2 erkannt wird. Solange also der Benutzer im Besitz der PIN ist und eine authentifizierter Kanal zwischen Client und RP existiert, kann sich niemand in dessen Namen authentifizieren. Wurde der Authentifikator über einen ehrlichen Client registriert, kann davon ausgegangen werden, dass alle Authentifizierungen mit ehrlichen Clients bei den korrekten RPs ausgeführt werden. Gerät der Authentifikator jedoch in falsche Hände, so ist dieser immer noch mit einer PIN geschützt. Allerdings wurde eine Schwachstelle in der *Firmware* von *Yubico* identifiziert. [32] *YubiKeys* sind weit verbreitete externe FIDO2-fähige Token, welche nun durch die Abhängigkeit zu einer externen Bibliothek schwachstellenbehaftet sind. Daher kann abgeschätzt werden, dass FIDO2 nach aktuellem Stand als sicher erachtet werden kann, aber der Einfluss bzw. die Zusammenarbeit mit externen Bibliotheken auf Authentifikatoren die Nutzerdaten angreifbar macht.

3. TLS1.3-Erweiterung mit FIDO2

Das passwortlose Authentifizieren soll nicht nur im Browser über HTTPS möglich sein, sondern auch bei Clientanwendungen, wie `OpenVPN`, eingesetzt werden können. Um das zu schaffen, soll FIDO2 in TLS eingebunden werden. Das geplante Vorgehen wird in diesem Kapitel erläutert.

TLS zu erweitern, bedeutet den TLS-*Handshake* zu erweitern. Denn hier werden initial Datenpakete zwischen Client und Server ausgetauscht, die essentiell sind, um später sicher miteinander kommunizieren können. Man stelle sich einen klassischen VPN-Anbieter vor. Dafür wird ein VPN-Server benötigt, bei welchem sich VPN-Clients authentifizieren können um so in einem internen Netzwerk Zugriff auf benötigte Ressource und Dienste bekommen. [46] Die Clients sind außerhalb des Netzwerk lokalisiert und befinden sich üblicherweise auf einem physischen Gerät wie einem Laptop. Anstatt sich über eine grafische Web-Oberfläche beim VPN-Server zu authentifizieren, ist dies auch über eine separate Software möglich – in diesem Fall der Client. Bevor der Client sich authentifizieren kann, wird zuerst eine TLS-Verbindung zwischen den beiden Entitäten aufgebaut. Da bisher über HTTPS das WebAuthn Protokoll für den Austausch der FIDO2-Parameter gesorgt hat, werden an dieser Stelle die Parameter bereits im TLS-*Handshake* übertragen, um so die Authentifizierung des VPN-Clients mittels FIDO2 zu ermöglichen. Auf diese Weise wird WebAuthn nicht mehr benötigt und trotzdem können sich Benutzer bei einem externen Server ohne die Verwendung eines Web-Browser authentifizieren.

Für die kommende Vorstellung eines geeigneten *Handshake*-Modells müssen außerdem Annahmen getroffen werden, um den Rahmen dieser Arbeit einzugrenzen. Es wird von einem vertrauenswürdigem Client ausgegangen und der Kanal zum Authentifikator wird als sicher angesehen. Das bedeutet demnach auch, dass CTAP2 nicht in die Sicherheitsbetrachtung mit einbezogen wird. Ebenso die Verbindungen zu möglichen Datenbanken werden als sicher vorausgesetzt. In Bezug auf den Server gelten die gleichen Voraussetzungen. Verbindungen vom Server zu einer anderen Entität, ausgeschlossen der Client, werden als sicher angesehen. Dazu zählen auch die zu dem Server verbundenen Datenbanken. Der Fokus liegt ausschließlich auf der Verbindung zwischen Client und Server. Außerdem wird im Zuge dieser Arbeit ausschließlich eine Erweiterung mit *resident Keys* implementiert, weshalb der Fokus auf diesen liegt.

3.1. Theoretische Erwägung

Tom-Lukas Johann Breitkopf und Mario Freund haben bereits eine Abschlussarbeit zu einer TLS-Erweiterung mit FIDO2 geschrieben. Innerhalb der Projekte wurde diese Idee einmal in der Programmiersprache *Python* umgesetzt, indem die Bibliothek und einmal in der Programmiersprache *C* unter Erweiterung der Bibliothek. [24] [19] Das Konzept für die Umsetzung dieser Arbeit beruht auf den Ansätzen der beiden genannten Arbeiten und wird im darauffolgenden Unterkapitel erklärt. Neben diesen Arbeiten existieren zahlreiche weitere Projekte, die FIDO2 mit verwenden. Diese sind jedoch für das Konzept dieser Arbeit nicht relevant. Beide Autoren haben sich ausschließlich mit der Authentifizierung auseinander gesetzt. Diese wurde in dieser Arbeit ebenso umgesetzt, jedoch mit einigen Modifikationen. Zusätzlich wird auch die Registrierung eines FIDO2-fähigen Tokens betrachtet. Die Vor- und Nachteile einer solchen Erweiterung werden folglich gegenübergestellt.

3.1.1. Vorteile

Breitkopf hat in seiner Arbeit ein Konzept erarbeitet, welches die FIDO2-Parameter in den TLS-*Handshake* einbindet. Er hat mehrere Ansätze kritisch betrachtet und am Ende eine Möglichkeit gefunden, dies zu erreichen. Dabei ist er nicht nur auf die Funktionsweise eingegangen, sondern die Sicherheit und die Nutzerfreundlichkeit flossen mit in seine Entscheidung ein. FIDO2 in den TLS-*Handshake* einzubinden, bringt einige Vorteile mit sich. Sobald eine Anwendung TLS unterstützt, ist diese in der Lage, ebenso FIDO2 als Authentifizierungsmethode anzubieten. Dadurch könnte FIDO2 häufiger zum Einsatz kommen, wie im genannten VPN-Beispiel. Außerdem wird es für die jeweiligen entwickelnden Personen leichter gemacht, FIDO2 mit TLS-basierten Anwendungen einzubinden, da durch die Integration in TLS bereits viele Operationen vorhanden sind und nur noch wenige Modifikationen an der Anwendung selbst vorgenommen werden müssen. Auf diesem Weg ist die Einbettung von FIDO2 in Online-Dienste weniger fehleranfällig. Wie auch bei anderen Authentifizierungsmethoden, welche zwischen der Transport- und Anwendungsschicht stattfinden, wird Datenvertraulichkeit, Datenintegrität und Authentizität sichergestellt sowie eine Endpunkt-Authentifizierung. Dabei wäre FIDO2 nicht die erste Authentifizierungsmethode, bei der dieser Fall eintreten würde. Clientzertifikate sind bereits ein fester Bestandteil von TLS und haben den Weg geebnet, Client-Authentifizierungen über TLS einzubinden.

3.1.2. Nachteile

Allerdings bezieht sich die Erweiterung ausschließlich auf TLS1.3, was auf der einen Seite mehr Sicherheit bietet, auf der anderen Seite stark einschränkend sein kann, da Anwendungen, welche TLS Versionen unter 1.3 verwenden, diese Erweiterung nicht verwenden können. Außerdem steigt das Risiko für Schwachstellen durch das Einfügen neuer Erweiterungen in TLS. Gerade bei komplexen Operationen, wie im Falle von FIDO2, muss darauf geachtet werden, keine Sicherheitslücken einzubauen. Mit Blick auf die Privatsphäre der Endnutzer kann dies starke Nachteile mit sich bringen, da sensible Nutzerinformationen innerhalb der Erweiterung zwischen Client und Server verschickt werden. Die Latenz der *Handshakes* ist ebenso von der Erweiterung betroffen und kann zur Einschränkung der Nutzerfreundlichkeit führen. Aus den genannten Gründen muss versucht werden, nur die wirklich notwendigen FIDO2-Parameter zwischen Client und Server zu verschicken und diese auch zusätzlich zu verschlüsseln zum Schutz vor *Cyber*-Angriffen.

3.1.3. Einbettung in TLS

Während dieser Überlegungen kamen mehrere Varianten zustande, wie FIDO2 in den *Handshake* eingebettet werden kann. Diese basieren zum Teil aus den genannten Vor- und Nachteilen der FIDO2-Erweiterung. Dabei bestand der Hauptunterschied dieser Erwägungen in der Anzahl der *Handshakes*. Je nach Art, also ob *resident* oder *non-resident Keys* verwendet werden, lässt dies Rückschlüsse auf die Anzahl der *Handshakes* ziehen. Hierbei kam es zu folgender Unterscheidung: TFE *with Name* (FN-Modus) und TFE *with ID* (FI-Modus). TFE steht für TLS mit FIDO2-Erweiterung. Dieser Name ist ebenso Bestandteil Breitkopfs Abschlussarbeit. Im FI-Modus werden *resident Keys* verwendet, welche clientseitig oder auf dem Authentifikator direkt gespeichert sind. Hierbei wird ein *Indication* Objekt zum Anstoßen der FIDO2-Authentifizierung verwendet. Im FN-Modus werden *non-resident Keys* verwendet, wobei die *Credentials* ausschließlich serverseitig gespeichert werden, weswegen initial der Nutzername vom Client zum Server übertragen werden muss, um die Authentifizierung zu starten. Für beide Modi kamen mehrere Ansätze in Frage, wobei diese sich in folgende Gruppen unterteilen ließen:

Einfacher *Handshake* mit statisch verschlüsseltem Nutzernamen

- Der Server besitzt einen öffentlichen Langzeit-Schlüssel mit welchem clientseitig die zu verschickenden Daten verschlüsselt werden können.
- Der *ClientHello*-Nachricht wird entweder das *Indication* Objekt oder der verschlüsselte Nutzername hinzugefügt.
- Anschließend finden server- und clientseitig die entsprechenden FIDO2-Operationen statt.
- Nachteil: Keine FS. Sobald ein Angreifer den privaten Schlüssel berechnet, kann dieser die Anmeldungen der Nutzer nachvollziehen.

Einfacher *Handshake* mit dynamischem Nutzernamen

- Bei der Registrierung sowie allen darauffolgenden Authentifizierungen wird dem Client ein vom Server zufällig erzeugter Nutzername zugeordnet.
- Dieser Name wird zusammen mit der *Challenge* an den Client übertragen.
- Beantwortet der Client die *Challenge* korrekt, so wird der Name auf beiden Seiten final gespeichert.
- Im FI-Modus ist diese Variante identisch zur vorigen.
- Nachteil: Ein Nachteil ist die hohe Komplexität. Der dynamische Nutzername muss clientseitig gespeichert und zwischen verschiedenen Geräten synchronisiert werden.

Erweiterter *Handshake*

- Nutzername oder *Indication* Object werden erst nach der *ClienHello*- und *ServerHello*-Nachricht übertragen.
- Server- und clientseitig werden die jeweiligen FIDO2-Operationen ausgeführt.
- Anschließend werden die *Finished*-Nachrichten übertragen.
- Nachteil: Der TLS-*Handshake* wird um eine Nachrichten-Phase erweitert, was zu mehr Latenz und Komplexität führt. Außerdem ändert dies das Kern-TLS deutlich ab.

Doppelter *Handshake*

- Erster *Handshake*: Versand des Nutzernamen und eines ephemerischen Namen an den Server. Dieser Austausch findet erst im letzten TLS-Schritt

statt, damit der Nutzername verschlüsselt übertragen wird.

- Zweiter Handshake: Der Client autorisiert sich beim Server mit dem ephemerischen Nutzernamen.
- Anschließend finden die FIDO2-Operationen statt, ohne, dass eine weitere Nachrichten-Phase hinzugefügt wird.
- Im FI-Modus muss nur ein *Handshake* ausgeführt werden, da kein Nutzername übertragen werden muss und das *Indication* Objekt keine nutzerspezifischen Informationen beinhaltet.
- Nachteil: Zusätzliche Latenz durch das Nutzen von zwei *Handshakes*.

Post-*Handshake*

- Im *Handshake* selbst wird dem Server lediglich mitgeteilt, dass nach dem *Handshake* die FIDO2-Authentfizierung stattfinden soll.
- Somit findet die Authentifizierung erst nach dem Handshake und damit über den verschlüsselten Kommunikationskanal statt.
- Nachteil: Authentifizierung findet nicht im *Handshake* statt und somit kann sich der Status der Authentifizierung nachträglich noch ändern.

Nach einer kritischen Abwägung hat sich Breitkopf, zumindest im FN-Modus, für den doppelten *Handshake* entschieden. Dieser ermöglicht die Verschlüsselung aller sensiblen Nutzerdaten und garantiert weiterhin alle Schutzziele von TLS, da die Authentifizierung ausschließlich über eine Erweiterung stattfindet und die restlichen TLS-Funktionen unberührt lässt. Die erhöhte Latenz kann sich allerdings negativ auf die Nutzerfreundlichkeit auswirken. Dieser Nachteil ist jedoch im Vergleich zum Verlust von Sicherheit und Privatsphäre des Endnutzers in Kauf zu nehmen. Im Rahmen dieser Arbeit wird dem Beispiel Breitkopfs gefolgt und dieser Ansatz ebenso gewählt für die Implementierung in *JSSE*. Im FI-Modus wurde bisher nur ein *Handshake* verwendet, da kein Nutzername zum Server übertragen werden muss. Allerdings werden vom Server zum Client nutzerspezifische Informationen übertragen, welche nach Breitkopfs Ansatz aktuell noch unverschlüsselt übertragen werden. Daher wird ein Ansatz gewählt, bei welchem der Client vom Server einen symmetrischen Schlüssel zugewiesen bekommt, mit welchem der Client die verschlüsselten Nachrichten des Servers wieder entschlüsseln kann.

3.2. TFE-Handshake

Die folgende *Handshake*-Variante wurde von Tom-Lukas Johann Breitkopf entwickelt und soll als Grundlage für die Implementierung dieser Arbeit dienen. Die bisherige Funktion und die Sicherheitsziele von TLS sollen unberührt bleiben. Das bedeutet, bis auf das Hinzufügen der FIDO2-Erweiterung sollen keine anderen Teile von TLS in dessen Funktion verändert werden. Insbesondere die FS soll bei der Erweiterung enthalten sein und es Angreifern unmöglich machen, dass aus abgefangenen Informationen während des *Handshakes* Rückschlüsse auf zukünftig ausgetauschte Informationen gezogen werden können. Die Erweiterung durch FIDO2 soll ähnlich zu bisherigen TLS-Erweiterungen erfolgen. [40] Es bleibt zu erwähnen, dass die FIDO2-Erweiterung in Kombination mit Clientzertifikaten eingesetzt werden muss, da die letzte Nachricht vom Client zum Server an die *Certificate*-Nachricht angehangen werden muss. So würde es sich um eine MFS handeln. Es spielt allerdings keine Rolle ob die Zertifikate korrekt erstellt worden sind oder es sich nur um *dummy* Zertifikate handelt. Entscheidend ist, dass mindestens die FIDO2-Authentifizierung den Aufbau eines TLS-Kanals autorisieren soll. Es kam die Überlegung auf, die Client-*Response* an die *Finished*-Nachricht anzuhängen, um sich unabhängig von Client-Zertifikaten zu machen. Die *Finished*-Nachricht bietet aktuell keine Funktionalität für Erweiterungen an (Unterunterabschnitt 2.1.3). Somit blieb am Ende nur die *Certificate*-Nachricht zum Erweitern übrig, weshalb Client-Zertifikate eingesetzt werden müssen.

Die nachfolgenden Abläufe orientieren sich an bisherige TLS-Erweiterungen und unterliegen daher einer bestimmten Terminologie, gegeben durch das *Request for Comments* (RFC) 8446. [40] Die Terminologie besteht aus folgenden Schlüsselwörtern:
"MUSS" (*MUST*), "DARF NICHT" (*MUST NOT*), "BENÖTIGT" (*REQUIRED*), "SOLL" (*SHALL*), "SOLL NICHT" (*SHALL NOT*), "SOLLTE" (*SHOULD*), "SOLLTE NICHT" (*SHOULD NOT*), "EMPFOHLEN" (*RECOMMENDED*), "NICHT EMPFOHLEN" (*NOT RECOMMENDED*), "KANN" (*MAY*) und "OPTIONAL" (*OPTIONAL*). Sobald diese Wörter in Großbuchstaben im Text vorkommen, wird ihnen nach *RFC2119* und *RFC8174* eine bestimmte Bedeutung zugeordnet. [6] [34] Im Rahmen der vorliegenden Arbeit werden keine weiteren Fehlermeldungen hinzugefügt, sondern auf die bereits im RFC 8446 vorhandenen zurückgegriffen. Die zu versendenden FIDO2-Nachrichten werden definiert durch den gegebenen WebAuthn-Standard. [9]

3.2.1. Schematischer Aufbau

TFE soll weiterhin die Authentifizierung eines Clients bei einem Server ermöglichen und gleichzeitig müssen beide Parteien die benötigten FIDO2-Parameter mit verarbeiten können. Dabei ist es irrelevant ob der Server selbst die Verarbeitung der Parameter vornimmt oder die Aufgaben an einen externen Server ausgelagert werden. Im Kontext der weiteren Abschnitt wird statt dem dem Begriff erneut der Begriff RP verwendet (siehe Unterunterabschnitt 2.2.2): Der schematische Aufbau sieht folgendermaßen aus (Abbildung 8). Der Client und die RP sind

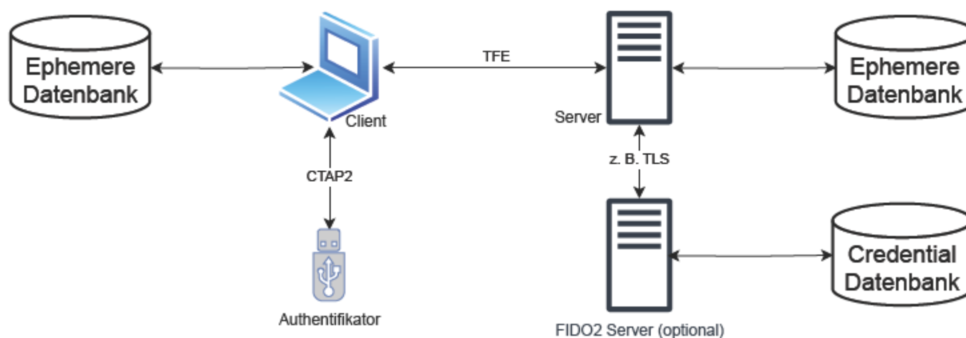

Abbildung 8: Aufbau der Client-Server-Anwendung
Quelle: Eigene Darstellung

jeweils zu einer Datenbank verbunden, welche ephemerische Daten speichern. Das bedeutet, diese können, aber müssen nicht *Session*-übergreifend gespeichert werden und dienen ausschließlich zum zeitweise Speichern von Daten. Der Authentifikator kommuniziert via CTAP2 mit dem Client, welcher wiederum über TLS1.3 mit der RP kommuniziert. Wie bereits erwähnt, können, aber müssen nicht FIDO-spezifische Operationen auf einem externen sogenannten FIDO-Server ausgeführt werden. Findet eine Auslagerung statt, so müssen FIDO- und der RP über einen gesicherten Kanal miteinander kommunizieren. Das kann z. B. über eine gesicherte Verkabelung, wie einer LAN-Leitung in einem abgesicherten Bereich passieren, über einen TLS-Kanal oder einer anderen Methode, welche verschlüsselt ist oder in einem isolierten Bereich angesiedelt ist. Außerdem bleibt es der Implementation überlassen, ob sich die RP oder der FIDO-Server die *Credentials* dauerhaft in einer Datenbank speichern. Um die Latenz zu reduzieren und die Komplexität zu minimieren, ist es empfehlenswert, die Daten auf dem externen Server zu speichern, sofern die Nutzung eines solchen vorgesehen ist.

3.2.2. Authentifizierung mit doppeltem Handshake

In dieser Arbeit liegt der Fokus auf der Authentifizierung im FI-Modus sowie der Registrierung. Daher werden in diesem Kapitel nur *resident Keys* betrachtet. Der erste *Handshake* dient zum Austausch eines symmetrischen Schlüssels und im zweiten *Handshake* werden die FIDO2-Parameter untereinander ausgetauscht. In Abbildung 9 ist der gesamte Ablauf des TFE-*Handshakes* dargestellt und wird mithilfe der folgenden Schritte näher beschrieben. Die farblich markierten Nachrichten heben die FIDO2-Nachrichten hervor. Diese erweitern jeweils die darüber stehende Nachricht, z. B. erweitert *preFIDO* den Nachrichtentyp *ClientHello* und *preFIDORequest* erweitert den Typ *Certificate*. Der symmetrische Schlüssel wird für das Verschlüsseln der sensiblen Nutzer-Parameter auf Serverseite verwendet, welche im zweiten *Handshake* von der RP zum Client übertragen werden. Um den aktuellen Standard zu entsprechen, MUSS *TLS_AES_128_GCM_SHA256* und SOLLTE *TLS_AES_256_GCM_SHA384* und *TLS_CHACHA20_POLY1305_SHA256* für die TLS-Verbindung eingesetzt werden. [40] Um auch bei der Wahl der Verschlüsselung nicht an Sicherheit einzubüßen, wird hierbei auch auf AES im Galois/Counter Mode (GCM Mode) gesetzt. GCM bietet sowohl die Verschlüsselung als auch Authentifizierung von Nachrichten an und ist ein weit verbreiteter Standard zur Verschlüsselung von Daten. [37] [42] Ein wichtiger Hinweis an dieser Stelle ist, dass keine *Session Resumption* eingesetzt werden darf. *Session Resumption* bedeutet, dass ein bereits aufgebauter TLS-Kanal nochmal benutzt werden darf und somit keine Zertifikate mehr ausgetauscht werden müssen. Stattdessen wird ein zuvor definiertes Geheimnis zum Authentifizieren des Clients verwendet. Würde *Session Resumption* aktiviert sein, würden im zweiten *Handshake* keine *Certificate*-Nachrichten ausgetauscht werden und die FIDO2-Authentifizierung könnte nicht stattfinden. Je nach verwendeter Programmiersprache kann dies explizit gesetzt oder die Session kann nach dem ersten *Handshake* invalidiert werden. Anhang B gewährt einen tieferen Einblick in die Nachrichten.

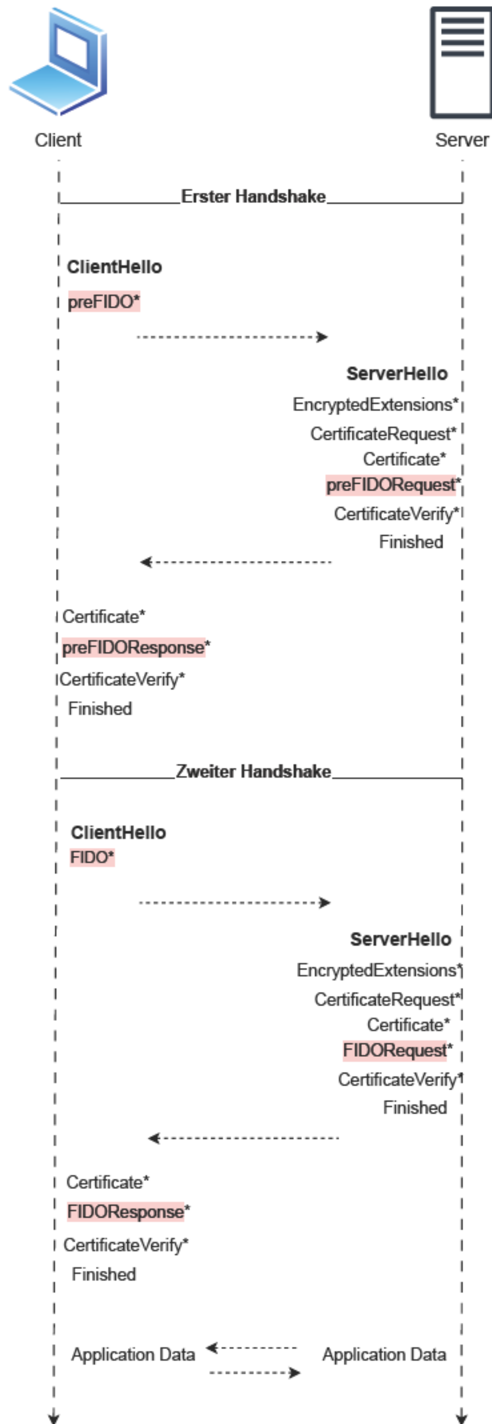

Abbildung 9: Doppelter *Handshake* mit Übergabe der FIDO2-Nachrichten
Quelle: Eigene Darstellung

Kurz zusammengefasst werden für die Registrierung als auch die Authentifizierung folgende Schritte ausgeführt. HS steht für *Handshake* und S für Schritt.

HS1,S0 Der Client signalisiert der RP die Nutzung von FIDO2.

HS1,S1 Die RP generiert eine ephemerische ID sowie einen GCM-Schlüssel und schickt beide Parameter dem Client.

HS1,S2 Der Client schickt der RP dessen Nutzernamen sowie das Ticket im Falle einer Registrierung.

HS2,S0 Der Client schickt der RP die zuvor empfangene ephemerische ID woraufhin die RP diese auf Korrektheit und damit Gleichheit validiert. Damit autorisiert sich der Client gegenüber der RP zum Ausführen weiterer Aktionen (Registrierung oder Authentifizierung).

HS2,S1 Die RP generiert entweder eine Registrierungs- oder Authentifizierungsanfrage, die dem Client verschlüsselt geschickt werden um eine Antwort vom Authentifikator zu erhalten.

HS2,S2 Der Client schickt die Antwort des Authentifikators an die RP, welche bei der RP entsprechend ausgewertet wird.

HS1,S0:
Mit der *HelloClient*-Nachricht verschickt der Client in der *preFIDO*-Erweiterung einen *Indication String* mit, welche der RP signalisiert, dass der Client FIDO2 für die Authentifizierung verwenden möchte. Das *Indication* Objekt beinhaltet keine relevanten Funktionen und kann daher unverschlüsselt an die RP übertragen werden. Diese Nachricht KANN auf "1" gesetzt sein. Unterstützt die RP kein FIDO2, so ignoriert dieser die *preFIDO*-Erweiterung und fährt mit dem Handshake wie gewohnt fort. Schickt der Client keinen *Indication String* und die RP erwartet diesen, KANN die RP den Handshake abbrechen und eine "missing_extension" Fehlermeldung werfen.

HS1,S1:
Für die Übertragung der FIDO2-Nachrichten wurde die *ServerHello*-Nachricht um eine *preFIDORequest*-Erweiterung erweitert. Darin enthalten MUSS einmal ein symmetrischer, von der RP zufällig generierter Schlüssel sowie eine ephemerisch, zufällig generierte ID vorhanden sein. Bei dem symmetrischen Schlüssel handelt es sich um den Schlüssel, welcher für die GCM Ver- und Entschlüsselung verwendet wird. Diese beiden Parameter werden ausschließlich für den darauffolgenden *Handshake* benötigt und werden für jede Authentifizierung neu generiert.

Der Client empfängt die Parameter und speichert sich diese, genauso wie die RP, *Session*-übergreifend ab. Sobald der zweite *Handshake* beendet ist, werden diese wieder client- und serverseitig gelöscht. Ist mindestens einer der beiden Werte leer, MUSS der Client den Handshake mit einer "unexpected_message" Fehlermeldung abbrechen. Wurden beide Parameter vor der RP gesetzt und an den Client verschickt, wird der *Handshake* wie gewohnt fortgesetzt.

HS1,S2:
Im letzten Schritt des ersten *Handshakes*, schickt der Client eine *preFIDOResponse*-Erweiterung der *Certificate*-Nachricht an die RP mit dessen Nutzernamen. Ist dieses Feld leer, so MUSS die RP den Handshake mit einer "unexpected_message" Fehlermeldung reagieren. In diesem Schritt sind die verschickten Daten bereits verschlüsselt, da die jeweiligen *Key Shares* bereits ausgetauscht und der Session-Schlüssel zum Verschlüsseln erstellt werden konnte.

HS2,S0
Der Client verschickt die vom ersten *Handshake* empfangene ephemerische ID an die RP innerhalb der *FIDO*-Erweiterung. Ist der mitgeschickte Wert leer, bricht die RP den *Handshake* mit *unexptected_message* ab. Die RP gleicht diese ID mit seiner zu dem Client gespeicherten ID ab und autorisiert bei erfolgreicher Validierung auf Gleichheit den Client, sich bei ihm zu authentifizieren. Damit ist die RP ebenso in der Lage, den angefragten Client anhand dessen ephemerischen ID zusätzlich zu identifizieren. Ist diese ID nicht korrekt, so kann es sich um einen versuchten MITM-Angriff handeln, bei welchem versucht wird, im Namen des eigentlichen Nutzers sich zu authentifizieren. Würde diese ID nicht überprüft werden, so würde spätestens bei Eingabe der Token-PIN der Angriff fehlschlagen oder schon beim versuchten Entschlüsseln der gesendeten FIDO2-Nachrichten der RP, falls der Angreifer nicht im Besitz des GCM-Schlüssels ist. Ist der Angreifer im Besitz beider Parameter, so benötigt der Angreifer dennoch die Token-PIN sowie physischen Zugriff zum Token selbst.

HS2,S1:
Ist die empfangene Nachricht leer, beendet die RP den *Handshake* mit "unexpected_message". Die in diesem Schritt versendeten Nachrichten werden mithilfe des GCM-Schlüssels verschlüsselt um diese vor möglichen MITM-Angriffen zu schützen. Die RP schickt alle benötigten Parameter an den Client, welche für die Generierung einer entsprechenden *Response* benötigt werden. Das bedeutet, die *FIDORequest*-Erweiterung beinhaltet unter anderem die *Challenge* sowie

optionale Parameter wie die *rpID* der RP. Der Client überprüft zuerst, ob die mitgesendete *rpID* mit dem Domänennamen übereinstimmt, bei welcher sich der Client authentifizieren möchte. Diese Überprüfung findet nicht im *Handshake* statt. Stimmen *rpID* und Domennämen überein, so beginnt der Client Parameter zu generieren, welche via *CTAP2* an den Authentifikator geschickt werden. Dazu gehört unter anderem das *ClientData*-Objekt, welche die *rpID*, *challenge* und *type* beinhaltet. *type* wäre in diesem Fall *webauthn.get*. Der Authentifikator gibt die entsprechende *Response* zurück, welche die Signatur, Authentifikator-Daten sowie den *UserHandle* enthält. In der Erweiterung *FIDOResponse* der *Certificate*-Nachricht schickt der Client die zuvor generierte *Response* an die RP zurück.

HS2,S2:
Beim Erhalt der *FIDOResponse*-Erweiterung MUSS die RP die darin enthaltenen Parameter gemäß FIDO2-Vorschriften auswerten. Ist diese allerdings leer, beendet die RP den *Handshake* mit einer "unexpected_message" Fehlermeldung. Ist die Verifikation erfolgreich, so ist die FIDO2-Authentifizierung abgeschlossen und der TLS-verschlüsselte Kommunikationskanal kann aufgebaut werden.

Sicherheit und Laufzeit
Durch das Verwenden von zwei *Handshakes* sowie das Ausführen von kryptografischen Operationen auf beiden Seiten, erhöht sich die Latenz. Ebenso müssen jeweils auf RP- und Clientseite während des *Handshakes* auf Daten zugegriffen werden, z. B. aus einer Datenbank heraus, was ebenfalls zur Erhöhung der Latenz führt. Dies könnte sich negativ auf die Nutzerfreundlichkeit auswirken. Ebenso muss auch sichergestellt werden, dass die Datenbanken nicht öffentlich zugänglich sind. Außerdem müsste zusätzlich für Redundanz der Daten gesorgt werden, falls einer der Datenbanken ausfällt, kompromittiert wurde oder in einen unsicheren Zustand gerät. Diese Betrachtung ist allerdings nicht Teil dieser Arbeit und sollte eher als weiteres Sicherheitsziel für eine eventuelle reale Anwendung mit berücksichtigt werden. Bezüglich der Sicherheit werden nach bisherigem Ansatz auf allen Wegen des Handshakes die Daten verschlüsselt übertragen. Nicht-sensible Daten, wie der *Indication* String, können unverschlüsselt übertragen werden, da sie keine benutzerspezifischen Informationen beinhalten.

3.2.3. Registrierung

Der Ablauf der Registrierung ähnelt stark der Authentifizierung. Es werden ebenfalls zwei *Handshakes* ausgeführt und teilweise identische Nachrichten, verglichen mit der Authentifizierung, ausgetauscht. Allerdings soll sich nicht jeder beliebige Client bei einer RP registrieren können. In bisherigen Anwendungen, welche bereits passwortlose Authentifizierung anbieten, wird vorausgesetzt, dass der Nutzer bereits einen Account besitzt. Wie beispielsweise bei Github muss der Nutzer bereits einen registrierten Account auf deren Webseite haben und kann im Nachhinein einen FIDO2-fähigen Token zusätzlich registrieren. [26] Das bedeutet, der Nutzer muss also bereit autorisiert sein, um dessen FIDO2-fähigen Token zu registrieren. In einer PoC-Implementation kann dies über ein sogenanntes Ticket bzw. gemeinsames Geheimnis ermöglicht werden. Dabei kann es sich um ein entweder nutzerunabhängiges oder nutzerabhängiges Attribut handeln, was auf beiden Seiten identisch sein muss und autorisiert den Nutzer dazu, sich bei der RP mit einem solchen Token zu registrieren. Wie dieses Ticket auf beide Seiten gelangt, ist erstmal irrelevant für diese Arbeit. Relevant ist, dass das Ticket beidseitig mit Konsens der RP gespeichert wurde. Für das Konzept dieser Arbeit wurde dafür ein einfacher *String* verwendet ohne die Einbindung von Nutzer- oder anderen Merkmalen. Diese Entscheidung kann je nach Belieben getroffen werden, sollte jedoch für jeden Client eindeutig sowie einmalig sein um unbefugte Nutzer auszuschließen. Sobald das Ticket jedoch nutzerspezifische Daten beinhaltet, sollte dieses Ticket verschlüsselt zum Client gelangen. Im Falle der PoC-Implementation ist der Wert des Tickets zufällig gewählt worden ohne Einbindung von Nutzerattributen und kann daher unverschlüsselt zum Nutzer gelangen. Folglich werden die Schritte für die Registrierung eines Tokens vorgestellt. Anhang A gewährt einen tieferen Einblick in die Nachrichten. Begonnen wird mit dem ersten *Handshake*.

HS1,S0+1:
Diese Schritte sind nahezu identisch zu den Schritten 0 und 1 des ersten Handshakes der Authentifizierung. Das *Indication* Objekt muss allerdings ”0” gesetzt sein.

HS1,S2:
Zusätzlich zum Nutzernamen wird hier ebenso das oben erwähnte Ticket mitgeschickt. Ist dieses identisch mit dem bei der RP gespeicherten Ticket zu diesem Client, ist der Client autorisiert den zweiten *Handshake* und damit die Registrie-

rung zu initiieren. Ist der vom Client empfangene Nachricht leer, MUSS die RP den *Handshake* mit einer ”unexpected_message” Fehlermeldung abbrechen.

HS2,S0:
Dieser Schritt ist identisch zum Schritt 0 des zweiten *Handshakes* der Authentifizierung.

HS2,S1:
Auch hier werden, wie bei der Authentifizierung, Pflichtwerte sowie optionale Werte mitgeschickt. Werte, die verschickt werden sollten, sind die *Challenge*, Nutzerdaten, Informationen über die RP sowie Parameter, die beschreiben, wie die neu zu erstellenden *Credentials* generiert werden sollen. Ist der mitgeschickte Wert leer, bricht die RP den *Handshake* mit ”unexpected_message” ab. Diese Werte dienen zum Erstellen einer Antwort auf Clientseite. Ebenso wird in den Nutzerdaten auch die von der RP generierte *userId* mitgeschickt, welche bei der Authentifizierung auch als *userHandle* bezeichnet wird.

HS2,S2:
Dieser Schritt ist ebenfalls nahezu identisch zum Schritt 2 des zweiten *Handshakes* der Authentifizierung. Ist der mitgeschickte Wert leer, bricht der Client den *Handshake* mit ”unexpected_message” ab. Der Client schickt seine Antwort an die RP und dieser speichert sich die neuen *Credentials* in dessen Nutzer-Datenbank. Zusätzlich wird an dieser Stelle auch die *Response* des Clients auf Korrektheit überprüft.

3.3. Diskussion

Die übertragene ephemerische ID und der GCM-Schlüssel werden aktuell unverschlüsselt übertragen. Dies bietet Angreifern die Möglichkeit, diese abzufangen und noch in der selben *Session* die von der RP geschickten Nachrichten als MITM abzufangen und so Nutzerdaten auszulesen. Alternativ dazu, kann ein Angreifer sich auch als den Benutzer vorgeben und so direkt die Nachrichten der RP empfangen. Spätestens beim Signieren der Anfrage kann der Angreifer nicht mehr darauf reagieren, da im Anschluss der *Handshake* abgebrochen werden würde bei einer leeren oder fälschlichen Antwort. Um diese Parameter sicher zum Client zu schicken, könnte hierzu ein weiterer initialer *Handshake* ausgeführt werden, welcher ebenso einen einmaligen Schlüssel generiert, um so die nachfolgenden Parameter, wie der ephemeren ID und den GCM-Schlüssel, sicher zu versenden.

Dies würde allerdings die Latenz weiter erhöhen.

Ebenso kann ein Benutzer beliebig oft einen *Handshake* triggern mit zufälligen Bytes in der FIDO2-Erweiterung. Dies würde einer *Denial-of-Service* (DoS) Attacke gleichkommen. Hierzu könnte eine Art *Lockout* Mechanismus eingebaut werden, der das verhindert. Dies könnte durch Browser-unabhängige *CAPTCHAS* umgesetzt werden.

Insgesamt bietet das Konzept die Möglichkeit, FIDO2 in den TLS-Handshake als Erweiterung einzubauen. Es weist allerdings weiterhin Verbesserungspotential auf in Bezug auf Sicherheit und Rechenleistung.

4. PoC Implementation

Die Implementierung des Konzepts ist vollständig in Java geschrieben. Für die Implementierung gibt es eine *Github Fork* des bestehenden *openJDK Repository* sowie eine dazu angefertigte Client-Server-Anwendung, welche die modifizierte *JDK* nutzt. [22] [23] Wie bereits in Unterabschnitt 3.1 erwähnt, existieren bereits Abschlussarbeiten in den Sprachen *C* und *Python*. Daher schien es sinnvoll, eine PoC-Implementation in einer weiteren, anerkannten und weitverbreiteten Programmiersprache zu entwickeln. Aktuell liegt Java nach dem *TIOBE Index* auf Platz 3 der beliebtesten Programmiersprachen weltweit. [29]

Java ist eine objektorientierte Programmiersprache, auf welche viele angebotene Services basieren. Veröffentlicht wurde die Sprache im Jahr 1995 von *Sun Microsystems* und wurde 2005 von *Oracle* übernommen. Sie ist ein fester Bestandteil der *Java*-Technologien. Zum einen besteht *Java* aus dem *Java*-Entwicklungswerkzeug (JDK, eng.: *Java Development Kit*) zum Erstellen von *Java*-Programmen und zum anderen aus der Laufzeitumgebung (JRE, engl.: *Java Runtime Environment*) zum Ausführen von *Java*-Programmen.

4.1. Verwendete Bibliotheken

Es besteht eine Vielzahl an Möglichkeiten, auf *Java*-Bibliotheken zurückzugreifen, die das TLS-Protokoll unterstützen. Daher musste eine ausgewogene Entscheidung getroffen werden. Die Bibliothek sollte nicht nur TLS-Funktionalitäten anbieten, sondern sollte auch gut dokumentiert sein und die Möglichkeit bieten, weitere Änderungen in einem angemessen Zeitraum einbauen zu können. In Tabelle 1 sind mögliche Kandidaten solcher Bibliotheken aufgelistet. Nach einer kritischen Abwägung fiel die Entscheidung auf *JSSE*. Sie ist bereits standardmäßig in der *JDK* integriert und bietet Funktionalitäten zum Bauen einer Client-Server-Anwendung. Da die Bibliothek bereits im Standard vorhanden ist, ist diese auch gut dokumentiert und bietet die Möglichkeit durch Modifikationen bestehender Klassen Erweiterungen einzubauen. Um jedoch die Bibliothek zu modifizieren, muss die *JDK* selbst heruntergeladen und gebaut werden (siehe Anhang C). Anschließend kann sie in einer weiteren Anwendung als zugrundeliegende *JDK* verwendet werden, damit die Modifikationen dort auch ausgeführt bzw. mit verwendet werden beim Ausführen.

Bibliothek	**Vorteile**	**Nachteile**
JSSE	- Bestandteil der Standard *Java* Bibliothek - Klassen und APIs bis TLS Version 1.3 vorhanden - Gut dokumentiert	- *JDK* muss selbst gebaut werden - Lange Kompilierzeiten
BouncyCastle	- Unterstützt TLS bis Version 1.3 - Bietet auch viele Kryptographie-Funktionen an	- Dokumentation nicht gut strukturiert
Apache HttpClient	- Unterstützt auch HTTPS	- Unterstützt nur clientseitige Funktionalitäten
Netty	- Unterstützt TLS	- Aufwendig zu erweitern - Eher unbekannt - Wenig dokumentiert

Tabelle 1: Kandidaten für eine geeignete *Java*-Bibliothek

Zudem braucht es eine Bibliothek, die FIDO2- und CTAP2-Operationen anbietet. Von *Yubico* selbst wird eine *Java*-Bibliothek angeboten, die genau diese beiden Möglichkeiten bietet. [31] Allerdings basiert die CTAP2-Funktionalität auf *JavaScript* und ist daher Browser-abhängig. Ein möglicher Umbau wäre sehr zeitintensiv. Daher musste eine weitere Bibliothek gefunden werden, die auch Terminal-basiert CTAP2-Funktionen anbietet. Hierfür gibt es keine offiziellen Quellen, daher wurde eine Bibliothek gewählt von einem unbekannten Dritten. [20] Diese Bibliothek bietet sowohl CTAP2- als auch FIDO2-Funktionalitäten. Diese Bibliothek soll ein *Java*-Abbild der offiziellen *Yubico-C*-Bibliothek darstellen. [25] Nach einer Durchsicht scheint die Bibliothek korrekt zu fungieren, allerdings muss hierbei auch darauf vertraut werden, dass diese dem WebAuthn-Standard entspricht. Da der Fokus dieser Arbeit auf der TLS-Erweiterung selbst liegt, ist dieses Risiko in Kauf zu nehmen, bemessen am zeitlichen Rahmen dieser Arbeit. Um Kompatibilitätsprobleme zu vermeiden, wurde vollständig auf die bereits erwähnte Dritt-Bibliothek vertraut für das Verwenden von FIDO2- und CTAP2-spezifischen Operationen. Zusätzlich wurde auch die Bibliothek *BounyCastle* verwendet, um vereinzelt kryptografische Funktionen zu verwenden. Diese wurde ebenso in der oben erwähnten Dritt-Bibliothek eingesetzt. Somit können Kompatibilitätsprobleme ausgeschlossen werden. Für das Bauen der Datenban-

ken wurde die Bibliothek *SQLite* verwendet aufgrund der Einfachheit in der Implementierung. [45]

4.2. Implementierung

Der Aufbau ist der Abbildung 10 zu entnehmen. *TLSClient* und *TLSServer* la-

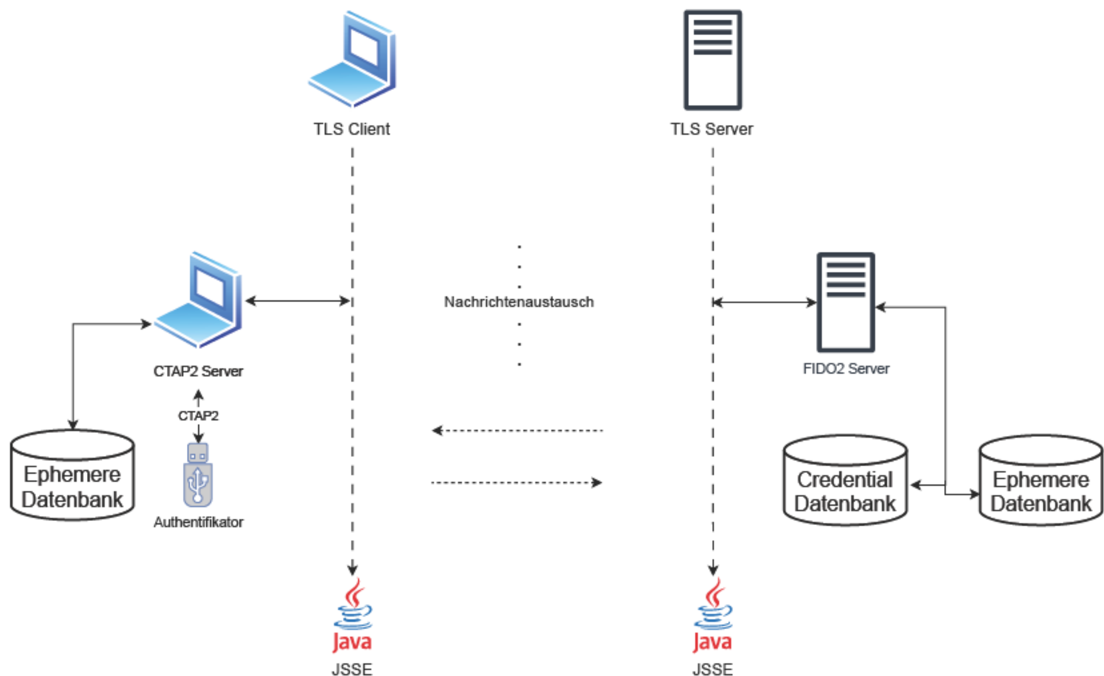

Abbildung 10: Aufbau der *Java*-Implementation
Quelle: Eigene Darstellung

gern jeweils alle Funktionen aus, die FIDO2- oder CTAP2-spezifisch sind. Der *CTAP2Server* wird von der darunterliegenden *JDK* aufgerufen sobald dieser Parameter zum Fortführen des *Handshakes* benötigt oder mit dem Authentifikator interagieren muss. Bei diesen Parametern handelt es sich um die ephemere ID sowie den GCM-Schlüssel. Die getroffene Entscheidung basiert auf der Prämisse, dass die Kommunikation mit dem Authentifikator über eine externe Bibliothek erfolgt und eine Integration externer Bibliotheken in die *JDK* nicht zulässig ist. Dieselbe Argumentation kann auch auf den *FIDO2Server* übertragen werden. Somit beinhalten *TLSClient* und *TLSServer* lediglich die nötigen Aufrufe zum Aufbauen einer TLS-Verbindung sowie nur wenige zusätzliche Modifikation zum Verwenden der FIDO2-Erweiterung. Hinzu kommen die jeweiligen client-

und serverseitigen Zertifikate, die zusätzlich erstellt werden müssen, um die *Certificate*-Erweiterungen nutzen zu können (siehe Anhang D).

Alle Entitäten stellen jeweils einen *Socket* dar, die eine Verbindung untereinander aufbauen. Ein *Socket* stellt einen Endpunkt einer bidirektionalen Kommunikationsverbindung zwischen zwei Programmen dar, welche innerhalb eines Netzwerks ausgeführt werden. [35] Zusammenfassend existieren folgende *Sockets*:

TLSClient: Clientseitiger *Socket* zum Bauen der TLS-Verbindung.

TLSServer: Serverseitiger *Socket* zum Bauen der TLS-Verbindung.

FIDO2Server: *Socket*, welcher alle FIDO2-Funktionen aufruft und implementiert. Dieser *Socket* ist ebenfalls zu der *Credential*-Datenbank verbunden.

CTAP2Server: *Socket*, welcher alle CTAP2-Funktionen aufruft und implementiert. Dieser *Socket* kommuniziert mit dem Authentifikator.

Dabei muss der *TLSClient* folgende TLS-Parameter setzen, damit die FIDO2-Erweiterung genutzt werden kann:

FIDO: Gibt die gewählte Aktion vor. Dieser Wert ist entweder auf "0" (=Registrierung) oder "1" (=Authentifizierung) gesetzt.

TICKET: Wird nur im Falle einer Registrierung gesetzt und beinhaltet einen *String*, der auf Client- und Serverseite identisch sein muss für eine erfolgreiche Registrierung.

USERNAME: Beinhaltet den Nutzernamen.

Der *TLSServer* hingegen setzt folgende Werte:

RPID: Gibt dessen RP ID an.

TICKET: Beinhaltet einen Wert, der auf Client- und Serverseite identisch sein muss für eine erfolgreiche Registrierung.

Zusätzlich zu den genannten TLS-Parametern, setzt der *TLSServer* folgende Werte fest, die für die Generierung der Registrierungs- und Authentifizierunsanfragen benötigt werden:

authenticatorAttachment: Dieser Wert muss auf "cross-platform" gesetzt werden, da die Implementierung nur physische *Hardware*-Token unterstützt.

residentKey: Dieser Wert muss auf "required" gesetzt werden, da die Implementierung die Nutzung von *resident Keys* voraussetzt.

userVerification: Dieser Wert muss auf "preferred" gesetzt werden, da die Implementierung eine Nutzer-Interaktion mit dem Autentifikator voraussetzt.

CredentialAlg: Dieser Wert muss auf "1" gesetzt werden. Dieser Wert steht für die Verwendung des ES256 Algorithmus und ist der einzige, den die Implementierung unterstützt.

Des Weiteren setzen *TLSClient* und *TLSServer* die genutzte *CipherSuite* auf *TLS_AES_128_GCM_SHA256* und das Protokoll auf *TLSv1.3*. Zusätzlich müssen *TLSClient* und *TLSServer* auf dem gleichen Port und auf dem Host *localhost* laufen.

Der *CTAP2Server* und der *FIDO2Server* setzen die *CipherSuite* auf *SSL_DH_anon_EXPORT_WITH_DES40_CBC_SHA*. Der *CTAP2Server* muss zusätzlich eine Verbindung zum *TLSClient* aufbauen über einen gemeinsamen Port. Analog gilt dies für den *FIDO2Server* und den *TLSServer*.

4.3. Einschränkungen

Die Implementation unterliegt gewissen Einschränkungen. Diese beruhen zum Teil aus den Einschränkungen der *Java*-Welt sowie dem zeitlichen Rahmen dieser Arbeit. Die Einschränkungen lassen sich folgendermaßen bündeln:

- **Registrierung und Authentifizierung**: Es können nur *resident Keys* registriert und authentifiziert werden. Zudem kann sich ein Authentifikator nicht für unterschiedliche Nutzer bei einer gleicher *RPID* registrieren und authentifizieren.
- **Nebenläufigkeit**: Der Token muss zuerst registriert und kann dann authentifiziert werden. Dies entspricht der vorgesehenen Reihenfolge, gegeben durch den Webauthn Standard. Daher ist dieser Punkt nicht als Einschränkung zu sehen. Jedoch müssen jeweils bei beiden Vorgängen initiale TLS-*Handshakes* ausgeführt werden. Dies kann zu Einschränkungen führen, falls einer der beiden Entitäten unerwartet ausfällt.
- **Skalierbarkeit**: Die Implementierung lässt nur die gleichzeitige Kommunikation mit einem *TLSClient* zu.

- **Port- und Host-Vorgabe**: *TLSClient* und *TLSServer* müssen auf dem gleichen Port laufen. Darüber hinaus laufen die *Sockets* ausschließlich über den *localhost*. Analog gilt dies auch für den *CTAP2Server* und den *FIDO2Server*.
- **Interoperabilität**: Bei den FIDO2-Nachrichten handelt es sich um serialisierte JSON-Objekte. JSON-Objekte sind zwar in allen Programmiersprachen vertreten, jedoch müssen diese einem festen Schema folgen. Daher werden auch optionale FIDO2-Parameter als leere Werte mitgeschickt.
- **Nachrichten-Pakete**: Durch das Verwenden von JSON-Objekten werden die FIDO2-Nachrichten nicht so klein wie möglich gehalten. Dies fällt zur Last der Rechenleistung sowie der temporären Speicherkapazität während des TLS *Handshakes*.
- **Laufzeit**: Die Laufzeit des FIDO2-erweiterten *Handshakes* unterliegt einer spürbaren Latenz. Dies schränkt die Benutzerfreundlichkeit ein. Zudem kommt für die Verwendung zwei zusätzlicher *Sockets* weitere Latenz hinzu.
- **Sicherheit**: Die *Socket*-Verbindungen zum *CTAP2Server* sowie zum *FIDO2Server* unterliegen unsicheren *CipherSuites*, was diese Kommunikationspunkte zum Angriffsziel macht, um Nutzerdaten abzufangen.
- **Abhängigkeiten**: Wie oben bereits erwähnt, muss auf die Korrektheit der verwendeten Dritt-Bibliothek vertraut werden. Dies kann zu fehlenden FIDO2-spezifischen Operationen führen.

4.4. Diskussion

Während der Implementierung kam es zu weitreichenden Herausforderungen, die teilweise dazu geführt haben, dass die vorliegende Implementation Sicherheitslücken aufweist. Dies bezieht sich ausschließlich auf die Verwendung der zusätzlichen *Sockets* für die Kommunikation zum Authentifikator sowie der auszuführenden Registrierungs- und Authentifizierungsanfragen. Hierbei mussten weitreichende Modifikationen an der *JDK*-Konfiguration vorgenommen werden, da die Kommunikation aus der *JDK* heraus ohne das Verwenden eines bestimmten Authentifizierunsschema erfolgen musste. Zukünftig sollten hierfür ausschließlich Standard-*Java*-Funktionen für FIDO2-spezifische Operationen implementiert werden, um diese zusätzlichen Kommunikationswege zu vermeiden und die Sicherheit zu erhöhen.

Die Verwendung von JSON bietet viele Vorteile. Es kann verschiedene Datentypen sowie Größen von Daten speichern und anschließend zu Bytes serialisiert werden. Allerdings können keine Byte Arrays direkt in JSON-Objekten gespeichert werden und es bietet nicht die Möglichkeit dynamisch und damit optionale Werte zu speichern, da JSON-Objekte einem bestimmten Schema unterliegen müssen. Dies könnte durch die Verwendung von *Concise Binary Object Representation* (CBOR) abgelöst werden. [8] CBOR baut auf JSON auf und bietet Vorteile gegenüber JSON womit Rechenleistung gespart werden kann. Zusätzlich muss bei CBOR kein bestimmtes Schema bestimmt werden, was bei JSON wiederum vorausgesetzt wird.

Letztlich kann die vorliegende PoC Implementation in vielen Bereich weiter verbessert werden in Bezug auf Sicherheit, Effizienz und Benutzerfreundlichkeit. Allerdings waren diese Verbesserungen in Bezug auf den zeitlichen Rahmen dieser Arbeit nicht umsetzbar, sollten jedoch für eine reale Anwendung in Betracht gezogen werden.

5. Fazit

FIDO2 bietet die Möglichkeit einer sicheren Ende-zu-Ende Authentifizierung. [15] [16] Mit Abschluss dieser Arbeit konnte dieser Standard in TLS erfolgreich integriert werden. Die PoC Implementation wurde vollständig in der Programmiersprache *Java* entwickelt, da die Bibliothek *JSSE* die Möglichkeit bietet, beliebige TLS-Erweiterungen einzubetten. Das im Zuge dieser Arbeit entwickelte Konzept beinhaltet nicht nur die Authentifizierung eines FIDO2-fähigen Authentifikators, sondern ebenso die Registrierung eines solchen. Damit wurde das bereits standardisierte TLS-Protokoll um ein passwortloses Authentifizierungsverfahren erweitert. Zudem schränkt die Erweiterung keine der anderen TLS-Funktionen ein, da alle FIDO2-Nachrichten innerhalb der Erweiterung liegen. Die Reduzierung des Risikos eines Datendiebstahls kann somit gewährleistet werden, während gleichzeitig die Möglichkeit besteht, dass sich ein Benutzer Browser-unabhängig mit einem Authentifikator authentifizieren kann.

Eine kritische Betrachtung ergibt, dass das Konzept weiter verbessert und gehärtet werden kann gegenüber potenziellen *Cyber*-Angriffen sowie mit Blick auf Effizienz und Nutzerfreundlichkeit. Zwar bietet es einen Weg, um das Ziel dieser Arbeit zu erreichen, allerdings unterliegt es auch einem bestimmten Verbesserungspotential. Dies gilt in Folge auch für die darauf aufbauende PoC Implementation. Im Rahmen dieser Arbeit konnten zwar alle Anforderungen des Konzepts implementiert werden, jedoch mit gewissen Abzügen, die die Sicherheit sowie die Laufzeit deutlich einschränken. Damit ist zum einen das Verwenden von einem doppelten *Handshake* gemeint, aber auch die Verwendung von zusätzlichen, unsicher konfigurierten Kommunikationskanälen. An dieser Stelle empfiehlt sich eine Aufarbeitung des Konzepts. Ebenso sollte die Implementierung auch alle darin enthaltenen Anforderungen erfüllen und keine zusätzlichen Sicherheitslücken einbauen. Hierfür müssten grundlegende Funktionen in *JSSE* eingebaut werden im Hinblick auf die Kommunikation mit dem Authentifikator sowie weiterer von WebAuthn vorgegebener FIDO2-Operationen. [9]

Wünschenswert ist die Aufnahme der FIDO2-TLS1.3-Erweiterung in den bestehenden TLS Standard. Für diesen Standard existieren bereits zahlreiche Erweiterungen, die bereits in der Praxis Anwendung finden. [40] Für die Erreichung dieses Ziels müssen noch einige Schritte getätigt werden, damit FIDO2 als standardisierte Erweiterung in TLS eingebettet werden kann.

Literaturverzeichnis

[1] *123456789: Das beliebteste Passwort 2023 | Hasso-Plattner-Institut — hpi.de*, https://hpi.de/news/jahrgaenge/2023/123456789-ist-das-beliebteste-passwort-2023-in-deutschland.html, [Accessed 07-10-2024].

[2] *Anzahl der entdeckten Phishing-Webseiten weltweit 2023 | Statista — de.statista.com*, https://de.statista.com/statistik/daten/studie/73876/umfrage/anzahl-der-gemeldeten-phishing-webseiten-weltweit/, [Accessed 07-10-2024].

[3] M. Barbosa, A. Boldyreva, S. Chen und B. Warinschi, „Provable security analysis of FIDO2," in *Advances in Cryptology–CRYPTO 2021: 41st Annual International Cryptology Conference, CRYPTO 2021, Virtual Event, August 16–20, 2021, Proceedings, Part III 41*, Springer, 2021, S. 125–156.

[4] N. Bindel, C. Cremers und M. Zhao, „FIDO2, CTAP 2.1, and WebAuthn 2: Provable Security and Post-Quantum Instantiation," in *2023 IEEE Symposium on Security and Privacy (SP)*, 2023, S. 1471–1490. doi: 10.1109/SP46215.2023.10179454.

[5] D. Boneh, „The Decision Diffie-Hellman problem," in *Algorithmic Number Theory*, J. P. Buhler, Hrsg., Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, S. 48–63, isbn: 978-3-540-69113-6.

[6] S. O. Bradner, *Key words for use in RFCs to Indicate Requirement Levels*, RFC 2119, März 1997. doi: 10.17487/RFC2119.

[7] *Building the JDK — openjdk.org*, https://openjdk.org/groups/build/doc/building.html, [Accessed 06-10-2024].

[8] *CBOR 2014; Concise Binary Object Representation | Overview — cbor.io*, https://cbor.io/, [Accessed 07-10-2024].

[9] W. W. W. Consortium, *Web Authentication: An API for accessing Public Key Credentials - Level 2 — w3.org*, https://www.w3.org/TR/webauthn-2/, W3C, 2021.

[10] *Creating a KeyStore in JKS Format (Configuring Java CAPS for SSL Support) — docs.oracle.com*, https://docs.oracle.com/cd/E19509-01/820-3503/ggfen/index.html, [Accessed 06-10-2024].

[11] C. Culnane, C. J. P. Newton und H. Treharne, „Technical Report on a Virtual CTAP2 WebAuthn Authenticator," *CoRR*, Jg. abs/2108.04131, 2021. arXiv: 2108.04131.

[12] A. Czeskis und C. Brand, „FIDO 2.0: Client To Authenticator Protocol," 2013.

[13] *Der YubiKey — yubico.com*, https://www.yubico.com/der-yubikey/?lang=de, [Accessed 07-10-2024].

[14] H. Feng, J. Guan, H. Li, X. Pan und Z. Zhao, „FIDO Gets Verified: A Formal Analysis of the Universal Authentication Framework Protocol,“ *IEEE Transactions on Dependable and Secure Computing*, Jg. 20, Nr. 5, S. 4291–4310, 2023. doi: 10.1109/TDSC.2022.3217259.

[15] *FIDO Alliance — fidoalliance.org*, https://fidoalliance.org/, [Accessed 07-10-2024].

[16] *FIDO UAF Architectural Overview — fidoalliance.org*, https://fidoalliance.org/specs/fido-uaf-v1.1-id-20170202/fido-uaf-overview-v1.1-id-20170202.html, [Accessed 07-10-2024].

[17] N. Frymann, D. Gardham, F. Kiefer, E. Lundberg, M. Manulis u. a., „Asynchronous Remote Key Generation: An Analysis of Yubico's Proposal for W3C WebAuthn,“ in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, Ser. CCS '20, Virtual Event, USA: Association for Computing Machinery, 2020, S. 939–954, isbn: 9781450370899. doi: 10.1145/3372297.3417292.

[18] S. Ghorbani Lyastani, M. Schilling, M. Neumayr, M. Backes und S. Bugiel, „Is FIDO2 the Kingslayer of User Authentication? A Comparative Usability Study of FIDO2 Passwordless Authentication,“ in *2020 IEEE Symposium on Security and Privacy (SP)*, 2020, S. 268–285. doi: 10.1109/SP40000.2020.00047.

[19] *GitHub - freundma/GNUTLSwithFIDO2Extension: This repository implements an extension of GnuTLS 3.6.15 with FIDO2. The code was developed in the context of a bachelor thesis. — github.com*, https://github.com/freundma/GNUTLSwithFIDO2Extension, [Accessed 07-10-2024].

[20] *GitHub - martinpaljak/FIDO2: FIDO2 toolbox in Java and X-FIDO JavaCard applet — github.com*, https://github.com/martinpaljak/FIDO2, [Accessed 06-10-2024].

[21] *GitHub - openjdk/jdk21: https://openjdk.org/projects/jdk/21 released 2023-09-19 — github.com*, https://github.com/openjdk/jdk21, [Accessed 07-10-2024].

[22] *GitHub - schmori/fido2-tls-extension-client-server — github.com*, https://github.com/schmori/fido2-tls-extension-client-server, [Last commit: 50676e25cf5acdff5786445fa756d5f39dd7fad5].

[23] *GitHub - schmori/jdk21: FIDO2-TLS-1.3-Extension in https://openjdk.org/projects/jdk/21 — github.com*, https://github.com/schmori/jdk21, [Last commit: 68d410456b5b0dcc2d76583f7d596e036bd812b6].

[24] *GitHub - tom95br/tlslite-ng: Git repository for the proof of concept implementation of the FIDO2 extension proposed in the bachelor's thesis "FIDO2 as a TLS 1.3 extension". — github.com*, https://github.com/tom95br/tlslite-ng, [Accessed 07-10-2024].

[25] *GitHub - Yubico/libfido2: Provides library functionality for FIDO2, including communication with a device over USB or NFC. — github.com*, https://github.com/Yubico/libfido2, [Accessed 06-10-2024].

[26] *GitHub: Let's build from here — github.com*, https://github.com/, [Accessed 07-10-2024].

[27] *Gut 5 % der Bevölkerung im Alter von 16 bis 74 Jahren in Deutschland sind offline — destatis.de*, https://www.destatis.de/DE/Presse/Pressemitteilungen/Zahl-der-Woche/2024/PD24_15_p002.html, [Accessed 07-10-2024].

[28] L.-S. Huang, S. Adhikarla, D. Boneh und C. Jackson, „An Experimental Study of TLS Forward Secrecy Deployments," *IEEE Internet Computing*, Jg. 18, Nr. 6, S. 43–51, 2014. doi: 10.1109/MIC.2014.86.

[29] P. Jansen, *TIOBE Index - TIOBE — tiobe.com*, https://www.tiobe.com/tiobe-index/, 2001.

[30] *Java HTTPS Client Certificate Authentication | Baeldung — baeldung.com*, https://www.baeldung.com/java-https-client-certificate-authentication, [Accessed 06-10-2024].

[31] *java-webauthn-server — developers.yubico.com*, https://developers.yubico.com/java-webauthn-server/.

[32] D. Knop, *Yubikey: Cloning-Angriff über Seitenkanal — heise.de*, https://www.heise.de/news/Yubikey-Cloning-Angriff-Offenbar-moeglich-aber-nicht-trivial-9856972.html, [Accessed 07-10-2024].

[33] P. Kushwaha, H. Sonkar, F. Altaf und S. Maity, „A Brief Survey of Challenge–Response Authentication Mechanisms," in *ICT Analysis and Applications*, S. Fong, N. Dey und A. Joshi, Hrsg., Singapore: Springer Singapore, 2021, S. 573–581, isbn: 978-981-15-8354-4.

[34] B. Leiba, *Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words*, RFC 8174, Mai 2017. doi: 10.17487/RFC8174.

[35] *Lesson: All About Sockets — docs.oracle.com*, https://docs.oracle.com/javase/tutorial//networking/sockets/, [Accessed 06-10-2024].

[36] I. Loutfi und A. Jøsang, „FIDO Trust Requirements," in *Secure IT Systems*, S. Buchegger und M. Dam, Hrsg., Cham: Springer International Publishing, 2015, S. 139–155, isbn: 978-3-319-26502-5.

[37] D. A. McGrew und J. Viega, „The security and performance of the Galois/Counter Mode (GCM) of operation,“ in *International Conference on Cryptology in India*, Springer, 2004, S. 343–355.

[38] *Mehr als ein Drittel der Weltbevölkerung ist offline — destatis.de*, `https://www.destatis.de/DE/Themen/Laender-Regionen/Internationales/Thema/wissenschaft-technologie-digitales/Internetnutzung.html`, [Accessed 07-10-2024].

[39] E. Rescorla, „Diffie-hellman key agreement method,“ Techn. Ber., 1999.

[40] E. Rescorla, *The Transport Layer Security (TLS) Protocol Version 1.3*, RFC 8446, Aug. 2018. doi: `10.17487/RFC8446`.

[41] E. Rescorla, „The transport layer security (TLS) protocol version 1.3,“ Techn. Ber., 2018.

[42] J. A. Salowey, D. McGrew und A. Choudhury, *AES Galois Counter Mode (GCM) Cipher Suites for TLS*, RFC 5288, Aug. 2008. doi: `10.17487/RFC5288`.

[43] B. Schmidt, S. Meier, C. Cremers und D. Basin, „Automated Analysis of Diffie-Hellman Protocols and Advanced Security Properties,“ in *2012 IEEE 25th Computer Security Foundations Symposium*, 2012, S. 78–94. doi: `10.1109/CSF.2012.25`.

[44] *Sichere Passwörter erstellen — bsi.bund.de*, `https://www.bsi.bund.de/dok/6596574`, [Accessed 07-10-2024].

[45] *SQLite Home Page — sqlite.org*, `https://www.sqlite.org/`, [Accessed 08-10-2024].

[46] R. Venkateswaran, „Virtual private networks,“ *IEEE Potentials*, Jg. 20, Nr. 1, S. 11–15, 2001. doi: `10.1109/45.913204`.

[47] *Web Authentication (WebAuthn) — iana.org*, `https://www.iana.org/assignments/webauthn/webauthn.xhtml`, [Accessed 06-10-2024].

[48] W.-Z. Yeoh, M. Kepkowski, G. Heide, D. Kaafar und L. Hanzlik, *Fast IDentity Online with Anonymous Credentials (FIDO-AC)*, 2023. arXiv: `2305.16758 [cs.CR]`.

A. Registrierungsnachrichten

In diesem Abschnitt werden die Nachrichten zum Registrieren eines FIDO2-fähigen Token vorgestellt. Jede Nachricht wird durch folgende Merkmale beschrieben: Name, Typ, Länge und eine kurze Beschreibung. Der angegebene Name ist identisch zu dem im beigelegten Quellcode befindlichen Variablen. Die Länge von jeder Nachricht ist entweder fest definiert, da der Variable ein statischer Wert zugeordnet wird oder der Rahmen der möglichen Werte begrenzt ist. Falls der Wert einer Variable dynamisch zugewiesen wird, wird die Länge als ”Variabel” beschrieben. Darüber hinaus wird jede Nachricht beschrieben, d. h. was sie tut und in welchem Kontext sie verwendet wird.

A.1. preFIDO

messageType

- Typ: String
- Länge: zwei Byte
- Beschreibung: Gibt an, dass es sich um eine *preFIDO*-Nachricht handelt.

fido

- Typ: String
- Länge: zwei Byte
- Beschreibung: Gibt an, dass es sich um eine Registrierung oder Authentifizierung handelt. Muss auf ”0” gesetzt sein.

A.2. preFIDORequest

messageType

- Typ: String
- Länge: zwei Byte
- Beschreibung: Gibt an, dass es sich um eine *preFIDORequest*-Nachricht handelt.

ephemeralUserID

- Typ: Byte Array
- Länge: Bis zu 256 Byte
- Beschreibung: Wird serverseitig als ein zufällig generiertes Byte-Array an den Client übergeben und dient zur Identifikation des Clients beim darauffolgenden Handshake.

gcmKey

- Typ: Byte Array
- Länge: Bis zu 256 Byte
- Beschreibung: Wird serverseitig als ein zufällig generierter Byte-String an den Client übergeben und soll gegenfalls zum Ver- und Entschlüsseln von sensiblen, nutzerspezifischen Nachrichten dienen.

A.3. preFIDOResponse

messageType

- Typ: String
- Länge: zwei Byte
- Beschreibung: Gibt an, dass es sich um eine *preFIDOResponse*-Nachricht handelt.

username

- Typ: String
- Länge: Variabel
- Beschreibung: Beinhaltet den Nutzernamen des Clients und soll zur Identifikation auf Serverseite dienen.

ticket

- Typ: String

- Länge: Variabel
- Beschreibung: Beinhaltet ein Geheimnis, welches Client und RP kennen. Dieses Ticket wird verwendet, um den Nutzer für eine Registrierung zu autorisieren.

A.4. FIDO

messageType

- Typ: String
- Länge: zwei Byte
- Beschreibung: Gibt an, dass es sich um eine *FIDO*-Nachricht handelt.

ephemeralUserID

- Typ: Byte Array
- Länge: Bis zu 256 Byte
- Beschreibung: ID wurde dem Client bereits im ersten Handshake übertragen und dient zur Autorisierung des Clients gegenüber dem Server.

fido

- Typ: String
- Länge: zwei Byte
- Beschreibung: Gibt an, dass es sich um eine Registrierung oder Authentifizierung handelt. Muss auf ”0” gesetzt sein.

A.5. FIDORequest

messageType

- Typ: String
- Länge: zwei Byte

- Beschreibung: Gibt an, dass es sich um eine *FIDORequest*-Nachricht handelt.

PublicKeyCredentialCreationOptions

- Typ: Serialisiertes JSON-Objekt
- Länge: Variabel
- Beschreibung: Enthält alle Parameter für die Erzeugung einer entsprechenden Antwort auf Clientseite. Diese Nachricht enthält unter anderem die Challenge, welche vom Authentifikator signiert an die RP zurück geschickt werden soll.
  - **rp**
  - Typ: Serialisiertes JSON Objekt
  - Länge: Variabel
  - Beschreibung: Enthält die ID sowie den Namen der RP.
  - **user**
  - Typ: Serialisiertes JSON Objekt
  - Länge: Variabel
  - Beschreibung: Enthält Angaben zum Nutzer. Dazu zählen Nutzername, Nutzer-ID sowie einen Anzeigename.
  - **challenge**
  - Typ: Byte Array
  - Länge: Mindestens 16 Byte
  - Beschreibung: Ein zufällig generiertes Byte Array.
  - **pubKeyCredParams**
  - Typ: Serialisiertes JSON Array
  - Länge: Variabel

– Beschreibung: Sammlung von *PublicKeyCredentialParameters*, das beschreibt, wie das neue *Credential* generiert werden soll.

  * **type**

  * Typ: String

  * Länge: 10 Bytes

  * Beschreibung: Gibt eine Liste an validen *Credential* Typen zurück. Aktuell ist nur ein Typ definiert namens ”public-key”.

  * **alg**

  * Typ: Integer

  * Länge: 1 Byte

  * Beschreibung: Dieser Wert beschreibt, welcher *COSE* Algorithmus für das Erstellen des *Credentials* verwendet werden soll. Aktuell existieren dafür folgenden Algorithmen: *ES256*, *ES384*, *ES512* und *EsDSA*. In dieser Arbeit wird aktuell nur *ES256* unterstützt.

– **timeout** (optional)

– Typ: Long

– Länge: Variabel

– Beschreibung: Angabe in Milisekunden. Gibt dem Client vor, wie lange die RP auf eine Antwort wartet. Dieses Feld kann auch vom Client überschrieben werden.

– **excludeCredentials** (optional)

– Typ: JSON Array

– Länge: Variabel

– Beschreibung: Gibt eine Liste von *PublicKeyCredentialDescriptor* zurück. Diese Liste beinhaltet alle vom Server akzeptierten *Credentials*, sortiert nach Priorität.

  * **type** (optional)

    * Typ: String
    * Länge: 10 Bytes
    * Beschreibung: Gibt eine Liste an validen *Credential* Typen zurück. Aktuell ist nur ein Typ definiert namens ”public-key”.
    * **id** (optional)
    * Typ: Byte Array
    * Länge: Mindestens 16 Bytes
    * Beschreibung: Enthält die Credential-ID des anfragenden Client.
    * **transports** (optional)
    * Typ: Integer
    * Länge: 1 Byte
    * Beschreibung: Enthält einen Wert, welcher auf ein bestimmtes Transportmittel zeigt. Dabei gibt es folgende Möglichkeiten: ”usb”, ”nfc”, ”ble” und ”internal”
  - **authenticatorSelection** (optional)
  - Typ: JSON Objekt
  - Länge: Variabel
  - Beschreibung: Es werden Vorgaben für die Registrierung des Authentifikators definiert.
    * **authenticatorAttachment** (optional)
    * Typ: String
    * Länge: Variabel
    * Beschreibung: Gibt vor, ob der Authentifikator Client-gebunden sein muss oder nicht. Folgende Werte geben dies vor: ”platform” und ”cross-platform”. In dieser Arbeit wird nur ”cross-plattform” unterstützt.
    * **residentKey** (optional)

  * Typ: String
  * Länge: Variabel
  * Beschreibung: Gibt die Präferenz der RP an, ob dieser *resident Keys* oder *non-resident Keys* bevorzugt. Dafür gibt es folgende Werte: ”discouraged”, ”preferred” und ”required”.
  * **requireResidentKey** (optional)
  * Typ: Boolean
  * Länge: 2 Byte
  * Beschreibung: Eine historisch gewachsene Variable, welche nur dann auf ”true” gesetzt wird, wenn *residentKey* auf ”required” gesetzt wird.
  * **userVerification** (optional)
  * Typ: String
  * Länge: Variabel
  * Beschreibung: Gibt an, wie der Nutzer seinen Authentifikator bedienen muss. Folgende Möglichkeiten stehen zur Auswahl: ”required”, ”preferred” (*Default*) und ”discouraged”.
- **attestation** (optional)
- Typ: String
- Länge: Variabel
- Beschreibung: Dieser Wert wird eingesetzt, um die Herkunft eines Authentifikators und der von ihm ausgegebenen Daten zu bestätigen. Folgende Werte können dafür gesetzt werden: ”none” (*Default*), ”indirect”, ”direct” und ”enterprise”. Aktuell unterstützt diese Arbeit ausschließlich ”none”.
- **extensions** (optional)
- Typ: JSON Array
- Länge: Variabel

  - Beschreibung: Angefragte WebAuthn-Erweiterungen, die bei *Internet Assigned Numbers Authority* registriert sein müssen. [47]

A.6. FIDOResponse

Nachrichtentyp

- Typ: String
- Länge: zwei Bytes
- Beschreibung: Gibt an, dass es sich um eine *FIDOResponse*-Nachricht handelt.

PublicKeyCreationResponse

- Typ: Serialisiertes JSON Objekt
- Länge: Variabel
- Beschreibung: Gibt die Antwort des Authentifikators zurück, womit sich der Client beim Server authentifizieren kann.
  - **clientDataJson**
  - Typ: JSON Objekt
  - Länge: Variabel
  - Beschreibung: Enthält die *challenge*, die *origin* vom Server sowie den *type*, wobei der *type* auf ”webauthn.get” gesetzt sein muss.
  - **attestationObject**
  - Typ: JSON Objekt
  - Länge: Variabel
  - Beschreibung: Beinhaltet Authentifikator- als auch *Credential*-Daten.
    - **aaguid**
    - Typ: String
    - Länge: Variabel

* Beschreibung: Enthält die *aaguid* des Authentifikators.
* **length**
* Typ: Integer
* Länge: Variabel
* Beschreibung: Gibt die Länge der *Credential*-ID an.
* **credentialID**
* Typ: String
* Länge: Variabel
* Beschreibung: Eine eindeutige Byte-Sequenz, die die ID des zu registrierenden *Credential* angibt. Wird als *Hex-String* übergeben.
* **publicKey**
* Typ: String
* Länge: Variabel
* Beschreibung: Eine eindeutige Byte-Sequenz, die die ID des zu registrierenden *Credential* angibt. Wird als *Hex-String* übergeben.

B. Authentifizierungsnachrichten

In diesem Abschnitt werden die Nachrichten zum Authentifizieren eines FIDO2-fähigen Token vorgestellt. Jede Nachricht wird durch folgende Merkmale beschrieben: Name, Typ, Länge und eine kurze Beschreibung. Die Länge von jeder Nachricht ist entweder fest definiert, da der Variable ein statischer Wert zugeordnet wird oder der Rahmen der möglichen Werte begrenzt ist. Falls der Wert einer Variable dynamisch zugewiesen wird, wird die Länge als "Variabel" beschrieben. Darüber hinaus wird jede Nachricht beschrieben, d. h. was sie tut und in welchem Kontext sie verwendet wird.

B.1. preFIDO

messageType

- Typ: String
- Länge: zwei Byte
- Beschreibung: Gibt an, dass es sich um eine *preFIDO*-Nachricht handelt.

fido

- Typ: String
- Länge: zwei Byte
- Beschreibung: Gibt an, dass es sich um eine Registrierung oder Authentifizierung handelt. Muss auf "1" gesetzt sein.

B.2. preFIDORequest

messageType

- Typ: String
- Länge: zwei Byte
- Beschreibung: Gibt an, dass es sich um eine *preFIDORequest*-Nachricht handelt.

ephemeralUserID

- Typ: Byte Array
- Länge: Bis zu 256 Byte
- Beschreibung: Wird serverseitig als ein zufällig generiertes Byte-Array an den Client übergeben und dient zur Identifikation des Client beim darauffolgenden Handshake.

gcmKey

- Typ: Byte Array
- Länge: Bis zu 256 Byte
- Beschreibung: Es handelt sch um einen symmetrischen AES-GCM-Schlüssel und wird serverseitig als ein zufällig generierter Byte-String an den Client übergeben und soll gegebenenfalls zum Ver- und Entschlüsseln von sensiblen, nutzerspezifischen Nachrichten dienen.

B.3. preFIDOResponse

Nachrichtentyp

- Typ: String
- Länge: zwei Byte
- Beschreibung: Gibt an, dass es sich um eine *preFIDOResponse*-Nachricht handelt.

username

- Typ: String
- Länge: Bis zu 256 Byte
- Beschreibung: Beinhaltet den Nutzernamen des Clients und soll zur Identifikation auf Serverseite dienen.

B.4. FIDO

messageType

- Typ: String
- Länge: zwei Byte
- Beschreibung: Gibt an, dass es sich um eine *FIDO*-Nachricht handelt.

ephemeralUserID

- Typ: Byte Array
- Länge: Bis zu 256 Byte
- Beschreibung: ID wurde dem Client bereits im ersten Handshake übertragen und dient zur Autorisierung des Clients gegenüber dem Server.

fido

- Typ: String
- Länge: zwei Byte
- Beschreibung: Gibt an, dass es sich um eine Registrierung oder Authentifizierung handelt. Muss auf ”1” gesetzt sein.

B.5. FIDORequest

messageType

- Typ: String
- Länge: zwei Byte
- Beschreibung: Gibt an, dass es sich um eine *FIDORequest*-Nachricht handelt.

PublicKeyCredentialRequestOptions

- Typ: Serialisiertes JSON Objekt
- Länge: Variabel

- Beschreibung: Enthält alle Parameter für die Erzeugung einer entsprechenden Antwort auf Clientseite. Diese Nachricht enthält unter anderem die Challenge, welche vom Authentifikator signiert an die RP zurück geschickt werden soll.
  - **challenge**
  - Typ: Byte Array
  - Länge: Mindestens 16 Byte
  - Beschreibung: Ein zufällig generiertes Byte Array.
  - **timeout** (optional)
  - Typ: Long
  - Länge: Variabel
  - Beschreibung: Angabe in Milisekunden. Gibt dem Client vor, wie lange die RP auf eine Antwort wartet. Dieses Feld kann auch vom Client überschrieben werden.
  - **rpId** (optional)
  - Typ: String
  - Länge: Variabel
  - Beschreibung: Gibt die RPID an. Meist ist dieser Wert identisch zum Domänen-Namen des Servers.
  - **allowCredentials** (optional)
  - Typ: JSON Array
  - Länge: Variabel
  - Beschreibung: Gibt eine Liste von *PublicKeyCredentialDescriptor* zurück. Diese Liste beinhaltet alle von der RP akzeptierten *Credentials*, sortiert nach Priorität.
    * **type** (optional)
    * Typ: String

    * Länge: 10 Bytes
    * Beschreibung: Gibt eine Liste an validen *Credential* Typen zurück. Aktuell ist nur ein Typ definiert namens "public-key".
    * **id** (optional)
    * Typ: Byte Array
    * Länge: Mindestens 16 Bytes
    * Beschreibung: Enthält die Credential-ID des anfragenden Client.
    * **transports** (optional)
    * Typ: Integer
    * Länge: 1 Byte
    * Beschreibung: Enthält einen Wert, welcher auf ein bestimmtes Transportmittel zeigt. Dabei gibt es folgende Möglichkeiten: "usb", "nfc", "ble" und "internal"
  - **userVerification** (optional)
  - Typ: String
  - Länge: Variabel
  - Beschreibung: Gibt an, wie der Nutzer seinen Authentifikator bedienen muss. Folgende Möglichkeiten stehen zur Auswahl: "required", "preferred" (*Default*) und "discouraged".
  - **extensions** (optional)
  - Typ: JSON Array
  - Länge: Variabel
  - Beschreibung: Angefragte WebAuthn-Erweiterungen, die bei *Internet Assigned Numbers Authority* registriert sein müssen. [47]

B.6. FIDOResponse

messageType

- Typ: String
- Länge: zwei Bytes
- Beschreibung: Gibt an, dass es sich um eine *FIDOResponse*-Nachricht handelt.

PublicKeyRequestResponse

- Typ: Serialisiertes JSON Objekt
- Länge: Variabel
- Beschreibung: Gibt die Antwort des Authentifikators zurück, womit sich der Client bei der RP authentifizieren kann.
  - **clientDataJson**
  - Typ: JSON Objekt
  - Länge: Variabel
  - Beschreibung: Enthält die *challenge*, die *origin* von der RP sowie den *type*, wobei der *type* auf ”webauthn.get” gesetzt sein muss.
  - **authenticatorData**
  - Typ: JSON Objekt
  - Länge: Variabel
  - Beschreibung: Beinhaltet alle Daten des Authentifikators als Byte Array. Dies schließt folgende Werte mit ein: *rpIdHash*, *flags*, *counter*, *attestedCredentialData* (optional) und *extensions* (optional).
  - **signature**
  - Typ: Byte Array
  - Länge: Variabel

- Beschreibung: Signatur über den *clientDataHash* (Hashwert von *clientDataJson*) und ausgewählten Werten aus **authenticatorData**.
- **userHandle**
- Typ: Byte Array
- Länge: Variabel
- Beschreibung: Byte Array, das das ausgewählte *Credential* des Nutzers eindeutig identifiziert.

C. Bauen einer JDK

Die modifizierte *JDK* ist das Kernstück dieser Arbeit, denn hier wurde die TLS-Erweiterung eingebaut. Es handelt sich um die Version 21. Hierfür wurde eine *Fork* vom dem von *openJDK* veröffentlichten Github-*Repository* erstellt. [21] Die *JSK* muss initial genau einmal gebaut werden, um sie in einer Client-Server-Anwendung nutzen zu können. Um die *JDK* bauen zu können, wir ein Linux-Terminal vorausgesetzt. Es folgt eine Anleitung, die dieses Vorgehen beschreibt. [7]

1. Navigiere in `jdk21`
2. `bash configure --with-boot-jdk=/mnt/c/Programme/Java/jdk-21`
3. `make images`
4. Das fertige *Image* liegt anschließend in `jdk21/build/*/jdk` wobei das verzeichnis "*" individuell auf das ausführende Gerät angepasst ist

Als nächstes muss sichergestellt werden, dass die zuvor gebaute *JDK* an die Programmierumgebung übergeben wird. Diese muss jeweils in der Projekt-Struktur sowie in den *Java Compiler* Einstellungen übergeben werden.

D. Zertifikate

In diesem Abschnitt wird erläutert, wie die Client- und Serverzertifikate erstellt und in die Projekt-Struktur eingebunden werden können. Um die Client-Server-Anwendung ausführen zu können, muss zuvor die *JDK* gebaut werden (siehe Anhang C). Um die Implementation ausführen zu können, wird eine entsprechende Programmierumgebung empfohlen. Der Autor hat sich für `IntelliJ IDEA 2023.2.5 (Ultimate Edition)` entschieden, da diese Edition kostenlos von der Humboldt Universität zu Berlin zur Verfügung gestellt wurde. Allerdings können auch beliebige andere Umgebungen verwendet werden. Es wurde auf einem Windows-System mit Version `Windows 10 Pro v22H2` gearbeitet.

In dieser Arbeit werden selbst-signierte Zertifikate erstellt. Darüber hinaus werden *keystores* und *truststores* für Server und Client verwendet. Im *keystore* wird das jeweils eigene Zertifikat gehalten und im *truststore* werden die Zertifikate gehalten, denen vertraut wird. Dies ist ein gängiger Weg bei Java-Anwendungen. [10] Diese wurden mit dem *Tool* `keytool` generiert. [30] Folgender Aufruf erstellt einen serverseitigen *keystore*:

```
keytool -genkey -alias serverkey -keyalg RSA -keysize 2048 -sigalg
   SHA256withRSA -keystore serverkeystore.p12 -storepass password -
   ext san=ip:127.0.0.1,dns:localhost
```

Als nächstes wird ein selbstsigniertes Server-Zertifikat erstellt und an den *keystore* übergeben:

```
keytool -exportcert -keystore serverkeystore.p12 -alias serverkey -
   storepass password -rfc -file server-certificate.pem
```

Beide Vorgänge werden für den Client wiederholt:

```
keytool -genkey -alias clientkey -keyalg RSA -keysize 2048 -sigalg
   SHA256withRSA -keystore clientkeystore.p12 -storepass password -
   ext san=ip:127.0.0.1,dns:localhost

keytool -exportcert -keystore clientkeystore.p12 -alias clientkey -
   storepass password -rfc -file client-certificate.pem
```

Zuletzt wird das Serverzertifikat an den *truststore* des Clients übergeben und das Clientzertifikat an den *truststore* des Servers.

```
keytool -import -trustcacerts -file server-certificate.pem -keypass
    password -storepass password -keystore clienttruststore.jks

keytool -import -trustcacerts -file client-certificate.pem -keypass
    password -storepass password -keystore servertruststore.jks
```

Zuletzt müssen die erstellten Zertifikate, *keystores* und *truststores* an geeigneten Stellen gespeichert werden. Da es sich um eine *PoC* Implementation handelt, wurden diese im *resources*-Ordner gespeichert. Das Übergeben des client- und serverseitigen *keystores* und *truststores* wird programmatisch gelöst.

E. Ausführung

In diesem Abschnitt werden beispielhaft explizite Werte für den *TLSClient* und den *TLSServer* vorgestellt. Für den *TLSClient* ergeben sich folgende Werte:

```
public final static int TLS_PORT = 4321;
public final static String FIDO = "1";
public final static String HOSTNAME = "localhost";
public final static String PIN = "1234";
public final static int CTAP2_PORT = 5555;
public final static CallbackHandler handler = new CLICallbacks();
public final static String TICKET = "clientticket";
public final static String USERNAME = "melanie";
```

Der *TLSServer* kann folgende Werte setzen:

```
public final static int TLS_PORT = 4321;
public final static String RPID = "fido.extension.com";
public final static String RPNAME = "FIDO2 TLS1.3 Extension";
public final static int FIDO_PORT = 6666;
public final static String TICKET = "clientticket";
public final static String authenticatorAttachment = "cross-
   platform";
public final static String residentKey = "required";
public final static String userVerification = "preferred";
public final static int CredentialAlg = 1;
```

Für die endgültige Ausführung, müssen die Klassen *TLSServer, FIDOServer, CTAP2* und zuletzt *TLSClient* gestartet werden.

Selbständigkeitserklärung

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

Berlin, den 8. Oktober 2024 ....