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

Embedding of U2F into TLS 1.3

Bachelorarbeit

zur Erlangung des akademischen Grades
Bachelor of Science (B. Sc.)

eingereicht von: Marco Reda
geboren am: 18.8.1991
geboren in: Bonn

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

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

Contents

1 Abstract

One limit to the spread of technology making login processes safer, is difficulty and effort to implement measures that are easy to utilise by end users. This document describes how U2F, a protocol for simple and user friendly second factor Authentication, can be embedded into TLS 1.3 to make implementing U2F as easy as patching an installed TLS version.

2 Introduction

We live in a world of ever-growing digital infrastructure. The intertwining of computer technology and our everyday life is reaching new heights regularly. By now most of us even carry a powerful computer around in our pockets every day.
Driven by this rapid development of computer technology other fields have to progress quickly to keep up. Information security is a topic often discussed lately, even having found its way into politics and mass media[Her17][Whe17][Alj17].
Keeping our information private, among other security goals, is becoming increasingly difficult, thus more and more of a challenge[Pag12][Ros16].

Nowadays we have a multitude of mechanisms to protect information from various risks. Typical attacks we need to avoid include data manipulation, breach of privacy or data theft. We can encrypt our data with algorithms capable of ciphering data to a point where any decryption attempts by most attackers are not feasible[DBN+01].
We can digitally sign sent data to give others ways to verify a message we sent did really originate from us and was not changed in the process[MKJR16]. These and many other protective measures give us a huge arsenal of countermeasures to attacks targeting our data.
Despite the broad availability of such security measures we hear of incidents preventable through comprehensive usage of information security methods quite often[Her17][Whe17][Alj17]. In many cases measures were not taken to save time and money, not implemented correctly due to lack of knowledge or simply ignored, because they did not seem important. In short, information security must be simple, efficient and easy to handle in order to increase and encourage broad usage[DBZ03, 281].

A typical example of an effective concept of information security is the TLS Protocol. Most internet users use it without substantial knowledge about modern encryption, often not knowing if and how the information they transmit is protected. In many modern webservices TLS usage is handled by user software automatically without any

need to bother the user at all.

Another concept often found in our daily life is multi-factor Authentication, a way of proving we are who we claim to be by more than one method at once. Financial transactions are often confirmed by giving information only we are supposed to know, such as a password or a PIN number, coupled with possession of an object supposedly unique, such as a mobile phone in online transactions or an debit card in cash dispenser usage.
Many programs with logins to accounts send a verification e-mail to the e-mail address with which the account was registered, once a login from a previously unknown location occurs. This e-mail has to be opened and the contained instructions followed, to successfully login[Ste15].

A standard implementing second-factor Authentication as part of logins is the U2F industry standard. A U2F device is registered on or together with an account, making both possession of said unique device and knowledge of the login credentials necessary to successfully log in[SBTC16]. Usage of the device is kept simple to make U2F easy to use, but there are other factors limiting the daily use of U2F in logins.

As of now any application that uses U2F needs to implement U2F Client functionality directly. Many developers do not see this as important enough to invest time and effort to make their software U2F capable.
A simple way of tackling this limiting factor is taking away the need to deal with U2F in application development by simply linking the application to a present implementation of a protocol already incorporating U2F technology.
By adding U2F functionality to TLS we can provide a socket to all applications already using TLS for encryption that utilise user logins, given the Servers they communicate with are capable of U2F as well. This would increase the number of U2F capable applications without any need for changing any user software except TLS implementations themselves.

3 Contents of this Thesis

This thesis contains information about the protocols mentioned, TLS, in particular version 1.3, and U2F, along with concepts of embedding the latter into the former.
Different approaches will be presented and explored in detail on a conceptual level. Implementation details depending on specific TLS implementations will not be addressed in this work, as that would go beyond the scope of a bachelor thesis.
After an overview of U2F, its components and procedures an explanation of the inner works of TLS in the currently widespread version 1.2 follows. The changes and innovations of version 1.3 are explained and solutions to embed U2F into those TLS versions are proposed and summarised.

4 U2F - Universal Second Factor

4.1 What is U2F?

Services providing account-based logins on the internet usually want to make sure that people trying to log into an account really are who they claim to be, the legitimate user behind the username they use.
This is usually made sure by password-based Authentication. Someone trying to log into a service must present secret knowledge that is supposed to be only known to the legitimate user of that account.

Over the years various standards for passwords have been established in reaction to both advancements in password cracking and increasing computing power[San14].
Most modern Registration processes force a set of guidelines for password choice.
Some typical examples are a minimum length, the need to have at least one character of each of a set of groups and avoidance of commonly used passwords.
Minimum length and necessary character classes, like needing to have at least one upper case letter, one number and one special character, protect passwords against brute force attacks, which is simply trying out combinations until one works.
Avoidance of common passwords protects against so-called dictionary attacks, where the most common passwords are tried out[Guc17].
According to a 2016 study, where over 10 million passwords from data leaks were analysed, it is estimated that more than 50 percent of all internet accounts use no more than 25 common passwords, which means that by simply trying out 25 passwords each we could attack countless password protected accounts successfully[Guc17].

Most people using the internet will have met guidelines like these while using the internet and are quite familiar with a problem that goes along with these protective measures: secure passwords get ever more long and complicated all the while we have to memorise more and more passwords if we want to avoid using the same passwords over and over. This would leave us wide open if someone were to find out a password of ours, this password could simply be used to access other services.

Many attempts have been made to avoid this problem without increasing the risk of using password-protected services[Lac16].
One typical and popular way of solving the aforementioned problem is adding a second factor to the Authentication. This may be presentation of something other than knowledge of a secret, to give proof that you are who you claim to be. Using this method can provide additional security to the Authentication process without changing the password-based first factor Authentication.
An often used method is a confirmation e-mail sent to the e-mail address connected with the account, that has to be read in order to log in, that is sent if something about a login attempt is unusual, e.g. a login attempt from a new location occurs[Ste15].
This would be a two-factor Authentication by two knowledge factors, knowledge of the

login credentials and knowledge of the credentials for the e-mail account.
Another example of two-factor Authentication in our everyday lives is almost every financial transaction beyond hard cash.

The FIDO Alliance has proposed a protocol for easy-to-use second-factor Authentication, based on knowledge and possession[SBTC16].
U2F, which stands for Universal Second Factor, is a protocol that "[...] enables relying parties to offer a strong cryptographic 2nd factor option for end user security."[SBTC16, 1]
The concept is simple, yet effective. In addition to a username and password check you present a unique device, analogous to a physical key, that is necessary for the Authentication. If this 'key' is truly unique and you make sure it does stay in your possession, then only you can authenticate yourself with it.

In the U2F protocol the role of the 'key' is played by a physical device connected to the computer used to log into a service. Once you attempt to log in via name and password, you receive a reply, called a challenge, that has to be answered correctly by the device and sent back to the service[SBTC16, 4][BEL16, 2].
This reply contains information which the device can use to make sure it is really receiving a reply from the service you want to log into. Also proof that said reply has not been changed on the way to the device is given[SBTC16, 6]. You are prompted to activate the device physically, e.g. by pressing a button on or touching the device, after which it answers to the service, if everything is in order. Once the service receives the answer and confirms its correctness, the Authentication ends in success[SBTC16, 3].

Since U2F is a technology aimed at usage with browsers, the Clients use a JavaScript API for communication with the Relying Party, which might be undesirable, as it limits the use of U2F to systems supporting JavaScript[BBL16].

4.2 The Components of U2F

In a U2F registration or Authentication process three instances are involved[BEL16, 1].

- The U2F Device, also called Token

  This is usually a physical device, handling cryptographic operations of the U2F protocol. It is connected to the Client Computer via USB, NFC or other local connection. Direct connection to the internet is not allowed[BEL16, 1].

- The Relying Party

  This is the service against which a user wants to authenticate oneself with a login procedure. This is usually a Web-Server. Said Relying Party is identified by an AppID[BEL16, 1].

- The Client

This is the program on the Client Computer that communicates with the Token, the User and the Relying Party. This is usually a Web-Browser[BEL16, 1].

Before a Token can be used to authenticate the user against a service, it has to be registered. This links a Key Handle to an AppID and a key pair. Hereby the key pair is used for cryptographic operations and the Key Handle tells both the Token and the Relying Party which key pair is used. This does not necessarily identify a user[BEL16, 5].
The key pairs consist of a private and a public key. The private key itself or the necessary information to derive it from the Key Handle is stored on the Token and must not leave it[SBTC16, 7]. The public key is sent to the Relying Party, identified by the AppID[SBTC16, 5]. The identity of the service, that is the Relying party, is also called Origin. The Registration or enrollment process is explained further in chapter 4.4.

After this is done the Token can be used, usually but not necessarily together with a username and password, to authenticate a user against a Relying Party. Since the U2F second factor Authentication is entirely an addition to username/password Authentication, multiple Tokens can be registered with the same account and the same AppID, analogous to multiple keys fitting the same lock[SBTC16, 10]. This can be convenient, for example when one user is working at changing locations and cannot take his Token with him, or when multiple legitimate users are working with only one account.

U2F uses challenge-response-based Authentication. The Relying Party constructs a challenge, to which the Token must respond in a certain way. The answer is then verified by the Relying Party[BEL16, 7].
To stay with our key analogy we might describe the process as follows.
The Relying Party presents a lock to the Token. The Token opens the lock and presents the opened lock back to the Relying Party, which verifies that it is in fact open. U2F also provides some additional security features, for example the built in Man-in-the-Middle protection, which is described in the FIDO Alliance Specification[SBTC16, 6].

Both Registration and Authentication consist of three phases, listed as Setup, Processing and Verification.
During Setup the Client connects to the Relying Party, identifies itself and initiates the Authentication or Registration. It receives a response from the Relying Party.
In the Processing step the Client transmits that information to the Token(s) connected to the machine, if the origin of the response matches the expected origin. A user response is prompted, after which the activated Token performs cryptographic operations and transmits the result to the Client.
In the following Verification step said response is transmitted back from the Client to the Relying Party, where it is verified.
If everything is in order then the exchange succeeds, in the case of an Authentication the Client has successfully authenticated itself against the Relying Party, in case of a Registration all information necessary for allowing future Authentications has been

exchanged and the Token is registered[BEL16, 2].

4.3 Communication Client - Token

As the communication between the Client and the Token(s) connected to the machine can be used in our case without any change, I will not go into detail as to how these instances communicate. This is sufficiently documented by the FIDO Alliance already and not of particular interest for this specification. Details to used message formats and their framing can be found in the FIDO U2F specification[BEL16].

4.4 The Registration Process

The Registration of a U2F Token begins with a regular non-U2F Authentication of a Client against the Relying Party[SBTC16, 9]. Once the Client has sent the information used for authentication a Registration request message is sent to the Relying Party. This will contain information to identify the User, such as a User Handle. This can be taken from the regular login credentials. The Relying Party stores this information and sends a U2FChallenge message back to the Client. This message contains a challenge constructed by the Relying Party, an Application ID (AppID) which identifies the Relying Party against the Client and additional information described in chapter 7. If the Client confirms the Challenge Message was sent by the adressed Relying Party by checking the AppID a message constructed from the contained information is sent to the Token which handles cryptographic operations like producing a signature and a certificate. The Token will ask for the User to activate the device manually. If this does not happen in time the Token will send an error message instead. If Activation is successful the result of these cryptographic operations is transmitted back to the Relying Party as a U2FRegisterResponse along with additional information described in chapter 7. This cryptographic information contains data to be saved and stored by the Relying Party, such as the public key generated by the token, the KeyHandle by which this public key is to be indexed and a Response to the Challenge[BEL16].

4.5 The Authentication Process

Authentication follows a similar structure. As with Registration a Request Message with the User Handle is sent after the credential-based login. The Relying Party responds with a Challenge Message same as in Registration, but will transmit all KeyHandles tied to the User Handle. Since a single User can register multiple Tokens with the same Relying Party this is not limited to one KeyHandle. The Client checks the AppID and contacts the Token to generate a Response to the Challenge. From this Response an Authentication Response Message is generated and sent to the Relying Party. The contained Response is checked by looking up the public key tied to the KeyHandle, completing authentication[BEL16, 6].

5 TLS - Transport Layer Security

5.1 What is TLS?

The Transport Layer Security Protocol (TLS), as specified in RFC 5246, provides a means to "establish a secure connection between two parties"[DR08, 6].
With the TLS protocol a secure connection between two partis, A and B, can be established. Any data transmitted over this connection cannot be interpreted by any other party C that intercepts any messages sent between A and B. This works by encrypting the communication between A and B[DR08, 4].
The peers decide how sent data is to be en- and later decrypted, without letting C know how to en- and decrypt said data[DR08, 4]. Once the negotiation, the so called handshake, is done, transmission of encrypted data can start[DR08, 27]. After the data exchange between A and B is finished and all pending data is sent the connection will be closed[DR08, 29]. TLS encryption is a widely used mechanism in modern online communication. For example every time a URL in a browser starts with https instead of http TLS is used for encryption.

5.2 The components of TLS 1.2

TLS 1.2 works by sending messages between both parties, which all have a specific message type.

The TLS Protocol consists of the following components:

- The Record Protocol

  This is the base protocol, the foundation of the other protocols. It handles transformation of application data to a transmittable format, so called records, transmission to the destination and transformation back from the transmittable state after arrival[DR08, 15f.].

- The Handshake Protocol

  This protocol is used to first negotiate common security parameters between both peers, when a TLS connection is about to be formed. These regulate how the Record Protocol transforms the data before sending[DR08, 26f.].

- The Change Cipher Spec Protocol

  During a handshake this protocol is used to change the current ciphering and compression strategies (the current state) to new ones (the pending state)[DR08, 27f.].

- The Alert Protocol

  This protocol specifies a type of transmitted data, a content type called alert type. This is used to signal problems, some of which must cause the connection to terminate immediately[DR08, 28f.].

5.3 The TLS Handshake

Before any application data can be sent from one peer to another both peers must know how to treat the data, e.g. how to encrypt or decrypt it. Similar to other protocols that form a connection, for example TCP, a process called handshake is executed, to form a connection and prepare both peers for upcoming data exchange. Before that handshake the data is neither encrypted nor compressed, as dictated by the initial current state[DR08, 16].
The Handshake Protocol is invoked to send Hello messages and negotiate a so called session between both peers[DR08, 16]. The information shared by this session negotiation consists of the following components:

- session identifier

  An arbitrarily chosen identifier, by which an active or resumable session can be identified[DR08, 26f.].

- is resumable

  An indicator wether or not a session can be resumed later. This will form a new connection using the same security parameters[DR08, 26f.].

- peer certificate

  An X.509v3 certificate to authenticate oneself against the other peer, this field may be empty[DR08, 26f.].

- compression method

  A proposed data compression and decompression method for transformation of application data by the record protocol[DR08, 26f.].

- cipher spec

  A proposed choice of algorithms and attributes for cryptographic transformation of application data by the record protocol[DR08, 26f.].

- master secret

  A shared secret of 48 Byte length, only known to the two peers. This value will be calculated from a negotiated premaster secret and exchanged random values[DR08, 26f.].

The negotiation begins with a ClientHello message from one peer to another, the former will be called Client for this document, the latter Server[DR08, 34].
A ClientHello message contains the following fields:

- client_version

  This field specifies what TLS version the Client wishes to negotiate with the Server[DR08, 39ff.].

- random

  This entry contains a 28 Byte random number and a current timestamp[DR08, 39ff.].

- session_id

  This unencrypted value is an identifier for a previous session that can be used for session resumption[DR08, 39ff.].

- cipher_suites

  Here the Client transmits a list of combinations of cryptographic algorithms the Client wants to use. The server must later pick one entry from this list, to form a cryptographic context[DR08, 39ff.].

- compression_methods

  Similarly a list of compression methods is conveyed[DR08, 39ff.].

- extensions

  This entry is optional. It is used to tell the Server which extensions the Client wishes to add[DR08, 39ff.].

The Server will reply with its own Hello message, the ServerHello message. Its contents are:

- server_version

  The highest TLS version the Server supports that is at the same time not higher than the client_version entry from the ClientHello[DR08, 42f.].

- random

  An own random, independently generated from the ClientHello random[DR08, 42f.].

- session_id

  The identifier for the current session. This is the same as the ClientHello session_id if a session was successfully resumed[DR08, 42f.].

- cipher_suite

  A cipher suite picked from the ClientHello cipher_suites list[DR08, 42f.].

- compression_method

  A compression method picked from the ClientHello compression_methods list[DR08, 42f.].

- extensions

  If there is any need to respond to ClientHello extensions, the ServerHello will respond with own extensions in this field[DR08, 42f.].

After the ServerHello message the Server will present an X.509v3 certificate in its own Certificate message, if the negotiation is not anonymous[DR08, 47]. This may be followed by a Server Key Exchange message, providing additional information for a premaster secret exchange, in case no certificate is sent or if the sent certificate does not provide enough data[DR08, 50ff.]. Usually for the premaster secret exchange RSA encryption or the Diffie-Hellman Algorithm are used[DR08, 50].

The Server may also request a certificate from the Client, followed by signaling the Server sided end of sending hello message associated information, the so called ServerHelloDone message[DR08, 53,55].
Now the Client will, if asked to do so, itself present a certificate and transmit a Client Key Exchange Message, after which a mutual premaster secret is set, from which the master secret is computed[DR08, 57]. Once the master secret is derived, the premaster secret is discarded[DR08, 64].
In order for the Record Protocol to transform and transmit data, it needs an environment to work with, a so called state, consisting of a read and a write state. The state which the Record Protocol uses at the time is called the current state. In order to make any change to this environment a new state, the pending state, must be first initialised, by usage of the negotiated session, and then changed to the new current state using the Change Cipher Spec Protocol[DR08, 16].
Both peers send a ChangeCipherSpec message and signal the Record Protocol to copy the write pending state to the write current state. Upon receiving the message the same is done with the read-state[DR08, 27f.].
The handshake ends after transmission of a verifying Finished message by both peers. This message is the last message sent by a peer in a handshake, it can however still read messages sent by the other side before the handshake ends[DR08, 63f.].

5.4 The TLS Record Protocol

Once the handshake is done, a new current read and write state for the TLS Record Protocol should have been established. The initial current state, which is initialised to the usage of no compression, encryption or message Authentication code (MAC), will have been replaced by a newly negotiated pending state, made current state by usage of the Change Cipher Spec Protocol.

A pending state is generated by setting the following parameters:

- connection end

  This specifies the role of the peer in the connection, possible values are Client and Server[DR08, 16ff.].

- PRF algorithm

  This specifies what pseudorandom function is used for generation of keys from the master secret. PRFs for this purpose are generally constructed in a way that does not allow reconstruction of the master secret from keys[DR08, 16ff.].

- bulk encryption algorithm

  This contains information about what ciphering algorithm is used, if it is a block-, stream- or AEAD ciphering algorithm, along with algorithm specific information, like initialisation vector (IV) length or block size[DR08, 16ff.].

- MAC algorithm

  This specifies what algorithm is used for message Authentication and the length of the algorithm output value[DR08, 16ff.].

- compression algorithm

  This specifies what algorithm is used for data compression along with all information required by this algorithm[DR08, 16ff.].

- master secret

  The master secret negotiated during the handshake. The length is 48 Byte[DR08, 16ff.].

- Client random

  The random value generated by the Client during the handshake. The length is 32 Byte[DR08, 16ff.].

- Server random

  The random value generated by the Server during the handshake. The length is 32 Byte[DR08, 16ff.].

The record protocol derives six values from this state, which are used in encryption and MACing. For both Client and Server each a set of write MAC key, write encryption key and write IV are generated[DR08, 18].
Once this is done the Change Cipher Spec protocol is used to instantiate the pending state, making it the new current state[DR08, 16]. Now procession of Application Data can start.
The connection state holds information over the current compression state, cipher state, MAC key and a sequence number for reading and writing each. The sequence numbers are initialised to zero. After each processed record the sequence number is incremented, up to a maximum of $2^{64} - 1$. Further increment of the sequence number makes a renegotiation necessary[DR08, 18f.].
The record protocol receives application data (appdata) from an application that uses TLS for encrypted communication with another application on another machine. This appdata will never be interpreted by the TLS protocol. Since the application can send

the data in blocks of arbitrary size they need to be split into a more manageable format. This process is called fragmentation and is the first step of preparing the data for transport. The record layer, which received the appdata, fragments it into blocks of no more than 2^{14} Bytes and packs them into TLSPlaintext records by adding information about the fragment length, the used protocol version and the content type of the record (e.g. application data, alert, handshake, change cipher spec). The length may only be zero, if the content type is application data, which is used against traffic analysis attempts[DR08, 19f.].
The TLSPlaintext blocks are now subject to compression, which transforms them into TLSCompressed blocks. This is done by compressing the fragment using the specified compression algorithm, which may be null (identity function, no change to the application data), always provides lossless compression and does not increase the fragment length by more than 1024 Byte. The values for content type and protocol version are kept the same as in the TLSPlaintext block, the length is adjusted accordingly with an upper limit of $2^{14} + 1024$ Byte[DR08, 20f].
Now the TLSCompressed block is encrypted and thereby transformed either into a generic block cipher, generic stream cipher or generic AEAD cipher format.
Stream ciphering adds a MAC, authenticating the content of the TLSCompressed block and sequence number, to the TLSCompressed block then encrypts both into stream TLSCiphertext.
Block ciphering works similarly by adding a MAC, slicing the TLSCompressed into blocks of a given block length and, if necessary, adding a padding to the last block to ensure equal block length over all blocks. These are then encrypted using the block cipher algorithm and are transmitted along with the IV used in the algorithm.
AEAD ciphering works by taking the TLSCompressed block, a nonce and "additional data" which is defined as a concatenation of the sequence number, the content type, the protocol version and the length taken from the TLSCompressed block, which is used in the Authentication check and in ciphering it. It is now combined with the nonce[DR08, 21ff.].

After translation to a ciphered format the result is transmitted to the other peer, then decrypted and authenticated, decompressed, reassembled to application data and given to the receiving application.

5.5 The TLS Alert Protocol

One record type of TLS Records is the Alert Type. Along with the alert name and number code the severity of the alert is transmitted, which can be a warning, a fatal alert or alert 255. In case of a fatal alert, the current session is to be terminated immediately and the session identifier invalidated, preventing resumption of this connection by reuse of the security parameters. This does not terminate other connections using the same security parameters that are already open. Alert records are compressed and encrypted as specified by the connection state[DR08, 28f.].

5.6 End of a TLS connection

Unless a TLS connection gets terminated by a fatal alert it ends after both peers have finished sending. Once a peer, called p1 for now, finishes sending it will send a close notify. After this close notify is received by the other peer, called p2, said p2 will not accept any data received from p1. The connection closure is initiated by that. After p2 sends his own close notify, data can still be sent from the p2 to p1. When both peers have received close notifies the connection is closed[DR08, 29f.].

5.7 Extensions

Even though TLS 1.2 does not make as much use of extensions, as TLS 1.3 does, it is necessary to take a closer look at some of them for this document. Generally speaking an extension is an appendix to a message sent by Client or Server. In order for an extension to be added to a message from the Server the Server must have just received a message from the Client, containing an extension that makes this reply-extension necessary. In TLS 1.2 the extensions are specified as Hello Extensions, meaning they are added to Hello messages[DR08, 44f.].
Every extension consists of a type identifier and an extension data field with a maximum length of $2^{16} - 1$ or 65535 Byte[DR08, 44]. For this specification the following extensions are used, which can be found in RFC 6066.

- ServerNameList

  This extension serves as a way for the Client to address its messages to a number of Servers, that can be identified by Server names. This may become necessary when a network address hosts multiple virtual servers, of which each could be the target recipient. In order to forward the message that carries this extension only to the right Servers, their Server names are listed in this extension.
  If a Server receives a message with this extension and does not find its own server name in the name list it should either continue with the handshake, or abort it. If the Server decides to carry on, this mismatch will become apparent to the Client, that can decide wether to carry on with the handshake, or to abort, as well.
  The extension contains a list of ServerName entries, that hold a name type and the name itself. In contrast to TLS 1.3 it is not explicitly stated if this extension may be added to a ServerHello message as well, so I decided to refrain from that in TLS 1.2[Eas11, 6f.].

- CertificateURL

  During TLS handshakes certificates may get exchanged between the parties. These tend to get large, making them an inefficient construct to send in a TLS handshake. The CertificateURL extension contains information where certificates can be found, in form of a URL. When a client sends this extension appended to a ClientHello message the server can decide to respond with an own CertificateURL

extension, appended to the ServerHello, to indicate that it accepts certificate URLs. This document will make use of this particular trait, repurposing the extension.
This extension contains information about the certificate chain type and a list of URLAndHash entries, each containing the URL itself, a padding and a SHA1 hash[Eas11, 9ff.].

These extensions are natively supported by TLS 1.2, as they are listed in the now obsolete RFC 4366 draft 12, which used to be a work in progress that led to RFC 6066[DR08, 44]. All the extensions listed here originating from RFC 6066 are contained in the 4366 draft 12 as well[Eas10, 2].

6 Changes in TLS 1.3

This document uses the IEFT tls13 Internet Draft 19 as basis for description. TLS 1.3 is still in development, but any changes to it are expected to be minor.
Among several changes from TLS 1.2, including the cease of support for some old and outdated mechanisms, such as DSA encryption, the most important change for this document is the rework of the handshake and its protocols[Res17, 6ff.].
Where TLS 1.2 took two round trips to shake hands, meaning a handshake took messaging from the Client to the Server and back twice in a row, TLS 1.3 is capable of using only one round trip, that is only one message cycle from Client to Server and back, to perform a handshake in most cases[Res17, 21].

Another important change is the introduction of new message types and removal of older ones, also changing which extensions may be added to which message types[Res17, 27,36f.]. For this document most important are the now removed Key Exchange messages and the newly added EncryptedExtensions message. The conveying of key exchange information has moved to extensions, while EncryptedExtensions was introduced as a message type for sharing information that is not required for establishing cryptographic parameters[Res17, 44f.,16].

6.1 Handshake in TLS 1.3

Probably the most important change from 1.2 to 1.3 is the rework that key sharing has undergone. In TLS 1.3 sharing of key information has been moved to an extension of the Client- and ServerHello messages.
In any first contact the Client starts with a ClientHello message[Res17, 29]. In TLS 1.3 the ClientHello contains the following fields:

- legacy_version

  This entry has no use for TLS 1.3 and is kept for compatibility with TLS 1.2. For this reason this value is set to 0x0303, which stands for TLS 1.2. The actual supported versions have been moved to an extension[Res17, 30ff.].

- random

  A 32 Byte random value[Res17, 30ff.].

- legacy_session_id

  TLS 1.3 does not support session resumption anymore. This entry is set to a length of zero[Res17, 30ff.].

- cipher_suites

  This list contains the cipher suites supported by the Client and related data. This usually includes the AEAD encryption algorithm and HKDF hash pairs. The supported groups for the Ephemeral (Elliptic Curve) Diffie-Hellman algorithm ((EC)DHE), which is the TLS 1.3 key sharing algorithm, are shared in an extension, if no pre shared keys are used instead of (EC)DHE[Res17, 30ff.].

- legacy_compression_methods

  Since TLS 1.3 does not use compression methods this field has to be set to a content of one Byte with the value of zero, which means no compression. This field, which has a length of 1 to 255 Byte will be of special interest to us later[Res17, 30ff.].

- extensions

  TLS 1.3 relies heavily on extensions. Any extension appended to a ClientHello message finds room here. The ClientHello has to contain at least either a key_-share extension or a pre_shared_key extension[Res17, 30ff.].

If everything is compliant and all lists offered contain acceptable entries the Server will respond with a ServerHello message. Unknown extensions from the ClientHello will be ignored[Res17, 32].

The ServerHello message contains the following fields:

- version

  This version field corresponds with the current connection and specifies which TLS version is used[Res17, 32f.].

- random

  A 32 Byte random value, independently generated from the ClientHello random[Res17, 32f.].

- cipher_suite

  The cipher suite chosen from the cipher_suites list from the ClientHello message[Res17, 32f.].

- extensions[Res17, 32f.]

Since the Server is only supposed to use extensions necessary for establishing the cryptographic connection and currently only key-share and pre-shared-key are relevant for this, the ServerHello will only contain one or both of these extensions. A part of the random value is replaced by a value used for version downgrade protection, as described in the TLS 1.3 specification[Res17, 32f.].
If the ClientHello contains acceptable parameters, but lacks sufficient information necessary to carry on with the handshake, the Server sends a HelloRetryRequest message: This message is structurally similar to a ServerHello, but without a random field. If reception of this message by the Client does not trigger a ClientHello message with the same cipher suite as the prior sent ClientHello, then instead of a ServerHello an illegal parameter alert must be sent to the Server, aborting the handshake[Res17, 34].

TLS 1.3 no longer supports resuming a session and using it to form a connection any more. Instead a mechanism called pre-shared keys is used for this purpose. After a handshake a shared secret for the next connection can be established. This key is tied to an identity which a Client can transmit to the Server with the pre_shared_key extension to quickly authenticate itself without an elaborate handshake. If such a pre-shared key is accepted by the Server then no full handshake is necessary. The Client can transmit its data without having to wait for the ServerHello message, resulting in a zero round trip connection setup[Res17, 50f.].

6.2 Extensions

In TLS 1.3 Extensions can be added to the ClientHello, ServerHello, Hello-Retry-Request and EncryptedExtensions messages[Res17, 36].
They are generally built in a request-response fashion such that extensions added to the messages originating from the Server, namely ServerHello, Hello-Retry-Request and EncryptedExtensions, are sent in reaction to extensions previously sent in a ClientHello message[Res17, 35].
If a Server needs to request a certificate from the Client it does so in a Certificate-Request message, to which the Client may respond with a Certificate-Message. An extension consists of an indicator of its extension type and an extension data field of at most $2^{16} - 1 = 65535$ Byte length with no positive minimum length[Res17, 35].
We will take a closer look into a few selected extensions from the TLS 1.3 specification draft. A full list of the native extensions to the TLS 1.3 protocol is listed in the TLS 1.3 specification draft[Res17, 36f.].

- Key-Share, appendable to ClientHello, ServerHello and Hello-Retry-Request

  This extension conveys the offered and selected groups and related key-share information for key negotiation between client and Server with (EC)DHE. If sent by the Client a list of groups is offered that the Client supports, from which the Server picks one. The Server responds with its own Key-Share extension to either accept one group from the list (ServerHello) or pick one group and request additional information (HelloRetryRequest). The initial list presented by

the Client may be empty. In this case the Server is prompted to present a list, from which the Client will pick instead. This will expand the handshake by one additional round trip[Res17, 44f.].

- Pre-Shared Key, appendable to ClientHello and ServerHello
- Pre-Shared Key exchange modes, appendable to ClientHello

  These extensions must be used in conjunction. If used, the ClientHello contains both the Pre-Shared Key extension and the Pre-Shared Key exchange modes extension, to which the Server will reply with a Pre-Shared Key extension, if a ServerHello message is sent in return. If a number of Pre-Shared Keys has been established between Client and Server, the Pre-Shared Key extension indicates which of these is used. Each of these Keys is tied to an identity.
  The ClientHello Pre-Shared Key extension contains a list of the identities the Client presents to the Server, the Pre-Shared Key exchange modes extension specifies key exchange modes that are acceptable for exchanging a Pre-Shared Key. This limits the choice of identities in the Pre-Shared Key extension. The Server then picks one from this list that has to meet the requirements of the Pre-Shared Key exchange modes extension. The Pre-Shared Key extension is always the last extension, the order of any other extensions is not specified[Res17, 47ff.].

- Early Data Indication, appendable to ClientHello, Encrypted-Extension and New-Session-Ticket

  One of the new features of TLS 1.3 is the so called 0-RTT resumption. This means that, given a valid Pre-Shared Key has already been shared, a Client can send Application Data along with its ClientHello message. If this is done, the Client sends this ClientHello with both Pre-Shared Key extension and Early Data Indication extension. It is simply an indicator for the Server that the ClientHello already contains Application Data[Res17, 47ff.].

- CertificateAuthorities, appendable to ClientHello and CertificateRequest

  This extension consists of nothing more than a list of names, indicating the certificate authorities supported by the sender. Due to being appendable to the CertificateRequest and the lack of need for client certification in our specification, this extension will be repurposed[Res17, 42].

7 ULS - U2F via TLS

7.1 What do we want?

Our goal is to make a modified version of TLS 1.2/1.3 capable of emulating the capabilities of the U2F protocol all the while staying as conform to the standards as possible. We can logically sort the communications in the U2F protocol into two

categories, one being the communication between Client and Relying Party, the other being the communication between Client and Token. The Token-Client communication can be handled by the libraries already present, which the Client side can simply call from the TLS implementation. This leaves the communication between Client and Relying Party.
We can logically split U2F into two different use cases with one being enrollment, where a U2F Token is registered to a Relying Party, the other being Authentication, where a Token is used to authenticate the Client against a Relying Party. The latter is much more important to cover, since for every registered Token there are many Authentications to be expected. Also the enrollment can simply be done once on a machine capable of U2F without much additional effort. Since the goal is to keep the U2F capabilities detached from application data at least the Authentication and, if possible, the enrollment as well must take place in the interpreted code part of the TLS protocol.
Most services still use a username/password-based Authentication, to which they might add U2F. This information can still be transmitted as application data, which means that the U2F exchange takes place before any username/password checks unrelated to U2F occur. We can also not use any application data in the TLS protocol itself, as this information is not interpreted by the layer in which TLS resides. The message prompting the user to activate the Token should be trivial to implement, but is highly dependent on the specific user interface, therefore it is not covered in this document.

7.2 U2F Authentication summarised

The Client-Relying Party communication during Authentication consists of two round trips. The first one begins with the Client initiating the Authentication process. This message must contain an indicator that the Client wishes to start Authentication and a user ID, for this document called User Handle to identify the user to the Server.
For this User Handle I will use a SHA-256 hash of the username of the underlying login process. Since SHA-256 produces a hash of 256 bit or 32 Byte length this is the length of the hashed username data. If there is no login process tied to the Authentication any other form of username hashed this way will suffice, as long as it is used in consistence with a User Handle given during enrollment.

The Server will respond with all Key Handles tied to the User Handle, its own AppID, a challenge, a version indicator and values for timeoutSeconds and requestID. The format of the challenge seems not to be explicitly defined in the FIDO U2F specifications. As the challenge is represented as a String in the Yubico U2F implementation libraries and will be transmitted to the Token as part of a SHA-256 hashed object I specify the maximum challenge length to be 32 Byte[Yub16].
Likewise the AppID will be given the same length for the same reason. The FIDO U2F specification lists a maximum length of the Key Handle indicated by the Key Handle length Byte, an unsigned int with the values of 0-255 without explicitly stating wether this is bit or Byte[BEL16, 6]. Since the example Key Handle[BEL16, 9] is 64 Byte long

I will assume it is byte and set the maximum Key Handle length to 255 byte[BEL16, 8]. The version indicator will be formatted as in the example from the U2F Specification, which means that 8 Byte should be enough, even for a two-digit version number. As timeoutSeconds and requestID are unsigned long values they take 4 Byte each.

In the final transport step the Client sends the challenge, the signature and counter given by the Token back to the Relying Party, where the signature is verified. The requestID is added here as well. The counter is represented by a 4 Byte big endian representation of the Token counter value in the FIDO specification[BEL16, 6].
The signature is created with Elliptic Curve DSA on P-256 over a String of up to 162 Byte[BEL16, 9]. This signature consists of two 256 bit Integers. These are encoded as ASN.1 Integers, so the addition of a lenght and a type byte are needed. Also ASN.1 calls for an extra bit to indicate wether the number is positive or negative. This sums up to a maximum of 35 Byte per Integer. These need to be capsuled in a structure endcoded with length and type as well, which gives us a total length of 72 byte.
In summary we will have:

- The Authentication request:

  32 Byte User Handle (if using the ULS extension)

  sufficient space for RSA-OAEP encrypted User Handle, estimated under 1kiB (if not using the ULS extension)

  5 Byte for ".u2fa"/".u2fr"

- The Challenge message:

  32 Byte Challenge

  32 Byte AppID

  x*255 Byte KeyHandle

  8 Byte version

  4 Byte requestID

  4 Byte timeoutSeconds

- The Response message:

  32 Byte Challenge

  72 Byte Signature

  4 Byte Counter

  4 Byte requestID

With x being the number of KeyHandles associated with the identified user for this Application.
Should any of these lengths turn out to be insufficient for an implementation they may be extended up to the maximum length they can have while staying short enough to be transported with the means specified in this document.

7.3 U2F Registration summarised

The Registration or Enrollment is similar to the Authentication in regards to the transported information. An enrollment request message is sent by the Client, containing a User Handle for Client identification and an indication it wishes to register a Token.

The challenge message is identical in format to the one in the Authentication process save for the Key Handle, as it is not constructed yet.

The only major difference is the response message. As in the Authentication response a signature, requestID and challenge are transmitted. During enrollment the response also contains the public key obtained from the Token, an attestation x.509 certificate and the User Handle.
The public key length is 65 Byte, the length of the signature is described as "variable length, 71-73 bytes"[BEL16, 5], so I will take it as 73 Byte.
In summary we have:

- The Registration request:

  32 Byte User Handle

  sufficient space for RSA-OAEP encrypted User Handle, estimated under 1kiB (if not using the ULS extension)

  5 Byte for ".u2fa"/".u2fr"

- The Challenge message

  32 Byte Challenge

  32 Byte AppID

  8 Byte version

  4 Byte requestID

  4 Byte timeoutSeconds

- The Response message:

  32 Byte Challenge

  65 Byte Public Key

  255 Byte KeyHandle

  73 Byte Signature

  4 Byte requestID

  A x.509 certificate. As we will transmit this whole by other means the size is not important.

In total 407 Byte plus indicators what type of message is sent are needed.

7.4 Quick vs. Clean, the solutions

Since our ultimate goal is to give U2F capabilities to the TLS 1.3 protocol there is a simple and clean solution to our problem. If we add extensions to the TLS 1.3 protocol that alter the handshake and transport our information in messages specifically designed for U2F over TLS 1.3 we can avoid many problems and pitfalls, that other more unconventional solutions might bring.
Such an extension would have to be approved by the Internet Assigned Numbers Authority IANA, which is a globally operating organisation, meaning a lot of administrative barriers would have to be climbed for the extension to get approved. Therefore I specify both an extension for U2F over TLS and in addition ways to add U2F capabilities to TLS without adding new extensions. This is what some may call a 'dirty hack' and perhaps not good practice, but it might help in spreading the idea behind this thesis.
As we mostly use String containers for most of our data and we usually do not suffer from any lack of space the given size requirements represent the bare minimum used space and more will be used if you transmit the given values as plain strings. This should not become problematic however, als we still stay in acceptable boundaries when transmitting an RSA-OAEP ciphertext as a string.

8 ULS without new extensions

If we do not specify new transmission formats for the U2F data in the form of an extension or message type we have to use existing message types and extensions to transport the data. Therefore size is important for this method, as space might be limited.
Since it is ill advised to change the TLS protocol too much, because maintaining security and compatibility is vital, this specification aims at staying close to the original handshake protocols.

8.1 TLS 1.2

TLS 1.2 uses a 2 round trip handshake by default, which seems convenient, since the U2F protocol exchange also uses 2 round trips both for Authentication and Registration. The TLS 1.2 basic handshake has the following form:

Client		Server
ClientHello	$\longrightarrow$	
		ServerHello
		Certificate*
		ServerKeyExchange*
		CertificateRequest*
	$\longleftarrow$	ServerHelloDone
Certificate*		
ClientKeyExchange		
CertificateVerify*		
[ChangeCipherSpec]		
Finished	$\longrightarrow$	
		[ChangeCipherSpec]
	$\longleftarrow$	Finished
Application Data	$\longleftrightarrow$	Application Data

Figure 1. Message flow for a full handshake
* Indicates optional or situation-dependent messages that are not always sent.[DR08, 36]

To make the ClientHello message capable of conveying a 32 Byte User Handle we have to use an existing extension, since no data field in the ClientHello message we can use is large enough. The ServerNameList extension (see 7.2) is described in RFC6066 as an addition to TLS 1.2. This extension usually indicates to which Server or Servers a Client wants to send, if the target network address hosts mutiple virtual Servers. The extension provides a name field of sufficient size to convey the User Handle. Before sending the User Handle must be encrypted.
Along with the User Handle the Client must tell the Server what kind of U2F action it wishes to initiate. For this purpose the String '.u2fa' for authentication or '.u2fr' for registration is concatenated to the User Handle after encryption. Since the Server-NameList extension is usually used to convey domain names we have to make sure our input is consistent with domain name formats. If we use an encryption that produces domain name compatible ciphertexts this wont be a problem.
All servers implementing ULS must recognise any ServerNames constructed in such manner. As a Server which does not recognize the given ServerName may abort the handshake but is not required to do so this cannot serve as a mechanism to ensure a Server supports ULS. If a Server aborts the handshake after receiving an unrecognised ServerName we can safely say it does not support ULS. If it does not abort we cannot be sure if it supports ULS.
Since the first ClientHello is not encrypted it is advisable to encrypt the User Handle before transmitting it, to hinder identification of a U2F Client by a third party during Registration or Authentication. The User Handle is to be encrypted with RSA-OAEP using the server's public key obtained beforehand by any other means. This will lengthen the User Handle significantly, but since we are using an entire extension for conveying the ciphertext and auth/reg string there should be no space issue.

The ClientHello CipherSuite list must contain at least one algorithm for key exchange that is also acceptable for TLS 1.3, to avoid deprecation. Since we cannot establish a cryptographic context, until the Server has sent a ServerHelloDone message and is thereby not able to send a Challenge to the Client anymore, we complete the Handshake in the usual manner for TLS 1.2. The chosen key exchange algorithm must be acceptable for TLS 1.3 as well for the aforementioned reason. This session will be closed and resumed immediately afterwards, provoking a new, albeit shortened handshake. This new handshake will be encrypted, as the current state of the resumed connection is reused.

The new Client Hello of the rehandshake contains a CertificateURL extension. The data in this extension is insignificant, but makes an answer by the Server that contains this extension as well conform to the TLS 1.2 specification. By default this extension data is empty, but may be filled with any value if there is a valid reason to do so.
The Server replies with a ServerHello message with an own CertificateURL extension. This extension contains a url field with sufficient maximum length of 65535 Byte, which is enough to convey the Challenge Request, unless more than 256 Key Handles are associated with the given User Handle and Application. These values are submitted as a plain text String, each separated by a ”.” character.
The certificate chain type is initialised to individual_certs(0), the fields for padding and hash are filled with zero Bytes.
The Server also asks for a Client Certificate, using the CertificateRequest message you would normally find in a regular handshake. If any kind of certificate from the Client had to be obtained for TLS security reasons that must already have happened in the full handshake of the first connection. This CertificateRequest simply prompts the Client to submit a certificate, which will be the container for the U2F response information.

Before any finished messages are exchanged the Client sends a pseudocertificate to the Server. This is simply a x.509 certificate with custom contained data. In case of Registration an actual x.509 certificate is generated by the Token to be verified by the Server. In this case the values for Challenge, Public Key, Key Handle, signature and requestID are appended to the issuer field, separated by ”.” characters from each other and the original issuer entry.
In case of Authentication an empty certificate is used and the values written into the issuer field in the same manner. During Registration the Server must abort the handshake, if the Attestation Certificate could not be verified. Be sure to remove these values from the transmitted certificate before handling verification. The rest of the re-handshake takes place as described in RFC 5246.

The full U2F over TLS 1.2 without new extensions handshake has the following form:

Client		Server
ClientHello(+ServerNameList)	$\longrightarrow$	
		ServerHello
		Certificate*
		ServerKeyExchange
		CertificateRequest*
	$\longleftarrow$	ServerHelloDone
Certificate*		
ClientKeyExchange		
CertificateVerify*		
[ChangeCipherSpec]		
Finished	$\longrightarrow$	
		[ChangeCipherSpec]
	$\longleftarrow$	Finished

Close connection
Rehandshake:

Client		Server
ClientHello(+certificateURL)		
		ServerHello(+certificateURL)
	$\longleftarrow$	CertificateRequest
Certificate		
Finished	$\longrightarrow$	
		Finished
Application Data	$\longleftrightarrow$	Application Data

8.2 TLS 1.3

In contrast to TLS 1.2's two round trip handshake TLS 1.3 uses only one round trip and does not directly support session resumption any more. The TLS 1.3 basic handshake has the following form:

	Client		Server	
Key |	ClientHello			
Exch |	+ key_ share*			
|	+ psk_ key_ exchange_			
|	modes*			
|	+ pre_ shared_key*	———>		
			ServerHello	| Key
			+ key_share*	| Exch
			+ pre_shared_key*	|
			{Encrypted	| Server
			Extensions}	| Params
			{Certificate	|
			Request*}	|
			{Certificate*}	|
			{CertificateVerify*}	| Auth
			{Finished}	|
		⟵	[Application Data*]	
|	{ Certificate*}			
Auth |	{ CertificateVerify*}			
|	{ Finished}	———>		
	[Application Data]	⟷	[Application Data]	

+ Indicates noteworthy extensions sent in the previously noted message.

* Indicates optional or situation-dependent messages/extensions that are not always sent.

{} Indicates messages protected using keys derived from a [sender]_handshake_traffic_secret.

[] Indicates messages protected using keys derived from traffic_secret_N

Figure 1: Message flow for full TLS Handshake[Res17, 15]

In TLS 1.3 we use the same way of treating and conveying the Request Message as in TLS 1.2.
After the ClientHello has been received by the Server and it has sent a ServerHello the rest of the handshake is covered by TLS encryption, so we do not have to worry about secrecy of the transmitted data any longer. This means there is no need to re-handshake in any way.
The Server must now send a CertificateRequest message with a CertificateAuthorities extension appended to it. Since we already know the Client supports U2F and the server will only send the message in the specified way if it does so too we do not need

actual Client certification during the U2F handshake. This means we can safely repurpose this extension. The data contained in the Challenge Message is simply written in the authorities field in the CertificateAuthorities extension. With the exception of the finished message, that must be delayed until the Client has responded to the CertificateRequest, all other messages are sent as in regular TLS 1.3. The Client must send a Certificate message, constructed in the same way, as in U2F over TLS 1.2 without new extensions. After the certificate has been received and, if needed, verified, the handshake can finish in the usual manner.

The full U2F over TLS 1.3 without new extensions handshake has the following form:

	Client		Server	
Key |	ClientHello			
Exch|	+ key_share*			
|	+ psk_key_			
|	exchange_modes*			
|	+ pre_shared_			
|	key*			
U2F|	+server_name_list	$\longrightarrow$		
Request|				
			ServerHello	| Key
			+ key_share*	| Exch
			+ pre_shared_key*	|
			{EncryptedExtensions}	| Serverparams
			{CertificateRequest}	| U2F
			+ CertificateAuthorities	| chall
			{Certificate*}	
		$\longleftarrow$	{CertificateVerify*}	
U2F |	{Certificate*}			
response |	{CertificateVerify*}			
	{Finished}	$\longrightarrow$		
		$\longleftarrow$	{Finished}	
	[Application Data]	$\longleftrightarrow$	[Application Data]	

9 The TLS 1.3 Extension for ULS

Since an extension has a maximum space of 65535 Byte for data, no problem in containing the data for every U2F message should occur. I define the extension for U2F data analogous to the extension definitions in the IEFT TLS 1.3 specification. Since we have enough space I will provide length fields for every data entry that does not have an explicitly stated length in the U2F specification. Lengths will be given in Byte. The extensions are defined explicitly for TLS 1.3, since TLS 1.2 is widely implemented

and used already and any additions to it might not spread as easily as additions to the relatively new TLS 1.3 protocol.

```
enum {
initiateAuthentication(0),
initiateRegistration(1),
authenticationRequest(2),
authenticationChallenge(3),
authenticationResponse(4),
registrationRequest(5),
registrationChallenge(6),
registrationResponse(7)
} U2FMessageType

struct {
U2FMessageType messageType;

uint userHandle;

uint challengeLength;
String challenge;

uint appIDLength;
String appID;

uint keyHandleLength;
String[] reqisteredKeys;

uint signatureLength;
String signature;

uint publicKeyLength;
String publicKey;

uint counter;

String version;

ulong timeoutSeconds;

ulong requestID;
} U2FExtension
```

This extension is appendable to ClientHello, ServerHello, EncryptedExtensions and CertificateRequest.

If any type of data is not supposed to be transmitted in a message, the length field is set to 0. Which messages are used for transmitting which types of data is covered in the chapter U2F via TLS. The userHandle, counter, timeoutSeconds and requestID values are set to 0, unless they are relevant in the current message. In the initiate messages no data must be transmitted.
The U2F exchange begins with a ClientHello message containing a U2FExtension with an initiate type. The accepting ServerHello message must contain a U2FExtension of the same type and content.
By now cryptographic protection for the connection has been established. As with U2F over TLS without new extensions we could transmit the User Handle in the ClientHello message, but would lose the protection and forward secrecy of the TLS encryption, leaving us vulnerable within the aforementioned limits. The method specified here avoids that problem at the cost of an additional round trip. If that is not desirable the ClientHello message can simply take over the role of the first EncryptedExtension message instead. It is advisable to use encryption over RSA-OAEP if that route is taken.
After this the transmission of actual U2F messages under the protection of TLS encryption can start. For the transmission of U2F messages the EncryptedExtensions and Certificate message types will be used, that convey nothing more than the extensions themselves.
The Client begins with an EncryptedExtensions message with a request type extension, containing the User Handle. The Server responds with an EncryptedExtensions message with a challenge type extension, containing the challenge, the AppID and, in the Authentication case, the registeredKeys. The final message by the Client contains a response type extension.
If it is an Authentication response, it is an EncryptedExtensions message and it contains challenge, signature and counter. If it is a Registration response, the message type is Certificate instead, which is how the x.509 certificate from the Token is transported, and contains challenge, public key, Key Handle and signature in its U2FExtension. The Key Handle is simply put in the first value of the registeredKeys array.
Once the signature has been verified by the Server it may send a finished message to end the handshake, if no further messages need to be sent by the Server for the TLS 1.3 handshake itself.

These message types must be sent in order, any reception of a message containing invalid information, like a public key in an Authentication response or a signature in a challenge message, or message out of order must abort the handshake with a fatal alert. A fitting and ideally unmistakeable alert description from the TLS 1.3 draft must be chosen, I propose unexpected_message(10) for U2F messages out of order and unsupported_extension(110) for invalid combinations of data and extension type. If the server does not support the U2FExtension the Client has to abort the handshake. Should the U2F Registration or Authentication itself fail the handshake must be aborted with a fatal handshake_failure(40) alert.
The full U2F extended TLS 1.3 handshake has the following form:

	Client		Server	
Key |	ClientHello			
Exch |	+ key_share*			
U2F |	+ psk_key_exchange_modes*			
init |	+ U2FExtension(init)			
|	+ pre_shared_key*	⟶		
			ServerHello	| Key
			+ key_share*	| Exch
			+ U2FExtension(init)	| U2Finit
		⟵	+ pre_shared_key*	|
U2F |	{EncryptedExtensions}			
req |	+ U2FExtension(request)	⟶		
			{EncryptedExtensions}	| U2F
			+ U2FExtension(challenge)	| challenge
			{CertificateRequest}	|Serverpar
			{Certificate*}	|
		⟵	{CertificateVerify*}	| Auth
|	{Certificate*}			|
U2F |	+U2FExtension(response)			|
resp |	{EncryptedExtensions*}			|
onse |	+U2FExtension(response)			|
|	{Finished}	⟶		|
		⟵	{Finished}	|
	[Application Data]	⟷	[Application Data]	

10 Summary

Embedding of U2F capabilities, as they are currently specified, into TLS will probably always add round trips to the handshake, be it with a new extension or not. But this is not necessarily a problem, since we usually authenticate ourselves against a service once at login and then use it for a while without any need to periodically reauthenticate. It is not given this will always stay the same, but I assume substantial change to this will not happen before TLS 1.3 becomes outdated and obsolete anyway.
The usual trade-off between speed or practicability and security is hardly relevant in our login use case, so there is probably little to lose and much to gain in terms of additional security.
Of all presented methods for embedding U2F into TLS 1.3 the most promising is the use of a custom-tailored extension, as this may prevent more unexpected problems and odd behaviour of protocols based on the new specification, than repurposing structures meant for different applications.

I do not expect this specification alone to have a large impact on online commu-

nication, but in order to make a difference, first the means must be provided. This specification makes the use of two-factor Authentication easier to use, which, if not a leap, is at least a step in the right direction. As often in internet security we have the technology, it just needs to be used more.

11 Examples

For the following examples I will use the U2F Token data provided in the U2F Raw Message Formats Documentation, namely the attestation certificate, public key, signature and Key Handle[BEL16, 18ff.]. Furthermore there are no other Key Handles stored before each example Registration and Authentication.
The following further example data is used:
User Handle: User1
with SHA256 Hash:

```
27a534a25cf745b6c985eb782079a6fe8641b00003dada14f392a2d01b9c790a
```

Be the RSA-OAEP-encrypted form:

```
120905301215196178389013977060182225512176399857474440928800881904404022837993655100224481032440695516875055228622600676702846382316656432127905709722690341406254568724291003352410343563380958414349679624485469506051717631263272452608549486813623406196017591843076168834145381300008802782252462939288599626951980892228101003549056405774374235591480522735004143079514020904037099513269880168003918480379760525532953647673708343256782040954276563345068352085064253742755093458580125753090179923640904064889967178016285487650181754939723010085103846587356669569615001213864245288286865675705380031563210082370649260937 42
```

Challenge: Challenge
with SHA256 Hash:

```
27cf1792f7bb4da955117bb4a15cb33f4e4705984cacfac9055a2884b061e4e1
```

AppID: example.com
with SHA256 Hash:

```
a379a6f6eeafb9a55e378c118034e2751e682fab9f2d30ab13d2125586ce1947
```

U2F Version 1
No previous requests
1024 seconds timeout

11.1 ULS 1.2 Registration

Client to Server:
ClientHello as specified in RFC 5246 with ServerName Extension. The name field contains the following String:

```
120905301215196178389013977060182225512176399857474440928800881904404022837993655100224481032440695516875055228622600676702846382316656432127905709722690341406254568724291003352410343563380958414349679624485469506051717631263272452608549486813623406196017591843076168834145381300008802782252462939288599626951980892228101003549056405774374235591480522735004143079514020904037099513269880168003918480379760525532953647673708343256782040954276563345068352085064253742755093458580125753090179923640904064889967178016285487650181754939723010085103846587356669569615001213864245288286865675705380031563210082370649260937 42.u2fr
```

Server to Client:
ServerHello, ServerKeyExchange and ServerHelloDone as specified.

Client to Server:
ClientKeyExchange, ChangeCipherSpec and Finished.

Server to Client:
ChangeCipherSpec, Finished, and close_notify(0).

Client to Server:
close_notify(0) and ClientHello with previous session_id and CertificateURL extension.

Server to Client:
ServerHello with CertificateURL extension. The url field contains the following String:

```
27cf1792f7bb4da955117bb4a15cb33f4e4705984cacfac9055a2884b061e4e1.
a379a6f6eeafb9a55e378c118034e2751e682fab9f2d30ab13d2125586ce1947.
0000000000000001.
00000000.
00000400
```

Then a Certificate Request.

Client to Server:
The following Certificate:

```
[
[
Version: V3
```

```
Subject: CN=PilotGnubby-0.4.1-47901280001155957352 Signature
Algorithm: SHA256withECDSA, OID = 1.2.840.10045.4.3.2

Key: EC Public Key
X:
8d617e65c9508e64bcc5673ac82a6799da3c1446682c258c463fffdf58dfd2fa Y:
3e6c378b53d795c4a4dffb4199edd7862f23abaf0203b4b8911ba0569994e101

Validity: [From: Tue Aug 14 11:29:32 PDT 2012, To: Wed Aug 14
11:29:32 PDT 2013]

Issuer: CN=Gnubby Pilot.
27cf1792f7bb4da955117bb4a15cb33f4e4705984cacfac9055a2884b061e4e1.
04b174bc49c7ca254b70d2e5c207cee9cf174820ebd77ea3c65508c26da51b657
c1cc6b952f8621697936482da0a6d3d3826a59095daf6cd7c03e2e60385d2f6d9.
2a552dfdb7477ed65fd84133f86196010b2215b57da75d315b7b9e8fe2e3925a6
019551bab61d16591659cbaf00b4950f7abfe6660e2e006f76868b772d70c25.
304502201471899bcc3987e62e8202c9b39c33c19033f7340352dba80fcab017db9230e
402210082677d673d891933ade6f617e5dbde2e247e70423fd5ad7804a6d3d3961ef871.
00000000

SerialNumber: [ 47901280 00115595 7352] ]

Algorithm: [SHA256withECDSA]
Signature:
0000: 30 44 02 20 60 CD B6 06 1E 9C 22 26 2D 1A AC 1D 0D. `.....&-...
0010: 96 D8 C7 08 29 B2 36 65 31 DD A2 68 83 2C B8 36 ....).6e1..h.,.6
0020: BC D3 0D FA 02 20 63 1B 14 59 F0 9E 63 30 05 57 ..... c..Y..c0.W
0030: 22 C8 D8 9B 7F 48 88 3B 90 89 B8 8D 60 D1 D9 79 "....H.;....`..y
0040: 59 02 B3 04 10 DF Y.....
]
```

Followed by Finished.

Server to Client after verification:
Finished.

11.2 ULS 1.2 Authentication

Client to Server:
ClientHello as specified in RFC 5246 with ServerName Extension. The name field contains the following String:

```
120905301215196178389013977060182225512176399857474440928800881904404022837993655100224481032440695516875055228622600676702846382316656432 12
```

790570972269034140625456872429100335241034356338095841434967962448546950605171763126327245260854948681362340619601759184307616883414538130000880278225246293928859962695198089222810100354905640577437423559148052273500414307951402090403709951326988016800391848037976052553295364767370834325678204095427656334506835208506425374275509345858012575309017992364090406488996717801628548765018175493972301008510384658735666956961500121386424528828686567570538003156321008237064926093742.u2fa

Server to Client:
ServerHello, ServerKeyExchange and ServerHelloDone as specified.

Client to Server:
ClientKeyExchange, ChangeCipherSpec and Finished.

Server to Client:
ChangeCipherSpec, Finished, and close_notify(0).

Client to Server:
close_notify(0) and ClientHello with previous session_id and CertificateURL extension.

Server to Client:
ServerHello with CertificateURL extension. The url field contains the following String:

27cf1792f7bb4da955117bb4a15cb33f4e4705984cacfac9055a2884b061e4e1.
a379a6f6eeafb9a55e378c118034e2751e682fab9f2d30ab13d2125586ce1947.
2a552dfdb7477ed65fd84133f86196010b2215b57da75d315b7b9e8fe2e3925
a6019551bab61d16591659cbaf00b4950f7abfe6660e2e006f76868b772d70c25.
0000000000000001.
00000001.
00000400

Then a Certificate Request.

Client to Server:
The following Pseudocertificate:

```
[
[
Version:
Subject:
Algorithm:

Key:
X:
```

Y:

Validity:

Issuer:27cf1792f7bb4da955117bb4a15cb33f4e4705984cacfac9055a2884b061e4e1.
304502201471899bcc3987e62e8202c9b39c33c19033f7340352dba80fcab017db9230e
402210082677d673d891933ade6f617e5dbde2e247e70423fd5ad7804a6d3d3961ef871.
00000000.
00000000

SerialNumber: [47901280 00115595 7352]]

Algorithm:
Signature:
]

Followed by Finished.

Server to Client after verification:
Finished.

11.3 ULS 1.3 Registration without Extension

Client to Server:
ClientHello with all necessary extensions for negotiating a TLS session and ServerName Extension. The name field contains the following String:

120905301215196178389013977060182225512176399857474440928800881904404022837993655100224481032440695516875055228622600676702846382316656432127905709722690341406254568724291003352410343563380958414349679624485469506051717631263272452608549486813623406196017591843076168834145381300008802782252462939288599626951980892228101003549056405774374235591480522735004143079514020904037099513269880168003918480379760525532953647673708343256782040954276563345068352085064253742755093458580125753090179923640904064889967178016285487650181754939723010085103846587356669569615001213864245288286865675705380031563210082370649260937 42.u2fr

Server to Client:
ServerHello with all necessary extensions and followup messages for negotiating a TLS session and a CertificateRequest with CertificateAuthorities extension. The Authorities field contains the following String:

27cf1792f7bb4da955117bb4a15cb33f4e4705984cacfac9055a2884b061e4e1.
a379a6f6eeafb9a55e378c118034e2751e682fab9f2d30ab13d2125586ce1947.
0000000000000001.

```
00000000.
00000400
```

Client to Server:
The following Certificate:

```
[
[
Version: V3
Subject: CN=PilotGnubby-0.4.1-47901280001155957352 Signature
Algorithm: SHA256withECDSA, OID = 1.2.840.10045.4.3.2

Key: EC Public Key
X:
8d617e65c9508e64bcc5673ac82a6799da3c1446682c258c463fffdf58dfd2fa Y:
3e6c378b53d795c4a4dffb4199edd7862f23abaf0203b4b8911ba0569994e101

Validity: [From: Tue Aug 14 11:29:32 PDT 2012, To: Wed Aug 14
11:29:32 PDT 2013]

Issuer: CN=Gnubby Pilot.
27cf1792f7bb4da955117bb4a15cb33f4e4705984cacfac9055a2884b061e4e1.
04b174bc49c7ca254b70d2e5c207cee9cf174820ebd77ea3c65508c26da51b657
c1cc6b952f8621697936482da0a6d3d3826a59095daf6cd7c03e2e60385d2f6d9.
2a552dfdb7477ed65fd84133f86196010b2215b57da75d315b7b9e8fe2e3925a6
019551bab61d16591659cbaf00b4950f7abfe6660e2e006f76868b772d70c25.
304502201471899bcc3987e62e8202c9b39c33c19033f7340352dba80fcab017db9230e
402210082677d673d891933ade6f617e5dbde2e247e70423fd5ad7804a6d3d3961ef871.
00000000

SerialNumber: [ 47901280 00115595 7352] ]

Algorithm: [SHA256withECDSA]
Signature:
0000: 30 44 02 20 60 CD B6 06 1E 9C 22 26 2D 1A AC 1D 0D. `.....&-...
0010: 96 D8 C7 08 29 B2 36 65 31 DD A2 68 83 2C B8 36 ....).6e1..h.,.6
0020: BC D3 0D FA 02 20 63 1B 14 59 F0 9E 63 30 05 57 ..... c..Y..c0.W
0030: 22 C8 D8 9B 7F 48 88 3B 90 89 B8 8D 60 D1 D9 79 "....H.;....`..y
0040: 59 02 B3 04 10 DF Y.....
]
```

Followed by Finished.

Server to Client after verification:
Finished.

11.4 ULS 1.3 Authentication without Extension

Client to Server:
ClientHello with all necessary extensions for negotiating a TLS session and ServerName Extension. The name field contains the following String:

120905301215196178389013977060182225512176399857474440928800881904404022837993655100224481032440695516875055228622600676702846382316656432127905709722690341406254568724291003352410343563380958414349679624485469506051717631263272452608549486813623406196017591843076168834145381300008802782252462939288599626951980892228101003549056405774374235591480522735004143079514020904037099513269880168003918480379760525532953647673708343256782040954276563345068352085064253742755093458580125753090179923640904064889967178016285487650181754939723010085103846587356669569615001213864245288286865675705380031563210082370649260937 42.u2fa

Server to Client:
ServerHello with all necessary extensions and followup messages for negotiating a TLS session and a CertificateRequest with CertificateAuthorities extension. The Authorities field contains the following String:

27cf1792f7bb4da955117bb4a15cb33f4e4705984cacfac9055a2884b061e4e1.
a379a6f6eeafb9a55e378c118034e2751e682fab9f2d30ab13d2125586ce1947.
2a552dfdb7477ed65fd84133f86196010b2215b57da75d315b7b9e8fe2e3925
a6019551bab61d16591659cbaf00b4950f7abfe6660e2e006f76868b772d70c25.
0000000000000001.
00000001.
00000400

Client to Server:
The following Certificate:

[
[
Version:
Subject:
Algorithm:

Key:
X:
Y:

Validity:

Issuer:27cf1792f7bb4da955117bb4a15cb33f4e4705984cacfac9055a2884b061e4e1.

```
304502201471899bcc3987e62e8202c9b39c33c19033f7340352dba80fcab017db9230e
402210082677d673d891933ade6f617e5dbde2e247e70423fd5ad7804a6d3d3961ef871.
00000000.
00000000

SerialNumber: [ 47901280 00115595 7352] ]

Algorithm:
Signature:
]
```

Followed by Finished.

Server to Client after verification:
Finished.

11.5 ULS Registration

Client to Server:
ClientHello with all necessary extensions for negotiating a TLS session and

```
U2FExtension{
U2FMessageType == 1;
userHandle == 0;
challengeLength == 0;
challenge == null;
appIDLength == 0;
appID == null;
keyHandleLength == 0;
registeredKeys == null;
signatureLength == 0;
signature == null;
publicKeyLength == 0;
publicKey == null;
counter == 0;
version == null;
timeoutSeconds == 0;
requestID == 0;
}
```

Server to Client:
ServerHello with all necessary extensions for negotiating a TLS session and

```
U2FExtension{
```

```
U2FMessageType == 1;
userHandle == 0;
challengeLength == 0;
challenge == null;
appIDLength == 0;
appID == null;
keyHandleLength == 0;
registeredKeys == null;
signatureLength == 0;
signature == null;
publicKeyLength == 0;
publicKey == null;
counter == 0;
version == null;
timeoutSeconds == 0;
requestID == 0;
}
```

Client to Server:
EncryptedExtensions message with

```
U2FExtension{
U2FMessageType == 5;
userHandleLength == 0;
userHandle ==
120905301215196178389013977060182225512176399857474440928800881904404022837993655100224481032440695516875055228622600676702846382316656432127905709722690341406254568724291003352410343563380958414349679624485469506051717631263272452608549486813623406196017591843076168834145381300008802782252462939288599626951980892228101003549056405774374235591480522735004143079514020904037099513269880168003918480379760525532953647673708343256782040954276563345068352085064253742755093458580125753090179923640904064889967178016285487650181754939723010085103846587356669569615001213864245288286865675705380031563210082370649260937423;
challengeLength == 0;
challenge == null;
appIDLength == 0;
appID == null;
keyHandleLength == 0;
registeredKeys == null;
signatureLength == 0;
signature == null;
publicKeyLength == 0;
publicKey == null;
counter == 0;
```

```
version == null;
timeoutSeconds == 0;
requestID == 0;
}
```

Server to Client:
EncryptedExtensions message with CertificateRequest and

```
U2FExtension{
U2FMessageType == 6;
userHandle == 0;
challengeLength == 64;
challenge == "27cf1792f7bb4da955117bb4a15cb33f4e4705984cacfac9055a2884b061e4e1";
appIDLength == 64;
appID == "a379a6f6eeafb9a55e378c118034e2751e682fab9f2d30ab13d2125586ce1947";
keyHandleLength == 0;
registeredKeys == null;
signatureLength == 0;
signature == null;
publicKeyLength == 0;
publicKey == null;
counter == 0;
version == "1";
timeoutSeconds == 1024;
requestID == 0;
}
```

Client to Server:
EncryptedExtensions message with

```
U2FExtension{
U2FMessageType == 7;
userHandle == 0;
challengeLength == 64;
challenge == "27cf1792f7bb4da955117bb4a15cb33f4e4705984cacfac9055a2884b061e4e1";
appIDLength == 0;
appID == null;
keyHandleLength == 128;
registeredKeys == {"2a552dfdb7477ed65fd84133f86196010b2215b57da75d315b7b9e8fe2e39
019551bab61d16591659cbaf00b4950f7abfe6660e2e006f76868b772d70c25"};
signatureLength == 142;
signature == "304502201471899bcc3987e62e8202c9b39c33c19033f7340352dba80fcab017db9
402210082677d673d891933ade6f617e5dbde2e247e70423fd5ad7804a6d3d3961ef871";
publicKeyLength == 128;
publicKey == 04b174bc49c7ca254b70d2e5c207cee9cf174820ebd77ea3c65508c26da51b657
```

```
c1cc6b952f8621697936482da0a6d3d3826a59095daf6cd7c03e2e60385d2f6d9;
counter == 0;
version == null;
timeoutSeconds == 0;
requestID == 0;
}
```

and Certificate

```
[
[
Version: V3
Subject: CN=PilotGnubby-0.4.1-47901280001155957352 Signature
Algorithm: SHA256withECDSA, OID = 1.2.840.10045.4.3.2

Key: EC Public Key
X:
8d617e65c9508e64bcc5673ac82a6799da3c1446682c258c463fffdf58dfd2fa Y:
3e6c378b53d795c4a4dffb4199edd7862f23abaf0203b4b8911ba0569994e101

Validity: [From: Tue Aug 14 11:29:32 PDT 2012, To: Wed Aug 14
11:29:32 PDT 2013]

Issuer: CN=Gnubby Pilot

SerialNumber: [ 47901280 00115595 7352] ]

Algorithm: [SHA256withECDSA]
Signature:
0000: 30 44 02 20 60 CD B6 06 1E 9C 22 26 2D 1A AC 1D 0D. `.....&-...
0010: 96 D8 C7 08 29 B2 36 65 31 DD A2 68 83 2C B8 36 ....).6e1..h.,.6
0020: BC D3 0D FA 02 20 63 1B 14 59 F0 9E 63 30 05 57 ..... c..Y..c0.W
0030: 22 C8 D8 9B 7F 48 88 3B 90 89 B8 8D 60 D1 D9 79 "....H.;....`..y
0040: 59 02 B3 04 10 DF Y.....
]
```

Followed by regular proceeding of the TLS 1.3 handshake

11.6 ULS Authentication

Client to Server:
ClientHello with all necessary extensions for negotiating a TLS session and

```
U2FExtension{
```

```
U2FMessageType == 0;
userHandle == 0;
challengeLength == 0;
challenge == null;
appIDLength == 0;
appID == null;
keyHandleLength == 0;
registeredKeys == null;
signatureLength == 0;
signature == null;
publicKeyLength == 0;
publicKey == null;
counter == 0;
version == "null";
timeoutSeconds == 0;
requestID == 0;
}
```

Server to Client:
ServerHello with all necessary extensions for negotiating a TLS session and

```
U2FExtension{
U2FMessageType == 0;
userHandle == 0;
challengeLength == 0;
challenge == null;
appIDLength == 0;
appID == null;
keyHandleLength == 0;
registeredKeys == null;
signatureLength == 0;
signature == null;
publicKeyLength == 0;
publicKey == null;
counter == 0;
version == "null;
timeoutSeconds == 0;
requestID == 0;
}
```

Client to Server:
EncryptedExtensions message with

```
U2FExtension{
U2FMessageType == 2;
```

```
userHandleLength == 0;
userHandle ==
120905301215196178389013977060182225512176399857474440928800881904404022837993655100224481032440695516875055228622600676702846382316656432127905709722690341406254568724291003352410343563380958414349679624485469506051717631263272452608549486813623406196017591843076168834145381300008802782252462939288599626951980892228101003549056405774374235591480522735004143079514020904037099513269880168003918480379760525532953647673708343256782040954276563345068352085064253742755093458580125753090179923640904064889967178016285487650181754939723010085103846587356669569615001213864245288286865675705380031563210082370649260937421;
challengeLength == 0;
challenge == null;
appIDLength == 0;
appID == null;
keyHandleLength == 0;
registeredKeys == null;
signatureLength == 0;
signature == null;
publicKeyLength == 0;
publicKey == null;
counter == 0;
version == null;
timeoutSeconds == 0;
requestID == 0;
}
```

Server to Client:
EncryptedExtensions message with

```
U2FExtension{
U2FMessageType == 3;
userHandle == 0;
challengeLength == 64;
challenge == "27cf1792f7bb4da955117bb4a15cb33f4e4705984cacfac9055a2884b061e4e1";
appIDLength == 64;
appID == "a379a6f6eeafb9a55e378c118034e2751e682fab9f2d30ab13d2125586ce1947";
keyHandleLength == 128;
registeredKeys == {"2a552dfdb7477ed65fd84133f86196010b2215b57da75d315b7b9e8fe2e3925a6
019551bab61d16591659cbaf00b4950f7abfe6660e2e006f76868b772d70c25"};
signatureLength == 0;
signature == null;
publicKeyLength == 0;
publicKey == null;
counter == 1;
```

```
version == "1";
timeoutSeconds == 400;
requestID == 1;
}
```

Client to Server:
EncryptedExtensions message with

```
U2FExtension{
U2FMessageType == 4;
userHandle == 0;
challengeLength == 64;
challenge == "27cf1792f7bb4da955117bb4a15cb33f4e4705984cacfac9055a2884b061e4e1";
appIDLength == 0;
appID == null;
keyHandleLength == 0;
registeredKeys == ;
signatureLength == 142;
signature == "304502201471899bcc3987e62e8202c9b39c33c19033f7340352dba80fcab017db9
402210082677d673d891933ade6f617e5dbde2e247e70423fd5ad7804a6d3d3961ef871";
publicKeyLength == 0;
publicKey == ;
counter == 1;
version == null;
timeoutSeconds == 0;
requestID == 1;
}
```

References

[Alj17] Global hacking attack infects 57,000 computers. Website, May 2017. Online at `http://www.aljazeera.com/news/2017/05/global-hack-attack-infects-57000-computers-170513005030798.html`; accessed on July 18th 2017.

[BBL16] Dirk Balfanz, Arnar Birgisson, and Juan Lang. *FIDO U2F JavaScript API - FIDO Alliance Implementation Draft 15.* FIDOAlliance, September 2016.

[BEL16] Dirk Balfanz, Jakob Ehrensvard, and Juan Lang. *FIDO U2F Raw Message Formats - FIDO Alliance Implementation Draft 15.* FIDOAlliance, September 2016.

[DBN+01] Morris J. Dworkin, Elaine B. Barker, James R. Nechvatal, James Foti, Lawrence E. Bassham, E. Roback, and James F. Dray Jr. *Advanced Encryption Standard (AES).* NIST, November 2001.

[DBZ03] Robert H. Deng, Feng Bao, and Jianying Zhou. *Information and Communications Security: 4th International Conference, ICICS 2002, Singapore, December 9-12, 2002, Proceedings*. Springer, 2003.

[DR08] T. Dierks and E. Rescorla. *RFC 5246 - The Transport Layer Security (TLS) Protocol Version 1.2*. Internet Engineering Task Force, IEFT, Network Working Group, August 2008.

[Eas10] Donald Eastlake 3rd. *RFC 4366 - Transport Layer Security (TLS) Extensions: Extension Definitions Draft 12*. Internet Engineering Task Force, IEFT, TLS Working Group, September 2010.

[Eas11] D. Eastlake 3rd. *RFC 6066 - Transport Layer Security (TLS) Extensions: Extension Definitions*. Internet Engineering Task Force, IEFT, Network Working Group, January 2011.

[Guc17] Darren Guccione. What the most common passwords of 2016 list reveals. Website, March 2017. Online at `https://blog.keepersecurity.com/2017/01/13/most-common-passwords-of-2016-research-study/`; accessed on July 17th 2017.

[Her17] Alex Hern. Cyberattack on uk political party 'only a matter of time'. Website, May 2017. Online at `https://www.theguardian.com/technology/2017/may/30/hacking-uk-political-party-matter-time-us-expert-phishing/`; accessed on July 18th 2017.

[Lac16] Javier Lacort. Digital security: 5 alternatives to passwords. Website, March 2016. Online at `https://www.bbvaopenmind.com/en/digital-security-5-alternatives-to-passwords/`; accessed on July 18th 2017.

[MKJR16] Ed. K. Moriarty, B. Kaliski, J. Jonsson, and A. Rusch. *RFC 8017 - PKCS #1: RSA Cryptography Specifications Version 2.2*, November 2016.

[Pag12] Pierluigi Paganini. Malware, a cyber threat increasingly difficult to contain. Website, August 2012. Online at `http://securityaffairs.co/wordpress/8202/malware/malware-a-cyber-threat-increasingly-difficult-to-contain.html`; accessed on July 18th 2017.

[Res17] E. Rescorla. *The Transport Layer Security (TLS) Protocol Version 1.3 - draft-ietf-tls-tls13-19*. Internet Engineering Task Force, IEFT, Network Working Group, draft 19 edition, March 2017.

[Ros16] Ben Rossi. Five years in information security what has changed? Website, May 2016. Online at `http://www.information-age.com/`

five-years-information-security-what-has-changed-123461477/; accessed on July 18th 2017.

[San14] Password protection policy. PDF, June 2014. Online at https://www.sans.org/security-resources/policies/general/pdf/password-protection-policy; accessed on July 18th 2017.

[SBTC16] Sampath Srinivas, Dirk Balfanz, Eric Tiffany, and Alexei Czeskis. *Universal 2nd Factor (U2F) Overview - FIDO Alliance Implementation Draft 15*. FIDOAlliance, September 2016.

[Ste15] Steam guard. Website, 2015. Online at https://support.steampowered.com/kb_article.php?ref=4020-ALZM-5519; accessed on July 18th 2017.

[Whe17] Aaron Wherry. Threat of foreign election hacking examined by federal security agent. Website, May 2017. Online at http://www.cbc.ca/news/politics/cse-hacking-election-1.4112445; accessed on July 18th 2017.

[Yub16] Client data (u2f core 0.16.0 api). PDF, 2016. Online at https://developers.yubico.com/java-u2flib-server/JavaDoc/; accessed on July 18th 2017.

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 July 29, 2018 .....