On Making U2F Protocol Leakage-Resilient via Re-keying

Donghoon Chang[1], Sweta Mishra[1], Somitra Kumar Sanadhya[2], Ajit Pratap Singh[1]

[1] IIIT Delhi, India. {donghoon,swetam,ajit1433}@iiitd.ac.in
[2] IIT Ropar, India. somitra@iitrpr.ac.in

Abstract. The Universal 2nd Factor (U2F) protocol is an open authentication standard to strengthen the two-factor authentication process. It augments the existing password based infrastructure by using a specialized USB, termed as *the U2F authenticator*, as the 2nd factor. The U2F authenticator is assigned two fixed keys at the time of manufacture, namely the device secret key and the attestation private key. These secret keys are later used by the U2F authenticator during the *Registration phase* to encrypt and digitally sign data that will help in proper validation of the user and the web server. However, the use of fixed keys for the above processing leaks information through side channel about both the secrets. In this work we show why the U2F protocol is not secure against side channel attacks (SCA). We then present a countermeasure for the SCA based on re-keying technique to prevent the repeated use of the device secret key for encryption and signing. We also recommend a modification in the existing U2F protocol to minimise the effect of signing with the fixed attestation private key. Incorporating our proposed countermeasure and recommended modification, we then present a new variant of the U2F protocol that has improved security guarantees. We also briefly explain how the side channel attacks on the U2F protocol and the corresponding proposed countermeasures are similarly applicable to Universal Authentication Framework (UAF) protocol.
Keywords: Password, Authentication, U2F, UAF, Side-channel attack, Re-keying.

1 Introduction

Password based authentication is the most widely accepted and cost effective authentication technique. To ensure the confidentiality of a password, it is usually stored in a one-way transformed form by applying a technique known as 'Password Hashing'. Generally, the user selected passwords are easy to predict [1] making it easy for an attacker to create a dictionary of the most commonly used passwords and apply the 'Dictionary attack' [2] to retrieve any password from its hash value. To prevent this offline dictionary attack, 'Password Hashing Competition' (PHC) [3] encouraged a resource consuming design for password hashing algorithms. The idea is to slow down the process of parallel computations by making computations resource intensive. To further enhance the security of passwords, use of cryptographic module for password hashing is suggested in [4]. However, these approaches are not sufficient to prevent other online attacks, such as, phishing, Man-In-The-Middle (MITM), etc. Therefore augmenting passwords by a second factor is slowly becoming an industry trend. Some of the common second factor solutions such as Time-based One-Time Password (TOTP) [5] and Short Message System (SMS) [6] are easy targets for phishing or MITM attacks which in turn affect user privacy severely and may cause major losses [7]. Frequent cases of phishing and monetary theft due to security flaws in password based solution have been reported. The U2F protocol, proposed by Fast IDentity Online (FIDO) alliance in 2014, has been introduced as a strong augmentation mechanism that can overcome all the known attacks currently faced in practice. Another protocol proposed by the FIDO alliance is called Universal Authentication Framework (UAF). The UAF protocol is an authentication protocol which supports biometric authentication to provide a unified and extensible authentication mechanism that supplants passwords [8]. As UAF supports biometric authentication, a different category of authentication in contrast to the U2F which supports passwords, in this work we focus only on the analysis of the U2F protocol.

The U2F solution is based on public-key cryptography [3]. Considering the claim of U2F developers that U2F is a significant contribution towards strengthening simple password-based authentication, a thorough third party analysis of U2F solution is needed.

[3] A public-key cryptosystem uses two mathematically related, but not identical, keys - a public key which is known to all and a private key which is known only to the owner. It accomplishes two functions: encryption and digital signature. For encryption a sender encrypts with the public key of the receiver so that only the receiver can decrypt with its own private key. For digital signature the sender signs a known message with its private key so that the receiver can verify with the public key of the sender. For more details one can refer to [9]

Motivation: At the time of manufacturing phase, the U2F authenticator is assigned a unique device secret key and an attestation private key. Both these keys are fixed throughout the life-time of the token. The device secret key is used to compute a value called *Keyhandle*. The attestation private key is used to perform a signature. Both these operations are performed during the *Registration Phase* of the U2F protocol. Multiple execution of the *Registration Phase* involving computations with these fixed keys leaks information through *side channel* about both the keys.

The side channel attacks (SCA), proposed by Kocher [10, 11], exploit weaknesses in the physical implementation of cryptosystem to recover the secret key information. These attacks treat ciphers as grey box and capitalize on the side channel leaks such as timing information [10], power consumption [11], electromagnetic leaks [12] etc. to correlate them with the internal states of the processing device which are dependent on the secret key. Several flavors of side channel attacks have been proposed depending upon the type of leakage investigated. Among them, power attacks are the most popular and extensively studied class of attacks. Power attacks analyze the power consumption of a cryptographic operation to recover the secret information. The side channel attacks are the most powerful and easy to implement attacks against cryptographic implementations [10,11].

In general, the U2F authenticator is a specialized USB and all crucial cryptographic operations are performed inside it. Physical access to the USB is not a difficult scenario and hence side channel attacks are easy to mount. Any cryptosystem is an easy target of these kinds of attacks if suitable countermeasures are not adopted. The U2F specifications [13,14] do not consider security against side channel attacks. However, the security of the U2F solution depends on the keys which can easily be disclosed though a side channel attack. Therefore, it is of utmost importance to protect U2F from this category of attacks.

The 'Differential Power Ananlysis (DPA)' attack [11] is a more advanced form of power attack which allows an attacker to compute the intermediate values within cryptographic computations by statistically analyzing data collected from multiple cryptographic operations [10]. When a cryptographic operation is performed over a fixed key, computation with different input-output pairs can easily reveal the key by applying DPA attack. Template attack [15] is another powerful side channel attack which can reveal the secret with very few input-output pairs. It requires significant pre-processing to be implemented but can break implementations secure against DPA attack as well [15]. As template attack is difficult to implement in practice [16], we analyze and attempt to harden the U2F protocol to achieve DPA resistance.

Given the easy applicability of side channel attacks on a device which can be in the hands of an attacker, we consider the leakage of secret keys under DPA. We show that the use of fixed keys in U2F protocol can leak information about both the secrets by applying the DPA attack. We explore the possible cryptographic solutions to prevent repeated use of the fixed keys for U2F protocol. We show that the rekeying technique, which generates different session keys, is a suitable countermeasure to this attack.

"Rekeying" is a technique where different subkeys are derived from a master key to limit the number of operations performed under the same master key [17]. It is considered an efficient approach to prevent leakage of a secret key information by limiting the number of operations performed with each generated keys. In our proposed solution, we utilize rekeying technique to protect the computations that are performed under fixed keys in the U2F protocol.

Our Contributions:

1. We first show why the U2F protocol cannot be secure against side channel attacks. We then propose a countermeasure to fix this issue. Our proposed solution is based on the rekeying technique that derives session keys from the device secret key. Use of different session keys instead of the fixed device secret key helps in keeping the device secret key confidential.
2. The signature function that uses the attestation private key also leaks information that can be captured through side channel analysis. Usually in public key infrastructure (PKI) [9], the verification of digital signature is satisfied by the use of certificates. Intuitively, it may seem that the use of session key will prevent the side channel attack on attestation private key as well. However, it is not feasible to get a fresh certificate from the CA for each private (session) key derived from the initial key. Hence, it is difficult to prevent side channel attacks using public key infrastructure. Therefore, we recommend a modification to the protocol for mitigating this attack.
3. Incorporating our proposed countermeasure and recommended modification, we then present a new variant of the existing U2F protocol that provides strong security guarantees.

4. There is a lack of clear and comprehensive literature that describes the U2F protocol with detailed security analysis. We present a detailed analysis of the U2F protocol including its security analysis. We also try to fill the gap in the description of the protocol by providing clear explanation of all the cryptographic operations.
5. We briefly explain how the side channel attacks on the U2F protocol and the corresponding proposed countermeasures are similarly applicable to Universal Authentication Framework (UAF) protocol as well.

The rest of the paper is organized as follows. In Section 2, we present an overview of the U2F protocol. In Section 3, we detail the existing security analysis on U2F protocol. Possible side channel attack points that can be exploited in U2F protocol are explained in Section 4. Subsequently, the countermeasure on these attacks and the complete description of our modified U2F protocol incorporating the countermeasure are documented in Section 5. This is followed by a detailed explanation of the design rationale behind our proposed modifications in Section 6. In Section 7, we discuss the security evaluation and some limitations of the U2F protocol. The Section 8 explains in brief how the side channel attacks of U2F and the corresponding countermeasures can be applied to UAF protocol as well. Finally, in Sections 9, we conclude our work.

2 Overview of the U2F Protocol

In a password-based authentication system, a server allows n users identified with their usernames such as $u_1, u_2, \ldots, u_n$, and their corresponding passwords $p_1, p_2, \ldots, p_n$. For instance, u_i and p_i denote the username and password of the ith user respectively. The server maintains a database listing pairs of username and corresponding hashed password as follows.

$$(u_i, H(p_i \parallel s_i))$$

where H is the password hashing algorithm imposed by the server and s_i is the server generated salt corresponding to each user u_i which is a fixed length random value. U2F is a second factor authentication protocol that augments this simple password-based authentication system and is explained ahead. The key notations used in this work are listed in Table 1.

The U2F is a protocol proposed by Fast IDentity Online (FIDO) alliance in 2014 as U2F v1.0 [13] and later in 2016 as U2F v1.1 [14]. The protocol description is similar in both the versions with the only differences being in the optimization of their implementation. Following are the listed differences between these two versions.

– The U2F protocol allows multiple tokens to be registered for the same AppId for a specific user. Therefore, for each token, different Keyhandles are generated for a specific AppId. In U2F v1.0, a separate AppId and challenge pairs are sent for every keyHandle, whereas U2F v1.1 suggests an optimization over it. The U2F v1.1 allows a single AppId and challenge pair for multiple keyHandles registered for the AppId.
– The U2F v1.1 JavaScript API specification supersedes JavaScript API of U2F v1.0. The major difference between these two versions is the way requests to be signed are formatted between the relying party and the client.

The U2F protocol allows online services to enhance the security of existing password infrastructure by adding a strong second factor, called 'Universal 2nd Factor' (hence the name U2F), at the time of user login. The augmentation is claimed to be a secure and user friendly solution. To use U2F, the user needs to login with a username and password and then perform a simple button-press on an associated device (which usually happens to be a USB in the case of U2F). A single U2F authenticator can be used across all online services that support the protocol with built-in support in the web browser of the client.

The protocol allows the U2F authenticator to be either a secure hardware or a software. In the following description we use a common term 'U2F token' to express both the hardware and the software. The U2F protocol supports two generic steps for online authentication, namely, 'Registration' and 'Authentication'. Both of these processes involve three parties: U2F token (hardware/software) which is also referred to as the 'FIDO Authenticator', the web browser of the client which is called 'FIDO client' and the web server which is referred to as the 'Relying Party'. The U2F token is a crucial part for the analysis of the protocol. Therefore, we explain the protocol in three steps, appending the process of assigning the secret keys to the U2F token by the manufacturer at the time of manufacturing as the first step. Following is the description of these three steps of the protocol.

Table 1: Notations

Relying Party/Origin	Web server or Server
FIDO Client	Web browser of the client
FIDO Authenticator	U2F token (hardware/software)
AppId	URL of the web server supporting U2F
DS_K	Unique secret key assigned to the U2F token
K_i	i^{th} session key generated from DS_K and the counter i
PK_u	Public key of the AppId corresponding to the username u
SK_u	Private key of the AppId corresponding to the username u
PK_M	Attestation public key of the manufacturer
SK_M	Attestation private key of the manufacturer
PK_T	Public key of the U2F token
SK_T	Private key of the U2F token
$Sign_x(m)$	Signing operation on input message m, using secret key x
ACert	Attestation certificate issued by a trusted CA to the manufacturer
$\|\|$	concatenation operator
$\mathcal{K}(\cdot)$	a randomized asymmetric keypair generation function
s	salt, a fixed length public random value
$f^{(i)}(K_0)$	i times recursive call of function f starting from initial value K_0
$x \xleftarrow{\$} \{0,1\}^n$	randomly generated n-bit value x
$DB \xleftarrow{add} x$	adding the value x to the database DB
R	An r-bit challenge randomly generated by the server
Kh	A key container called Keyhandle generated by the U2F token at the time of 'Registration' which contains AppId and private key SK_u
$H(x)$	Computing hash on the value x where H is any cryptographic hash function
CA	Trusted Certificate Authority under PKI infrastructure
S_db	A database maintained by the server to keep records
U2F_db	A database maintained by the U2F token to keep records

2.1 U2F Manufacturing Phase

This phase is executed at the time of manufacturing of the U2F token as shown in Fig 1. In this phase, the U2F token is provided with a randomly generated unique secret called Device Secret Key DS_K at Step1. At Step2, a public-private key pair (PK_M, SK_M) is provided to the token. The (PK_M, SK_M) keypair is not a one-time generation. Once randomly generated from a key generation function, the manufacturer provides the same (PK_M, SK_M) pair to all the tokens of a specific model manufactured by it. Note that the manufacturer can keep the same key pair for all the tokens manufactured by it (that is, it can treat all the tokens manufactured by it as the same model). At Step3, a certificate Acert issued by a trusted CA which includes PK_M (the public key of the manufacturer) is provided to the token. During the registration phase, the signature with SK_M is verified with PK_M which is extracted from Acert by the server. This verification proves the genuineness of the U2F token. Therefore, both DS_K and (PK_M, SK_M) are fixed for a token while the same (PK_M, SK_M) values are provided to multiple tokens which preserve the anonymity of the token. An integer counter 'ctr' is provided at Step4. This 'ctr' is initialized to value zero and gets incremented after each successful execution of the authentication phase as explained in Section 2.3.

2.2 U2F Registration Phase

The registration procedure is explained in Fig 2. Initially the username u and password p are communicated to the server at Step1. At Step2 the server verifies the values u and $H(p \parallel s)$ which is the hash of the password where s is the salt. On successful verification, the server generates a r bit random challenge R to differentiate each request, else it rejects the registration request. The server sends its identity as AppId and the value R to the browser at Step3. At Step4 the values are forwarded to the U2F token along with the (TLS)

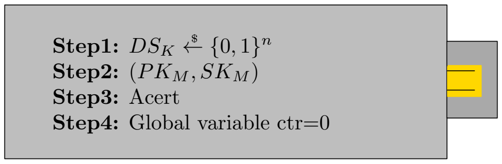

Step1: $DS_K \xleftarrow{\$} \{0,1\}^n$
Step2: (PK_M, SK_M)
Step3: Acert
Step4: Global variable ctr=0

Fig. 1: **U2F Manufacturing Phase.** The key DS_K is a randomly generated n-bit secret. The keypair (PK_M, SK_M) is the asymmetric keypair of the manufacturer shared with the token. Acert is the certificate signed by a trusted CA which includes PK_M. At the time of registration phase, a signature with SK_M is verified with PK_M from Acert by the server to prove the genuineness of the U2F token. The global variable ctr is initialized to zero and incremented after each successful authentication as explained in Section 2.3

'channelId' of the browser for the server. The TLS channelId is a TLS extension, originally proposed in [18] as Origin-Bound Certificate and its refined version is available as an IETF Internet-Draft [19]. Receiving the information, at Step5, the U2F token first generates a website specific public-private key pair represented as (PK_u, SK_u) by applying a random key generation function $\mathcal{K}(\cdot)$. The function $\mathcal{K}(\cdot)$ can be an openSSL key pair generation library which needs as input an elliptic curve such as the NIST standard P-256 curve. We denote a function $f(\cdot)$ which depends on a secret key K as $f_K(\cdot)$. The U2F token computes the function with key DS_K and outputs f_{DS_K} (AppId $\|$ SK_u) which is called the Keyhandle Kh. The above function can be instantiated with any block cipher such as AES in CBC mode or HMAC as shown in Fig. 3 and 4 respectively. The values (PK_u, SK_u) and Kh can optionally be stored in a database denoted as U2F_db inside the U2F token. It then computes a signature $\mathcal{S}$ as Sign_{SK_M}(AppId, R, ChannelId, PK_u, Kh). At Step6, the token sends the values PK_u, Kh and $\mathcal{S}$ along with ACert to the browser. The browser forwards the received values to the server at Step7. After receiving the values, the server first verifies the signature $\mathcal{S}$ with PK_M from the certificate Acert. This verification proves the genuineness of the U2F token. On verification, the server updates its database S_db with the values (PK_u, Kh, Acert) corresponding to the username u and it shows the successful registration of the U2F token else the server rejects the registration request. The U2F protocol allows a user to register multiple tokens with the same account.

2.3 U2F Authentication Phase

The authentication procedure is explained in Fig. 5. Once second factor authentication with U2F is registered, for example, when the values $(u, H(p \mid\mid s), PK_u, Kh$, Acert) are registered with the server for a particular user u, the subsequent login to the website needs to verify the registered values by communicating with the U2F token through browser. The steps are as follows. After receiving username u and password p at Step1, the server checks its database to retrieve the Keyhandle Kh for u at Step2. On successful verification, the server generates a r bit random challenge R and sends it along with Kh and its AppId to the browser at Step3. The browser forwards the received information with ChannelId to the U2F token at Step4. At Step5, the U2F token first verifies the received values by performing inverse computation ${f_{DS_K}}^{-1}(Kh) =$ (AppId$'$ $\mid\mid$ SK'_u). It then compares the received AppId$'$ with the stored AppId. Similar checks can be done if $f_K(\cdot)$ is a keyed hash as shown in Fig. 4. If a successful match happens, the token increments a counter represented as 'ctr'. This 'ctr' can be a global or a local variable. If it is local variable then each AppId gets its own 'ctr' else a single 'ctr' is used across all registered AppId. Throughout the explanation we consider 'ctr' as a global integer variable. The value of the 'ctr' is incremented after each successful authentication by the U2F token. This value 'ctr' is introduced to detect cloning of the U2F token.

A common danger with lightweight user-held devices like tokens is the duplication or the 'cloning' of the device. A cloned U2F device can be created by copying all the information, including the current 'ctr', from the original device. The U2F protocol thwarts the 'cloning attack' as follows. At Step5, the U2F token signs the values received from step4 and 'ctr' with the signing key SK_u, in addition to the generation of the Keyhandle. The signature generated at this step is represented as $\mathcal{S}$. The U2F token sends $\mathcal{S}$ and ctr to the browser at Step6. The browser in turn forwards these values to the server at Step7. Finally the server verifies the received signature $\mathcal{S}$ with the stored key PK_u for the username u from S_db. To detect cloning, the server checks if the received 'ctr' is greater than previously stored 'ctr'. If this is not the case then inconsistency in

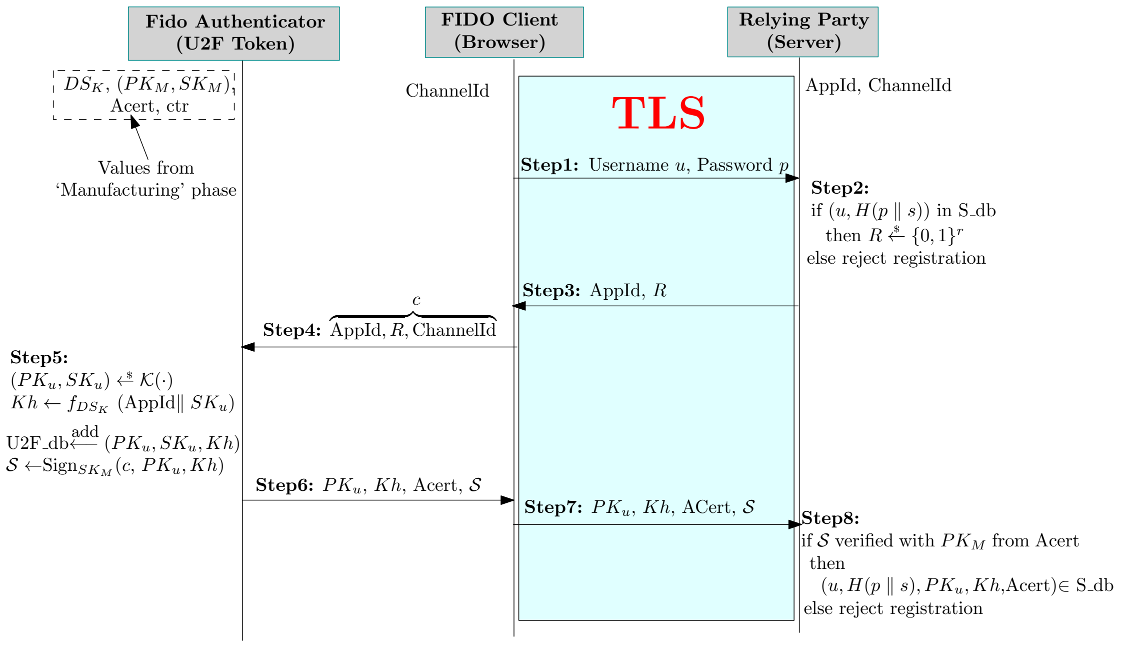

Fido Authenticator (U2F Token)
FIDO Client (Browser)
Relying Party (Server)
DS_K, (PK_M, SK_M), Acert, ctr
Values from 'Manufacturing' phase
ChannelId
TLS
AppId, ChannelId
Step1: Username u, Password p
Step2: if $(u, H(p \parallel s))$ in S_db then $R \xleftarrow{\$} \{0,1\}^r$ else reject registration
Step3: AppId, R
Step4: $\overbrace{\text{AppId}, R, \text{ChannelId}}^{c}$
Step5: $(PK_u, SK_u) \xleftarrow{\$} \mathcal{K}(\cdot)$
$Kh \leftarrow f_{DS_K}$ (AppId$\parallel$ SK_u)
U2F_db$\xleftarrow{\text{add}}$ (PK_u, SK_u, Kh)
$\mathcal{S} \leftarrow \text{Sign}_{SK_M}(c, PK_u, Kh)$
Step6: PK_u, Kh, Acert, $\mathcal{S}$
Step7: PK_u, Kh, ACert, $\mathcal{S}$
Step8: if $\mathcal{S}$ verified with PK_M from Acert then $(u, H(p \parallel s), PK_u, Kh,\text{Acert}) \in$ S_db else reject registration

Fig. 2: **U2F Registration Phase.** The key DS_K, the asymmetric keypair$(PK_M,\ SK_M)$ with attestation certificate Acert which includes PK_M and the counter 'ctr' are from the manufacturing phase. AppId is the URL of the server. ChannelId is the TLS ChannelId. S_db is the server database. R is a r bit random number generated by server. $\mathcal{K}(\cdot)$ is a random key generation function which generates the keypair $(PK_u,\ SK_u)$. $f_{DS_K}(\cdot)$ with key DS_K is the function to generate Keyhandle Kh as shown in Fig. 3 and 4, U2F_db is the database at U2F token.

'ctr' is detected. This could be an attempt to clone a legitimate token and this detection is possible when the legitimate token is successfully authenticated at least once by the server after being cloned. Only on successful verification does the server store the updated 'ctr' value in the database.

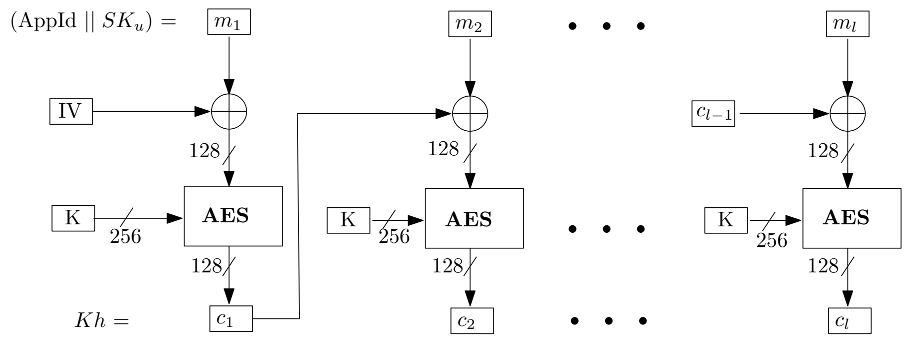
3 Existing Security Analysis of U2F Protocol

This section presents the current security analysis of U2F protocol.

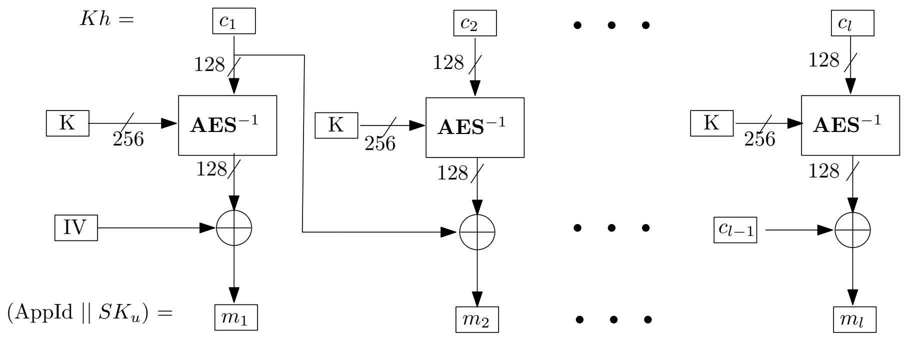
3.1 Denial of Service (DoS) Attack

Denial-of-service attacks are characterized by an explicit attempt by attackers to prevent legitimate users of a service from using that service [20]. We explain some ways in which such an attack can be launched against the U2F protocol.

1. Following the explanation of the protocol in Section 2, a user specific public-key and Keyhandle (PK_u and Kh) are generated by the U2F token at the time of registration. The secret key is kept inside the token and the public key is stored on the server, to identify the Keyhandle later. To bind a Keyhandle with a token, a database of the public-key Pk_u and the Keyhandle Kh is maintained at the server.
   If an attacker can modify the stored public key Pk_u at the server then it will cause a DoS attack for a legitimate user at the time of authentication [21] since access will be denied to the legitimate user after verification in Step8 of Fig. 11. In order to eliminate this attack, secure storage of PK_u is required. Note that use of certificates (as in traditional PKI setting) is not feasible here since the asymmetric keypairs are generated by the token and getting certificate from a trusted CA for each new registration is not practical for the token.

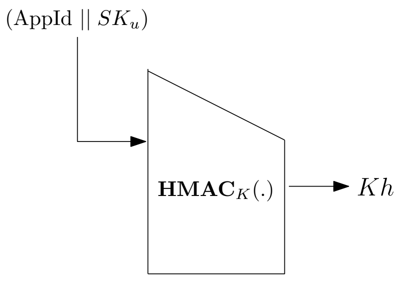


$f_K(\cdot)$ = CBC-AES_K(AppId || SK_u), Cipher-block chaining (CBC) mode encryption using AES with 256-bit key K, input (AppId || SK_u) is padded and split in block-sized l chunks, i.e., (AppId || SK_u) = m_1 || m_2 || … || m_l and corresponding output is $Kh = c_1$ || c_2 || … || c_l, IV is the initialization vector.



$f_K^{-1}(\cdot)$ = CBC-$AES_K{}^{-1}(Kh)$, Cipher-block chaining (CBC) mode decryption using AES^{-1} with 256-bit key K, input is Keyhandle $Kh = c_1$ || c_2 || … || c_l and output is (AppId || SK_u) = m_1 || m_2 || … || m_l

Fig. 3: Instantiation of $f_K(\cdot)$ and $f_K{}^{-1}(\cdot)$ using AES.



$f_K(\cdot) = HMAC_K$(AppId || SK_u)
where K is key, input is (AppId || SK_u) and output is Kh

Fig. 4: Instantiation of $f_K(\cdot)$ using HMAC.

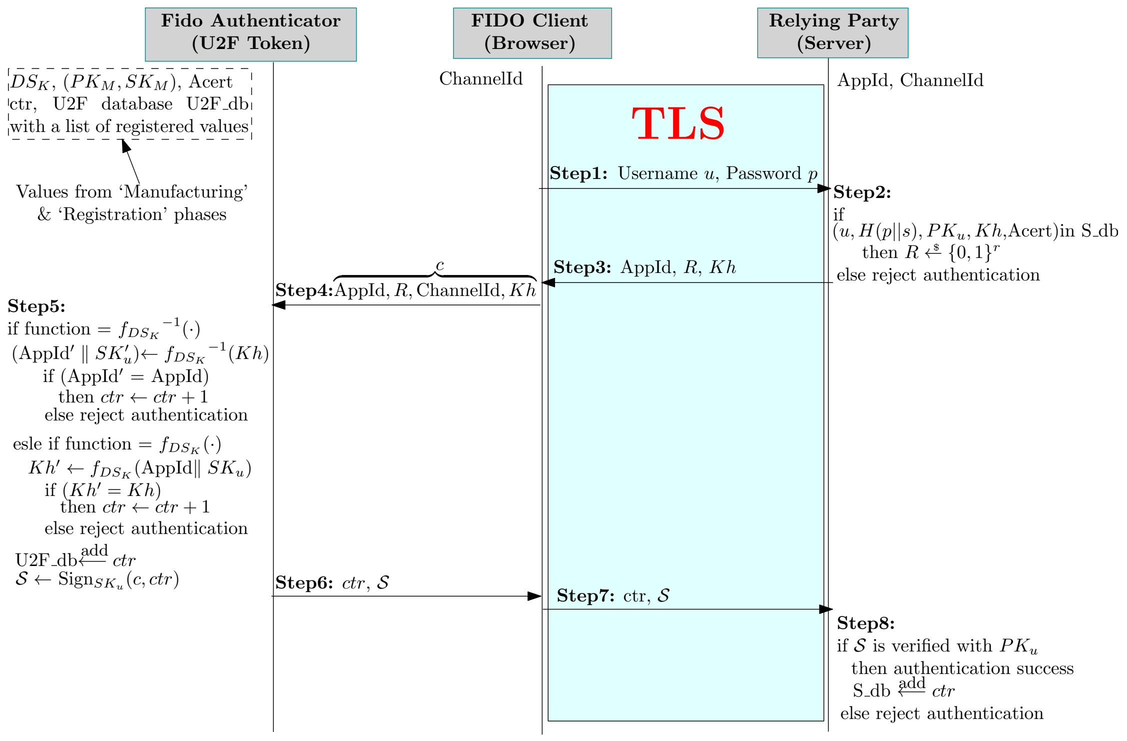

Fido Authenticator (U2F Token)
FIDO Client (Browser)
Relying Party (Server)
DS_K, (PK_M, SK_M), Acert ctr, U2F database U2F_db with a list of registered values
Values from 'Manufacturing' & 'Registration' phases
ChannelId
AppId, ChannelId
TLS
Step1: Username u, Password p
Step2: if $(u, H(p||s), PK_u, Kh$,Acert)in S_db then $R \xleftarrow{\$} \{0,1\}^r$ else reject authentication
Step3: AppId, R, Kh
Step4: $\overbrace{\text{AppId}, R, \text{ChannelId}, Kh}^{c}$
Step5:
if function = ${f_{DS_K}}^{-1}(\cdot)$
(AppId′ ∥ SK_u')← ${f_{DS_K}}^{-1}(Kh)$
if (AppId′ = AppId)
then $ctr \leftarrow ctr + 1$
else reject authentication
esle if function = $f_{DS_K}(\cdot)$
$Kh' \leftarrow f_{DS_K}$(AppId∥ SK_u)
if $(Kh' = Kh)$
then $ctr \leftarrow ctr + 1$
else reject authentication
U2F_db$\xleftarrow{\text{add}}$ ctr
$\mathcal{S} \leftarrow \text{Sign}_{SK_u}(c, ctr)$
Step6: ctr, $\mathcal{S}$
Step7: ctr, $\mathcal{S}$
Step8: if $\mathcal{S}$ is verified with PK_u then authentication success S_db $\xleftarrow{\text{add}}$ ctr else reject authentication

Fig. 5: **U2F Authentication Phase.** The key DS_K, the asymmetric keypair(PK_M, SK_M) with attestation certificate Acert which includes PK_M and counter 'ctr' are provided at the manufacturing phase. The U2F_db contains all the registered values from registration phase. The server database S_db contains all the required values generated at the registration phase corresponding to a registered username. R is a r bit random number generated by the server. AppId is the URL of the server. ChannelId is the TLS ChannelId. The value Kh is the Keyhandle. The verification of received Kh for both the cases of Step5 is shown in Fig. 3 and 4. 'ctr' is the counter used to record the number of authentication requests successfully satisfied by the U2F token.

2. The U2F specification [13] states that “The relying party’s dependence on passwords is reduced. The password can even be simplified to a 4 digit PIN. End users carry a single U2F device which works with any relying party supporting the protocol”. Therefore U2F claims to provide enhanced security even with weak password. At client side, an attacker with access to a different U2F token may register U2F authentication for an account of a third party without his notice. This is possible with the assumption that the password of a user account is known to the attacker (or the password is easy to guess). This way the legitimate user would be prevented to access his own account for not having the registered token with him. This attack is difficult to prevent as it needs the passwords to be difficult to guess and known only to its user.
3. Another possibility of DoS attack is when the attacker gets access to the U2F token of a legitimate user. Since any hardware token has a fixed and limited memory, an attacker may attempt to exhaust all of it. For example, an attacker can register multiple websites of its own choice, leaving no storage for any other request. As there is no way to know the list of websites registered under a U2F token by its user, it is difficult for a legitimate user to notice this attack. The user can only receive denial for new registration requests. It shows that keeping the U2F token safe is mandatory to resist this DoS attack.

3.2 Parallel Signing Request attack

Consider a user interacting with two U2F enabled web pages [7] and assume that an attacker can detect the web page with which the user is interacting at a given instance. In this case, the attacker can synchronize the two U2F operations initiated for two different websites by the user. As per the specification of U2F v1.0, there is no way to notify the user about which operation is performed by the U2F token. The user merely performs a button-touch for both ‘Registration’ and ‘Authentication’ processes since only the user-consent is mandatory for the execution of both these operations.

Suppose a website is requesting for U2F authentication and is visible at the foreground of the user’s display, and at the same time a transaction page (initiated by the attacker) is in the background (opened as an iframe). The user provides his consent for the displayed website while the first request that reaches to the U2F could be for confirming the transaction. This scenario shows a successful execution of a transaction without the knowledge of the user. Therefore the second factor authentication with U2F is not safe for monetary transactions and such transactions could be at high risk.

With growing popularity of Internet banking, cases of online frauds have also increased manifold resulting in considerable financial losses [22]. The banking frauds have increased 93% in 2009-2010 [23], and 30% in 2012-2013 [24] and these are ever growing. Internet banking or online transaction frauds are difficult to analyze and detect because the attack techniques are rapidly evolving while their detection is usually delayed [22]. Therefore, transaction-nonrepudiation[4] is an essential property for any secure authentication scheme.

FIDO alliance has already mentioned in the protocol specification that U2F does not claim transaction non-repudiation. Therefore, the scenario described above is technically not an attack on U2F. However, existing second factor authentications, such as SMS or OTP do provide this assurance. Even the version U2F v1.1 recommends to have notification for each operation but it is not mandatory. Therefore, with the current specifications, U2F is not a secure solution for password-based authentication. Augmenting OTP and U2F together can protect the transaction, however it adds overhead and contradicts one of the goals of U2F which is to be user-friendly. As U2F lacks transaction non-repudiation, it is not a secure solution to protect password-based authentication.

3.3 Phishing attack

The concept of the phishing attack was first introduced in 1987 at Interex conference by Jerry Felix and Chris Hauck [26,27]. NIST in [28] defines phishing as ‘a technique which refers to use of deceptive computer-based means to trick individuals into disclosing sensitive personal information’. Commonly, it is a form of social engineering that uses email or malicious websites to solicit personal information from an individual or company by posing as a trustworthy organization or entity.

[4] Nonrepudiation is the assurance that a participating entity in a protocol cannot deny an action done by it later [25].

U2F claims to be phishing resistant solution and two different scenarios prove the claim by showing how phishing can easily be detected at the time of authentication phase of the U2F protocol [7]. In one scenario, when the U2F is registered with a genuine website (AppId) and at authentication phase if a phishing website (AppId′) forwards the Keyhandle and challenge from that genuine website (corresponding AppId), the U2F can easily detect the phishing attempt with origin mismatch. Following Fig. 5, the scenario can be described as the U2F receives AppId′ at Step4 and the Keyhandle provides AppId at Step5 and hence both AppIds differ in value. Hence phishing is detected at the U2F token. In the second scenario, if an attacker forwards challenge (R) from a genuine website and Keyhandle (Kh') from a phishing website, U2F token signs the challenge with the corresponding private key of the received Keyhandle. If the signature is forwarded to the genuine website, it will reject it as the signature would not be verified (Step8 of Fig. 5). Therefore, because of the signature and origin specific Keyhandle, existing approaches for phishing attack fail against the U2F protocol.

3.4 Man-In-The-Middle (MITM) attack

In the MITM attack, the common scenario involves two legitimate endpoints (victims), and a third party (attacker) [29]. The attacker has access to the communication channel between two endpoints, and can eavesdrop, insert, modify or simply replay their messages. According to [29], the term 'MITM attack' was first mentioned by Bellovin et al. in [30].

The U2F specifications [13,14] claim that this technique prevents MITM attack where the MITM attacker intermediates between the user and the webserver. As per the U2F specifications, the MITM attack is possible when the following two conditions are satisfied.

1. MITM attacker is able to get a server certificate for the origin name issued by a trusted CA, and
2. Channel IDs are not supported by the browser.

However the MITM attack can be carried out even when these conditions are not satisfied, contradicting the developers claim. In [31], an MITM attack is shown on TLS with Channel ID support which is based on the following conditions.

1. MITM attacker is able to get a server certificate for the origin name issued by a trusted CA, and
2. Channel ID is supported by the browser.

The attack is similar to the attack in [32] which is also applicable to the U2F based authentication and called 'Man-In-The-Middle-Script-In-The-Browser (MITM-SITB)' [31]. The MITM-SITB attack works as follows. It assumes that TLS Channel ID is supported by the browser and the communicated webserver possesses valid/invalid certificate as it works with invalid certificates as well. This is due to the fact that the web browser only sends warning of invalid certificate to the user and proceeds according to the user action. In many cases, users ignore such warnings. Hence, even invalid certificate can allow the attacker to launch the attack. The details of the attack are provided next.

We assume that an attacker knows the website request of the user in advance and hence compromises the DNS server to divert the route of that legitimate website to a malicious website which contains same origin address (URL) but which is hosted on a different server (with different IP address). Next, when the user initiates authentication to the compromised website using U2F, it establishes a TLS connection with the malicious website. The malicious website pushes a malicious JavaScript code to the browser, without the notice of the user browser, and terminates the connection. The browser re-establishes a fresh TLS connection but with the legitimate website, as the attacker places the legitimate IP at DNS server, for subsequent communication with the website. The attacker becomes passive after injecting the code and the user authentication is performed over the connection between the browser and the legitimate website. Browsers commonly support 'Single Origin Policy (SOP)', i.e., the webpages can interact with each other only if they belong to the same origin (URL). The browser thus prevents interaction between multiple websites with different origins (URL's). Since the injected malicious code contains the origin of the legitimate website, the browser allows the code to access the webpages of the legitimate website. It is difficult to detect this attack as the TLS channel ID will be the same for both the browser and the server. The U2F only verifies the Channel ID of the browser which is origin-specific and hence can not detect the attack.

Real world attacks are multi faceted. All the above attacks are based on standard attacks employed on browsers and internet traffic, and none of these attacks require tampering of the U2F token or exploiting the U2F protocol. However, there are other class of attacks which can exploit the U2F token/protocol. One such class of attacks is the side channel attack. In the next section, we describe an attack on the U2F protocol which is based on the powerful side channel analysis. When a cryptographic device can get in the hands of the adversary or otherwise become easily accessible then one needs to pay close attention to side-channel attacks. The U2F solution is based on a hardware token (in common practice) and hence can easily be accessible to the attacker. As side channel attacks are easy to implement and the most successful cryptographic attacks to compromise secrets, it is of utmost importance to protect U2F from side channel attacks.

4 Possible Side Channel Attacks on U2F Protocol

The side channel attack is one of the most powerful attacks since the seminal work of 1996 by Paul Kocher [10]. In this class of attack, information is gained from the physical implementation of a cryptosystem [33]. There exists different forms of leakage under side channel. For example, timing information [10], power consumption [11], electromagnetic leaks [12] etc., provide information about the secrets that can be exploited. It is considered essential that the implementations of cryptographic algorithms are side channel resistant, specially when the device could be in the hands of the attacker. For some specific cryptographic operations, algorithmic countermeasures are available.

Kocher's first major contribution included exploiting the correlation of cryptographic secret keys with the time to perform certain cryptographic operations [10]. In a subsequent work, he showed how the secrets from a device can be extracted by utilizing the 'Differential Power Ananlysis (DPA)' attack [11]. DPA is a more advanced form of power analysis which allows an attacker to compute the intermediate values within cryptographic computations by statistically analyzing data collected from multiple cryptographic operations [10]. When a cryptographic operation is performed over a fixed key, computation with different input-output pairs can easily reveal the key. In the following section our proposed attack is based on the observation that different input-output pairs in U2F protocols are computed over a fixed key. Therefore, we claim that the DPA attack is possible on the implementation of U2F protocol. The 'Template attack' [15] is another powerful side channel attack which can break implementations secure against DPA attack but it is difficult to implement in practice [16]. The template attack requires significant pre-processing to be implemented. Specifically, it requires a great number of traces to be preprocessed before the attack can begin. Given the current state-of-the-art in side channel attacks, it is necessary to have a cryptographic implementation to be DPA resistant at the very least.

Next we explain how a DPA based side channel attack can be launched on the U2F protocol.

4.1 Side Channel Key Recovery attack on Device Secret Key DS_K

According to the specifications [13, 14], all U2F tokens contain a unique device secret key DS_K which is used to generate Keyhandle Kh as shown in Fig. 3 and 4. Specifically, at the time of registration, the U2F token computes $Kh \leftarrow f_{DS_K}(\text{AppId} \parallel SK_u)$ where AppId is the URL of the server and SK_u is the private key for the username u corresponding to the AppId. With the access to the U2F token, an attacker can collect multiple power traces for different input-output pairs operated under the function $f_{DS_K}(\cdot)$ with fixed but unknown key DS_K. This is possible when the attacker gets the U2F token and requests registration for different AppIds for U2F authentication. We assume that the communication between the browser and U2F token is easy to monitor by the attacker. This assumption is based on the fact that the information received or sent by the browser can be easily captured through the Document Object Model (DOM) tree created by the browser as explained in [34]. This allows the attacker to collect many different input-output patterns (the AppId and the corresponding Keyhandle) and thus enables him to launch a successful DPA attack to extract DS_K from $f_{DS_K}(\cdot)$. The computation at which the DPA can be launched at the time of registration of the token is shown in Fig. 6 as 'Attack point-I'.

The DPA attack has proven successful in efficiently attacking smart cards and many dedicated embedded systems which perform computations on the basis of a stored secret key [35]. In the classical form of DPA [36], an adversary collects multiple power traces and ciphertexts corresponding to a computation under a fixed

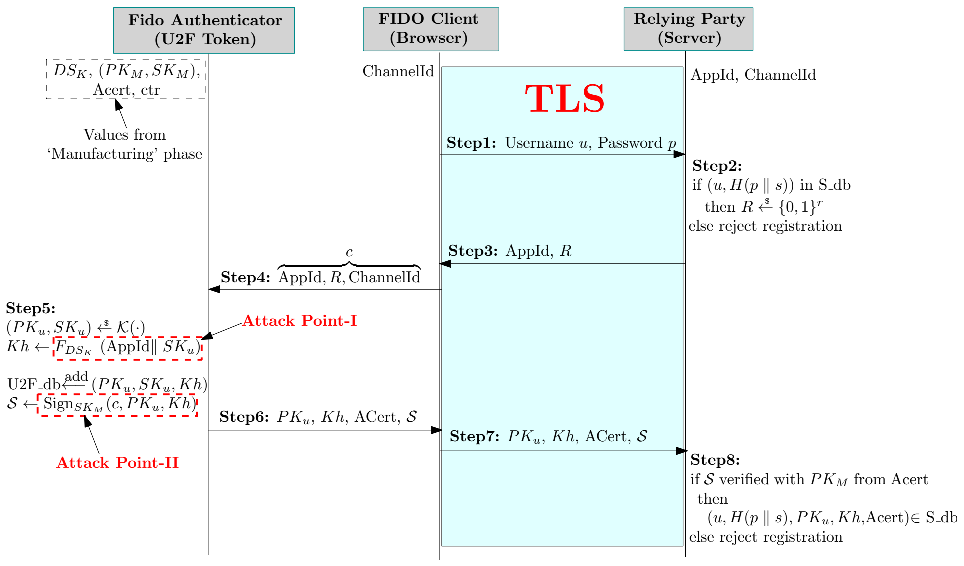

Fido Authenticator (U2F Token)
FIDO Client (Browser)
Relying Party (Server)
DS_K, (PK_M, SK_M), Acert, ctr
Values from 'Manufacturing' phase
ChannelId
TLS
AppId, ChannelId
Step1: Username u, Password p
Step2: if $(u, H(p \parallel s))$ in S_db then $R \xleftarrow{\$} \{0,1\}^r$ else reject registration
Step3: AppId, R
Step4: $\overbrace{\text{AppId}, R, \text{ChannelId}}^{c}$
Step5: $(PK_u, SK_u) \xleftarrow{\$} \mathcal{K}(\cdot)$
$Kh \leftarrow F_{DS_K}$ (AppId$\parallel$ SK_u)
Attack Point-I
U2F_db $\xleftarrow{\text{add}}$ (PK_u, SK_u, Kh)
$\mathcal{S} \leftarrow$ Sign$_{SK_M}(c, PK_u, Kh)$
Attack Point-II
Step6: PK_u, Kh, ACert, $\mathcal{S}$
Step7: PK_u, Kh, ACert, $\mathcal{S}$
Step8: if $\mathcal{S}$ verified with PK_M from Acert then $(u, H(p \parallel s), PK_u, Kh,$Acert$) \in$ S_db else reject registration

Fig. 6: **Side channel attack points at Registration Phase.** The key DS_K, the asymmetric keypair(PK_M, SK_M) with attestation certificate Acert which includes PK_M and the counter 'ctr' are from the manufacturing phase. AppId is the URL of the server. ChannelId is the TLS ChannelId. S_db is the server database. R is a r bit random number generated by server. $\mathcal{K}(.)$ is a random key generation function which generates the keypair (PK_u, SK_u). The function $f_{DS_K}(.)$ with key DS_K generates Keyhandle Kh as shown in Fig. 3 and 4. U2F_db is the database at U2F token. The Attack Point-I is to compromise the key DS_K and the Attack Ponit-II is to compromise the key SK_M.

key. It then chooses an intermediate value of the computation which depends both on the input and a small part of the secret key. The adversary then predicts the key bits and the corresponding power consumption. If the guessed key bits are correct then the predicted power consumption should match the observed one. On the other hand, if the key guess is wrong then the power consumption trace should be different from the observed one with a very high probability. As explained above, such an attack can be performed against the U2F token.

To prevent this attack, repeated use of fixed DS_K for different input-output (AppId-Keyhandle) pairs should be prevented. Use of session key is a standard solution [17] and our proposed countermeasure is based on the same idea.

4.2 Side Channel Key Recovery Attack On Attestation Private Key SK_M

As explained in Section 8.1, the attestation private key SK_M along with the certificate Acert, which contains PK_M signed by some trusted CA, are provided by the manufacturer to the U2F token at the time of manufacturing. This key SK_M is used as a signature key at the time of registration to prove genuineness of the U2F token to the server. The same attestation key is provided by the manufacturer to a large number of tokens generated during some specific time period to preserve anonymity of the tokens. The signature $\mathcal{S}$ =Sign$_{SK_M}$(AppId, R, ChannelId, PK_u, Kh) with signing key SK_M which is provided at the time of registration also leaks information about the key SK_M through side channels. The computation which may leak information about SK_M is represented as 'Attack Point-II' in Fig. 6. Therefore, an attacker with access to the U2F token can collect multiple traces for different inputs (containing different values of AppId, R, ChannelId, PK_u, Kh) and outputs (signature) pairs signed with a fixed but unknown key SK_M. It is easy to launch a DPA attack in a manner similar to the one explained in Section 4.1. For a particular token, all the signatures generated at the time of registration are produced using this fixed key SK_M. Moreover, as already mentioned, a large number of devices are using this same key SK_M. Therefore compromising a single SK_M on a U2F token is enough to compromise a large number of tokens which contain the same key SK_M. After extracting SK_M, an attacker can simulate the U2F token in a software program and use it to prove the possession of the token. It is almost impossible to detect which device has been compromised and therefore the only remedy is to invalidate all tokens with the compromised SK_M. Use of different public-private key pair for attestation can prevent this mass compromise.

There are two common approaches to develop a side channel attack resistant system. In the first approach, masking and other similar techniques are implemented at the hardware level to prevent the leakage of data which can be exploited by an attacker. In the second direction of work, algorithmic modifications are made such that the data which is available to the attacker becomes essentially random. In the following sections, we present an algorithmic solution to prevent/mitigate DPA based side channel attacks on U2F protocol.

5 Proposed Countermeasures

We describe a coutermeasure to prevent the recovery of the device secret key from side channel attacks. We use a counter-based re-keying process, a concept which was first analysed in [17] and then an efficient tree-based approach to implement it was described in [37]. We also suggest a minor modification in the approach of assigning attestation private key to the U2F tokens to mitigate the attack mentioned in Section 4 which allows recovery of the key SK_M.

5.1 Countermeasure to protect Device Secret Key

Our proposal to protect the device secret key DS_K from SCA is to generate session keys sequentially from DS_K, a concept known as "re-keying" in the literature [17]. Our counter based re-keying technique is explained in Algorithm 1. The function g of the proposed algorithm takes the previous session key and current counter as inputs and generates the current session key. The use of counter for re-keying is a natural way to avoid repetition of keys. However, counter-based re-keying technique has various drawbacks depending on the setting and application. In the case of client-sever model, which is the most common setting, this method suffers from synchronization issues when the session key is generated by the client and the corresponding counter is communicated to the server. The server can compute the same secret only when it knows the initial secret and the current counter. Therefore, a necessary assumption in this technique is that the initial secret is known to both the client and the server.

Considering the U2F protocol where sharing of keys with the server is not required, a significant problem is the computation overhead inside the U2F token. The overhead increases with the value of the counter since the computation starts from the initial counter value and sequentially proceeds till the current counter value. We address how to reduce the overhead of this sequential computation next.

Generally, there are two approaches to derive session keys. In the parallel approach, the ith session key K_i is generated as $g(K, i)$ when K is the initial secret and g is some function. In the sequential approach, the ith session key is generated by applying a function on the $(i-1)$th session key [17]. The sequential approach is better to avoid DPA as it changes the key at each execution depending on the previously generated key. In the parallel approach, every execution depends on the initial secret and potentially leak information about the initial secret via side channels. The sequential approach also provides forward secrecy, for instance, disclosure of the current key does not reveal the previously generated session keys. However, this approach is inefficient. The value of the counter reveals the number of computations required to derive the session key starting from the initial secret. If there is a significant difference in the current key counter and the initial counter, and if the previous values are not stored then huge number of computations are needed to reach the desired value. For example, to compute the i^{th} session key, it requires a single computation if the $(i-1)^{th}$ session key is stored. However, if the j^{th} session key is required to be re-generated and the last stored subkey is the i^{th} one, with $i \gg j$, then we need j computations starting from the initial secret to re-derive the j^{th} session key. This is a clear overhead if $i \gg j$.

We discuss how to reduce this computational overhead in our proposed techniques and also explain the usability of each proposal below.

We provide some practical approaches to instantiate the function g of Algorithm 1. We represent the function g as g(key, counter) where key is an n-bit secret value and *Counter* is an integer.

Algorithm 1 To generate session key from device secret key
Input: AppID, *Counter*, Device secret key DS_k, Previous session key, U2F database U2F_db
Output: A tuple (Keyhandle, *Counter*)

```
1: Global var $Counter$ initialized at 1;
2: $K_0 \leftarrow DS_K$
3: Set $i = Counter$
4: $(PK_u, SK_u) \overset{\$}{\leftarrow} \mathcal{K}(.)$
5: $K_i \leftarrow g(K_{i-1}, i)$
6: $Kh_i \leftarrow f_{K_i}(\text{AppId} \mid\mid SK_u)$
7: U2F_db $\xleftarrow{add} (Kh_i, i)$
8: $Counter = Counter + 1$
9: return $(Kh_i, i)$
```

Proposed Technique 1: In this case $g(\text{key}, Counter) = HMAC_{key}(Counter)$. The function g is the standard HMAC with any cryptographically secure hash function, such as SHA256. The key is the previous n-bit session key K_{i-1} and the input is the current m-bit counter i which produces the next n-bit session key $HMAC_{K_{i-1}}(i)$ as shown in Fig. 7. Initially a system-wide global variable *Counter* i is initialized at value one. It gets incremented at each registration request satisfied by the U2F token. The *Counter* is handled internally by the token therefore no external control is possible. As *Counter* i is incremented for each new registration, we get different session keys K_i. This K_i is then used for computing Keyhandle $Kh_i \leftarrow f_{K_i}(\text{AppId} \parallel SK_u)$. Therefore, the value i is associated with the value Kh_i and the pair (Kh_i, i) is stored internally inside a database of U2F token represented as U2F_db. At the time of authentication, the value of Kh_i is verified only if the correct value of i is provided.

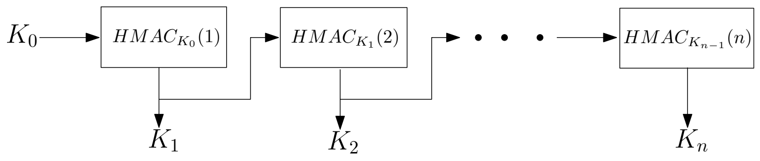


Fig. 7: **The function** $g(K_{i-1}, i) = HMAC_{K_{i-1}}(i) = K_i$ where $K_0 = DS_K$. It generates n session keys sequentially.

.

Security: The function $HMAC(\text{key}, Counter)$ works as a pseudorandom number generator [5]. Therefore, it increases the lifetime of the device-secret key and achieves the provable security gains in practice [17]. This sequential approach of key generation also provides forward secrecy, i.e., disclosure of the current key does not reveal the previously generated session keys. To protect future keys, use of random nonce instead of *Counter* is advisable. In that case nonce value should not be shared outside the U2F token and the keyhandle can be used as the index of the database inside the U2F token.

Usability: Our aim is to achieve DPA resistance and at the same time have efficient computation inside the U2F token. The *Counter* based approach avoids repeating the same session key for each registration but it is a huge computational overhead to compute the session key every time. When the current *Counter* is much larger than the initial value, it increases the required number of computations significantly if only the initial secret is available.

The computation of every new session key needs one extra hash computation if the previous session key is stored. Therefore, at the time of registration, the overhead is just one hash computation. We assume that the current counter value is i, the corresponding session key K_i is stored at the U2F token, and at this instance,

[5] A pseudorandom number generator is an algorithm which generates a sequence of numbers that are computationally indistinguishable from true random numbers [38].

an authentication request for j^{th} registered value with $j << i$ reaches the token. As $(j-1)^{th}$ session key is not stored, it requires j computations to verify the compute the session key starting from the initial counter. Therefore, at the time of authentication, if the current counter value is much larger than the initial counter, the computation overhead is also significant to re-derive the corresponding session key (assuming that the computation starts with the initial secret). However, we can greatly reduce the overhead of authentication phase. One possible solution is estimate what will be a good gap between the stored session keys, that is, every kth session key can be stored rather than only the first one, for a suitable value of k. Since the tokens have a limited memory, and the number of registrations are upper bounded by available memory, storing multiple session keys does not cause significant overhead. This is due to the fact that k is likely to be quite large as we explain next.

The value of k can be estimated by considering the time to perform a single HMAC computation and estimating the number of feasible HMAC computations that can be performed in an allowable time. The session keys can then be stored while maintaining a fixed gap between different counters. Keeping every kth session key, at most $k-1$ HMAC computations are needed to regenerate any session key, and this is within allowable time period due to the choice of k.

We assume that the computation of both Keyhandle and session key requires the same cryptographic primitive (HMAC). Since Keyhandle generation logic is already supposed to be implemented in the token, no extra implementation overhead is caused for the computation of session keys. We recommend to store the the Keyhandle and the corresponding *Counter* for each registered relying party at the U2F token. The requirement of this storage is explained in detail in section 7. Considering the existing storage capacity of any hardware token, space overhead due to (Keyhandle, *Counter*) pair inside the U2F token is not significant.

Proposed Technique 2: This technique to generate session keys is due to Kocher [37]. An initial secret key, which happens to be DS_K in the case of U2F, is used in conjunction with a *Counter i* to derive a session key K_i. Two invertible functions F_A and F_B, and their corresponding inverses ${F_A}^{-1}$ and ${F_B}^{-1}$ respectively, are also needed in this method. The function g of Algorithm 1 is instantiated by one of the 4 functions F_A, F_B, F_A^{-1}, or F_B^{-1}, depending on the value of counter i [37]. The technique is briefly explained next.

Fig 8 shows the sequence of states (shown as dots) indexed with counters starting from the initial value 1. A state with counter i represents the session key K_i and a method to derive it. The index depth D is an additional parameter, which is a predefined constant, and represents the ‘levels’ of session keys present. The value of D is fixed by estimating the possible number of session keys that are needed to be derived. For example, Fig 8 is for the case $D = 5$, meaning that five levels of key values are present. Each rectangular box in Fig. 8 represents a specific session key. Thus, multiple dots in a rectangular box represent different states sharing the same session key.

The top level (row 0) in the figure has a single box containing states 1, 31 and 61 all of which share the same value of session key. The next level (row 1) contains two boxes, both of which contain 3 states. The left box contains states 2, 16 and 30 which share the same value of session key. Similarly, the right box contains states 32,46 and 60 which share a common value of session key different from the left one. Similarly, the other levels and boxes represent different session keys. There is a unique session key per box.

There is only one box which has one node at level 0. There are 2 boxes containing 3 nodes each at level 1. Going forward, there are 4 boxes each of which contains 3 nodes at level 2. In general, all the levels N except the last one (that is $1 \leq N < D-1$) contain 2^N boxes each of which contains 3 session keys. The last level ($N = D-1$) contains 2^N boxes and each box has a single key. Thus, the total number of keys for a session key structure with depth D described here is given by Eqn. 1. For the example figure presented here, D is 5 and it results in 61 session keys according to this equation. These 61 states are seen in Fig. 8. It is clear from the description that any particular session key is used no more than 3 times within the structure.

$$\text{No. of session keys } \leq \; 2^{D-1} + \sum_{i=0}^{D-2} 3(2^i) = 2^{D-1} + 3(2^{D-1}-1) = 2^{D+1} - 3 \tag{1}$$

The session keys are derived iteratively following the relation given below.

$$K_1 = g(DS_K)$$
$$K_i = g(K_{i-1}) \;\text{ for } 2 \leq i \leq (2^{D+1}-3), \;\text{ where } g \text{ is as defined in Eqn. 2.}$$

$$g = \begin{cases} F_A & \text{if moving downwards from a parent node to the child node on the left} \\ F_B & \text{if moving downwards from a parent node to the child node on the right} \\ F_A^{-1} & \text{if moving from the left node to its parent} \\ F_B^{-1} & \text{if moving from the right node to its parent.} \end{cases} \tag{2}$$

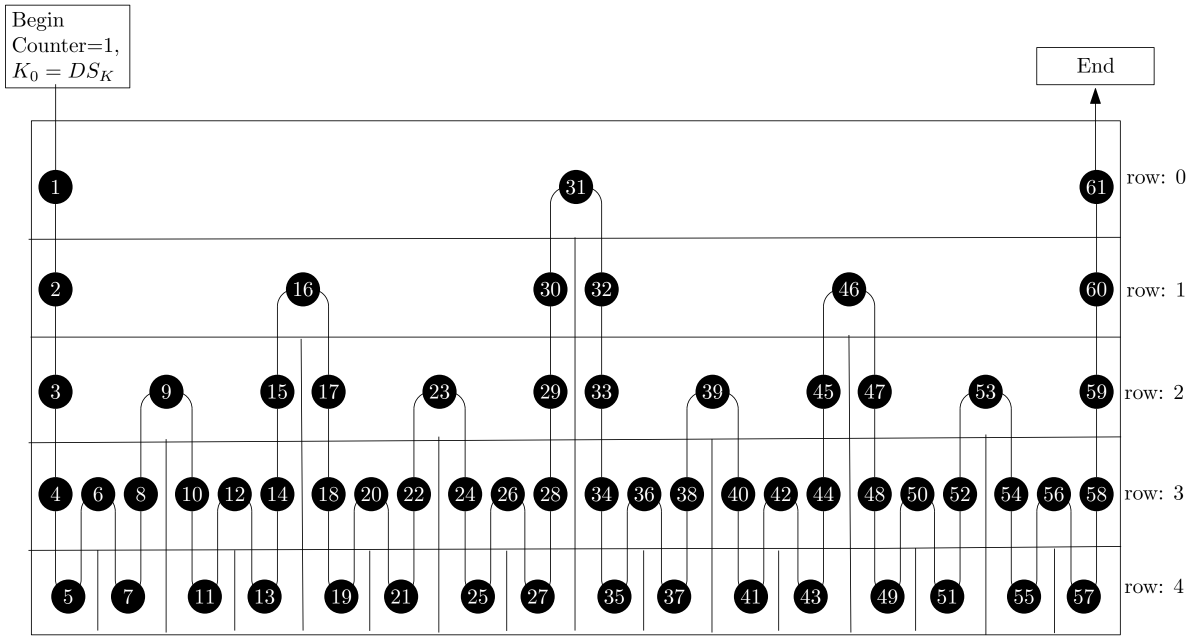


Fig. 8: Kocher's session key generation method with function $g(K_{i-1}, i) = K_i$ and index depth $D = 5$, taken from [37]. The function g of Algorithm 1 is instantiated by one of the 4 functions F_A, F_B, F_A^{-1} or F_B^{-1} in a sequence depending on the *Counter* i as explained in the text.

Security: This approach of re-keying can be seen as a hybrid approach incorporating sequential and parallel approaches [17]. The session key is derived by applying a series of computations depending on the value of the *Counter*. A session key is not repeated more than three times in this construction therefore it is hard to gather enough traces for leakage analysis of the session keys. The proposal requires the use of a reversible function and any secure block cipher can be used. The security analysis of the scheme in [17] shows that it is a secure approach.

Usability: The implementation needs four functions $F_A, {F_A}^{-1}, F_B, {F_B}^{-1}$ which can be easily instantiated with a single block cipher like AES computed with the previous session key to generate the current session key. At the time of registration, the computation for a session key requires one AES evaluation if the previous session key is already stored. This is a small computational overhead which is not an issue. At the time of authentication, the maximum number of AES evaluations is equal to the depth D, i.e. the complexity is $log(n)$ where n is the total number of possible session keys. When $D = 39$, the number of derived session keys is more than 1 trillion ($\approx 10^{12}$) which means that practically we never require more than 39 AES evaluations to derive a session key. This overhead is not significant if the AES-NI instruction set extension is available on the token. Current implementations of AES-128 on Intel processors supporting AES-NI can allow approximately 2^{29} operations per second [39].

With the assumption that the computation of Keyhandle requires the same block cipher (AES), no additional hardware/software is required for session key computations. Storage for the registered (Keyhandle, *Counter*) pair inside the U2F token is required. Considering the existing storage capacity of hardware token, this overhead is not significant.

5.2 Countermeasure to minimize the effect of Attestation Private Key Recovery

The signature function that uses the attestation private key also leaks information that can be captured through side channel analysis as explained in Section 4.2. Usually in public key infrastructure (PKI) [9], the verification of digital signature is satisfied by the use of certificate. Intuitively, it may seem that use of session key will prevent the side channel attack on attestation private key as well. However, getting valid certificate for each session key generated from attestation private key is not practical in asymmetric cryptosystem. Hence, it is difficult to prevent side channel attacks using the existing U2F design for signature generation. Therefore, we suggest a minor modification in the current approach for assigning the attestation private key to the U2F token. The approach can not prevent the recovery of the attestation private key but minimizes the effect of the attack. Before explaining our proposed recommendation, we explain the existing issues with attestation private key for U2F token.

Trust on FIDO The attestation certificate Acert to prove the genuineness of the U2F token is issued to the manufacturer by some trusted CA. The FIDO server maintains a centralized database of all valid certificates. One of the goals as mentioned in the specifications [13, 14] for U2F is to make it a standard solution. A standard should be acceptable to all. This means FIDO needs to acquire global acceptance with a trust on this centralized approach. However, it is difficult to develop trust globally. One possible solution could be a decentralized approach. If a government agency under each country takes the responsibility to maintain a server with trusted list of CAs and list of valid certificates, the citizens can be expected to trust it. The FIDO server should also include all the valid certificates and in that case the U2F approach could be used as a standard. To prove the genuineness of the token, a relying party can verify with country-level trusted server or the FIDO server or both. This can lead to a standard U2F solution which is globally accepted.

Genuineness or Anonymity? In the specifications [13,14], it is mentioned that a manufacturer of a U2F token provides an attestation certificate signed by some trusted CA along with the attestation private key, permanently written inside the token. The attestation certificate and signature with signing key proves the genuineness of manufacturer and not the token. This is because a large bundle of U2F tokens are provided with the same attestation private key and corresponding certificate to preserve anonymity of the token. However, if a single device is compromised, then all the tokens with the same identity are compromised. This can be prevented if each device has it's own attestation public-private key pair. This attestation certificate is required to show genuineness of the token. As U2F is a second factor authentication device, and the user association is already established with the username and password, there is no enhanced security obtained with anonymity of the token. Therefore, token should only prove its legitimacy which is the main goal of the tokens. Therefore, we see a trade-off between security and anonymity in the case of U2F protocol. Since the compromise of a single attestation private key impacts a large number of tokens, we believe that security should be preferred over anonymity.

Countermeasure to minimize the effect of Attestation Private Key Recovery The main idea behind the 'private key' of asymmetric cryptosystem is to keep it private with the owner and not to share it with any third party. However, the scenario of sharing same attestation private key with a huge number of tokens can be visualized as the manufacturer keeping its own private key for all the tokens manufactured by it. Therefore the manufacturer should not distribute the same attestation private key among a large number of U2F tokens. Instead, it should have its own private key and corresponding certificate signed by a trusted CA. A unique public-private keypair should be generated by the manufacturer for each token. To prove genuineness in this scenario, the manufacturer should sign the public key of the token. The signature by the manufacturer together with the certificate by a trusted CA including the manufacturer public key proves the genuineness of the U2F token. This two layers of signature prevents the sharing of private key and proves the genuineness of the U2F token which is the goal behind the attestation concept of U2F tokens. This also enhances the security as attacker has to forge two signatures now. The unique attestation private key is still vulnerable to the attack which is shown as Attack point-II in Fig. 6 but limited to target a single U2F token at a time instead of a large number of tokens.

5.3 Overview of the U2F protocol with Our Proposed Modifications

In this section, we briefly explain the U2F protocol incorporating the proposed countermeasure and the recommendation.

Our Modified U2F Manufacturing Phase: Unique Secrets for U2F Token This phase is executed at the time of manufacturing of the U2F token as shown in Fig 9. In this phase the U2F token is provided with a randomly generated unique secret key called Device Secret Key DS_K at Step1. At Step2, a unique public-private key pair (PK_T, SK_T) is generated from a key generation function $\mathcal{K}(\cdot)$ for the token. The asymmetric keypair of the manufacturer is represented as (PK_M, SK_M). At Step3, a certificate Acert which certifies the key PK_M (public key of the manufacturer) is provided to the U2F token thus verifying the signature $\mathcal{S}_M$ of the manufacturer. The Acert is issued by a trusted CA to the manufacturer. At Step4, the key PK_T is signed by the manufacturer using his secret key SK_M. The signature obtained in this step is denoted as $\mathcal{S}_M$. Since the corresponding public key of the manufacturer is certified by a trusted CA, the genuineness of the token is established by the chain of trust. Both DS_K and (PK_T, SK_T) are fixed and unique for a token. A global variable *ctr* initialized to zero is provided at Step5. This helps to detect cloning as explained in Section 2.3 during the authentication phase. A global variable *Counter* is provided at Step6 which is initialized to 1 and used to generate session keys from DS_K to prevent repeated use of DS_K at the following registration phase.

As can be seen from the above description, our proposal adds two extra steps (namely Step 2 and Step 6), and makes a minor modification in Step4 of the original protocol. The asymmetric key pair (PK_T, SK_T) supplied with each token solves the problem of sharing the private key which is described in Section 5.2 earlier.

Our Modified U2F Registration Phase: Use of Session Key The registration process is explained in Fig. 10. At Step1, the username and password are provided to the the server. On verification of the received values the server generates a r-bit random value R at Step2. At Step3, the value R and server AppId are sent to the browser. Adding the ChannelId, the browser forwards the received values to the U2F token at Step4. At Step5, the U2F token first generates the site-specific key pairs (PK_u, SK_u) applying a random key generation function $\mathcal{K}(.)$. The value of *Counter* is assigned to a variable i which gets incremented after each registration. It then computes the session key K_i following the steps of Algorithm 1. The session key generation function $g(K_{i-1}, i)$ can be instantiated with either the proposed hashed based approach or using Kocher's technique as explained in Section 5 and shown in Fig. 7 and 8 respectively. Next the Keyhandle Kh_i is generated from the function $f_{K_i}(.)$ following one of the approaches as explained in Fig. 3 and 4. Finally the token adds the values (PK_u, SK_u, Kh_i, i) in a database represented as U2F_db which is stored in the token. It computes a signature $\mathcal{S}$ with the key SK_T on the values (c, PK_u, Kh_i, i). At Step6 the token sends the values PK_u, Kh_i, i, Acert, $\mathcal{S}_M$ to the browser. The browser forwards the received values to the server at Step7. On receiving the values the server first verifies the signature $\mathcal{S}$ with PK_T after verification of $\mathcal{S}_M$ with Acert. If verified it keeps the values $(u, H(p \parallel s), PK_u, Kh_i, i$, Acert, $\mathcal{S}_M)$ in its database, else it rejects the registration request.

As can be seen from the above description, our proposal modifies Step5 by adding the computation of session key K_i and updating the U2F database with values (Kh_i, i). The signature $\mathcal{S}$ is performed by the token specific signing key SK_T instead of SK_M. The modifications of Step5 correspondingly make a minor modification in Step6 and Step7 of the original protocol. At Step8 the genuineness of token is established by verifying two levels of signature. The use of session key for the generation of each keyhandle prevents the side channel attack mentioned in Section 4.1.

Our Modified U2F Authentication Phase At the time of authentication, receiving the request with username u and password p at Step2, the server checks for the corresponding registered values. If registered, the server sends the values AppId, Challenge R, Keyhandle Kh_i and *Counter* i to the browser at Step3. The bowser forwards the received values along with TLS ChannelId to the U2F token at Step4. At Step5 the U2F token first checks for the entry of the pair (Kh_i, i) in its database. If the entry is verified, it re-derives the session key K_i from function $g(K_{i-1}, i)$ following the re-keying technique explained in Section 5 and then verifies the received value Kh_i. If the function is ${f_{K_i}}^{-1}(\cdot)$, the U2F token computes ${F_{K_i}}^{-1}(Kh_i)$ which provides the output (AppId$' \parallel SK'_u$). It then verifies if AppId$'$ = AppId. Else for the case when the function

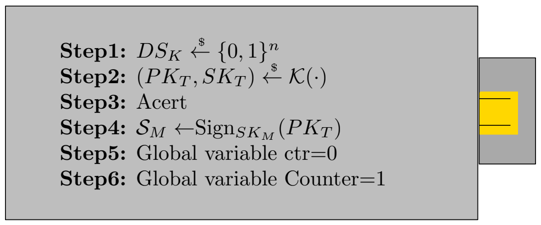

Step1: $DS_K \xleftarrow{\$} \{0,1\}^n$
Step2: $(PK_T, SK_T) \xleftarrow{\$} \mathcal{K}(\cdot)$
Step3: Acert
Step4: $\mathcal{S}_M \leftarrow \text{Sign}_{SK_M}(PK_T)$
Step5: Global variable ctr=0
Step6: Global variable Counter=1

Fig. 9: **Our Modified U2F Manufacturing Phase**, DS_K is a randomly generated n bit device secret. $\mathcal{K}(.)$ is a random key generation function to generate a unique asymmetric keypair (PK_T, SK_T) for the token. The asymmetric keypair of the manufacturer is (PK_M, SK_M). Acert is the certificate issued by a trusted CA which contains PK_M. $\mathcal{S}_M$ is a signature on PK_T by the manufacturer with signing key SK_M. At registration phase, a signature with SK_T is verified with PK_T which is again verified from $\mathcal{S}_M$ and Acert. The global variable ctr is initialized at value zero which is incremented after each successful authentication as explained in Section 2.3. The global variable *Counter* is initialized at value one which is incremented after each successful registration as explained in Section 2.2.

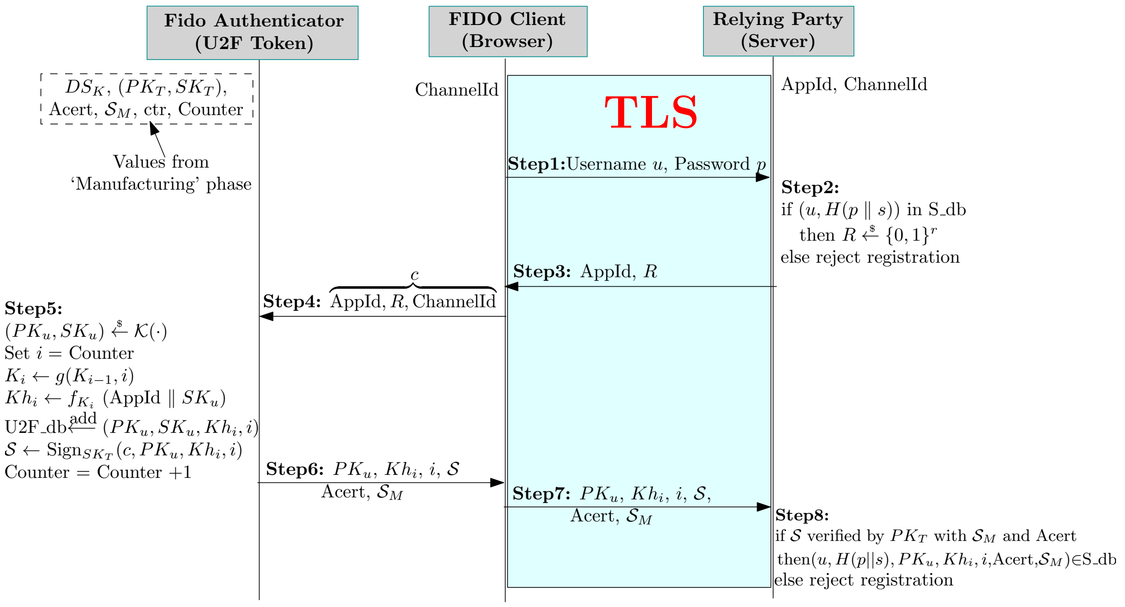
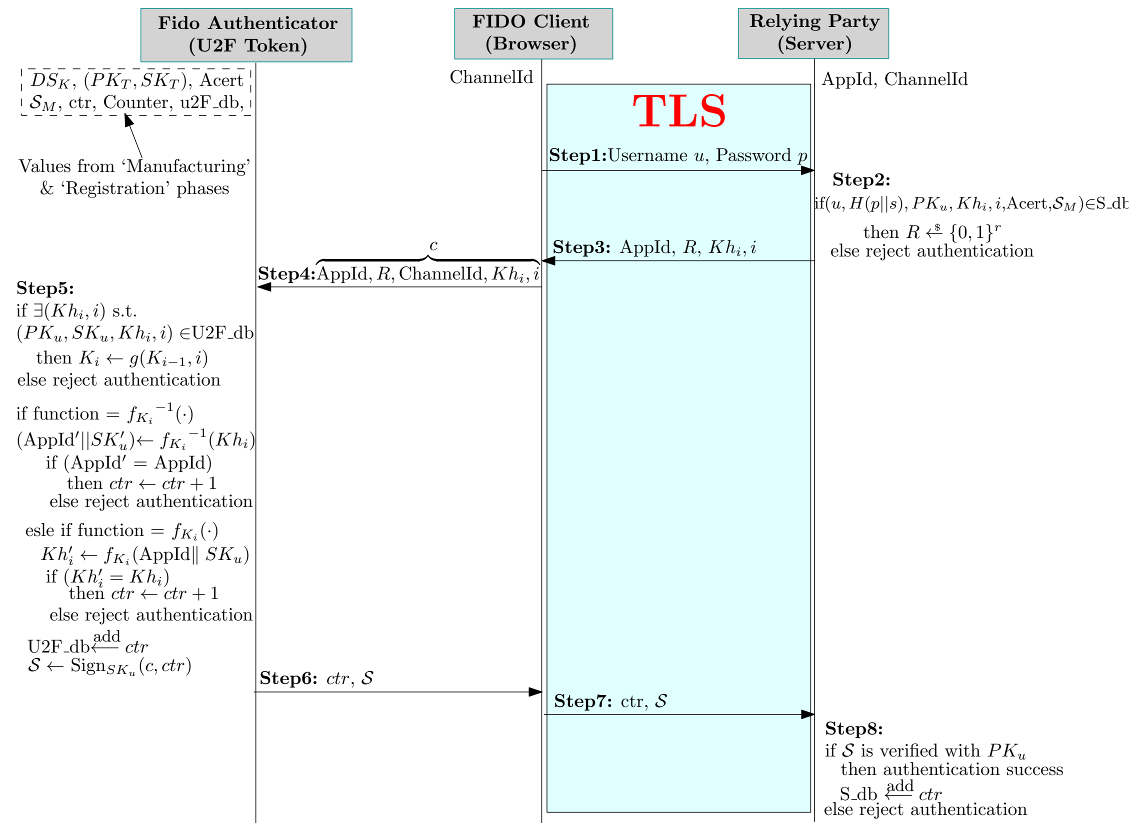

Fido Authenticator (U2F Token)
FIDO Client (Browser)
Relying Party (Server)
DS_K, (PK_T, SK_T), Acert, $\mathcal{S}_M$, ctr, Counter
Values from 'Manufacturing' phase
ChannelId
TLS
AppId, ChannelId
Step1: Username u, Password p
Step2: if $(u, H(p \parallel s))$ in S_db then $R \xleftarrow{\$} \{0,1\}^r$ else reject registration
Step3: AppId, R
Step4: $\overbrace{\text{AppId}, R, \text{ChannelId}}^{c}$
Step5: $(PK_u, SK_u) \xleftarrow{\$} \mathcal{K}(\cdot)$ Set i = Counter $K_i \leftarrow g(K_{i-1}, i)$ $Kh_i \leftarrow f_{K_i}$ (AppId $\parallel SK_u$) U2F_db$\xleftarrow{\text{add}}(PK_u, SK_u, Kh_i, i)$ $\mathcal{S} \leftarrow \text{Sign}_{SK_T}(c, PK_u, Kh_i, i)$ Counter = Counter +1
Step6: PK_u, Kh_i, i, $\mathcal{S}$ Acert, $\mathcal{S}_M$
Step7: PK_u, Kh_i, i, $\mathcal{S}$, Acert, $\mathcal{S}_M$
Step8: if $\mathcal{S}$ verified by PK_T with $\mathcal{S}_M$ and Acert then$(u, H(p||s), PK_u, Kh_i, i$,Acert,$\mathcal{S}_M)\in$S_db else reject registration

Fig. 10: **Our Modified Registration Phase.** the key DS_K, the asymmetric keypair$(PK_T$, $SK_T)$ with a signature $\mathcal{S}_M$ containing PK_T, Acert, counters 'ctr' and *Counter* are from the manufacturing phase. AppId is the URL of the server. ChannelId is the TLS ChannelId. S_db is the server database described in Section 6.4. R is a r bit random number generated by the server. $\mathcal{K}(.)$ is a key generation function to generate site-specific random keypairs (PK_u, SK_u). The function g generates the session key K_i as shown in Fig. 7 and 8. $f_{K_i}(.)$ is the function to generate Keyhandle Kh_i as shown in Fig. 3 and 4, U2F_db is the database at U2F token as explained in Section 6.3.

is $f_{K_i}(\cdot)$, it computes $F_{K_i}(\text{AppId} \parallel SK_u)= Kh'_i$ and verifies if $Kh'_i = Kh_i$. Both the cases are shown in Fig. 3 and 4 respectively. On verification for both cases, the token increments a counter, we represent it as 'ctr' which is initialized at value zero as mentioned in [13,14]. The value ctr is incremented and at Step6, a signature $\mathcal{S}$ including ctr are sent to the browser by the U2F token. The browser forwards the received values to the server at Step7. Finally at Step8 the server verifies the signature $\mathcal{S}$ with PK_u and after verification it updates the value of ctr else the authentication is rejected.

As can be seen from the above description, our proposal makes minor modifications at Step3 and Step4 by adding *Counter* i which was introduced at the modified registration phase. The verification process of Step5 needs one extra computation to re-derive the session key K_i. The signature $\mathcal{S}$, which was used to sign the values received from Step4 originally, now additionally signs the value i and the corresponding changes are reflected in Steps 6 and 7. The use of token specific private key SK_T mitigates the attack mentioned in Section 4.2 and the use of session key at Step5 helps in secure authentication of each registered user.



Fido Authenticator (U2F Token)
FIDO Client (Browser)
Relying Party (Server)
DS_K, (PK_T, SK_T), Acert
$\mathcal{S}_M$, ctr, Counter, u2F_db,
Values from 'Manufacturing' & 'Registration' phases
ChannelId
AppId, ChannelId
TLS
Step1: Username u, Password p
Step2:
if$(u, H(p||s), PK_u, Kh_i, i$,Acert,$\mathcal{S}_M) \in$S_db
then $R \xleftarrow{\$} \{0,1\}^r$
else reject authentication
Step3: AppId, R, Kh_i, i
Step4:$\overbrace{\text{AppId}, R, \text{ChannelId}}^{c}, Kh_i, i$
Step5:
if $\exists(Kh_i, i)$ s.t.
$(PK_u, SK_u, Kh_i, i) \in$U2F_db
then $K_i \leftarrow g(K_{i-1}, i)$
else reject authentication
if function $= f_{K_i}{}^{-1}(\cdot)$
(AppId$'||SK'_u)\leftarrow f_{K_i}{}^{-1}(Kh_i)$
if (AppId$'$ = AppId)
then $ctr \leftarrow ctr + 1$
else reject authentication
esle if function $= f_{K_i}(\cdot)$
$Kh'_i \leftarrow f_{K_i}(\text{AppId}\parallel SK_u)$
if $(Kh'_i = Kh_i)$
then $ctr \leftarrow ctr + 1$
else reject authentication
U2F_db$\xleftarrow{\text{add}} ctr$
$\mathcal{S} \leftarrow \text{Sign}_{SK_u}(c, ctr)$
Step6: ctr, $\mathcal{S}$
Step7: ctr, $\mathcal{S}$
Step8:
if $\mathcal{S}$ is verified with PK_u
then authentication success
S_db $\xleftarrow{\text{add}} ctr$
else reject authentication

Fig. 11: **Our Modified Authentication Phase.** The key DS_K and the asymmetric keypair(PK_T, SK_T) with signature $\mathcal{S}_M$ which includes PK_T, Acert, the counters 'ctr' and *Counter* are from manufacturing phase. The database U2F_db as explained in Section 6.3 contains all registered values from registration phase. S_db is the server database explained in Section 6.4. R is a r bit random number generated by the server. AppId is the ULR of the server. ChannelId is the TLS ChannelId. Kh_i is the Keyhandle corresponding to the *Counter* i. The function $g(K_{i-1}, i)$ generates the session key K_i as shown in Fig. 7 and 8. The function $f_{K_i}(.)$ is shown in Fig. 3 and 4. 'ctr' is the counter used to record the number of authentication requests successfully satisfied by the U2F token.

6 Design Rationale

In this section we analyse the role of both the counter values, namely *Counter* and 'ctr', and the databases S_db and U2F_db for the execution of our modified U2F protocol.

6.1 Role of *Counter*

The global variable *Counter* is initialized at 1 and gets incremented after each successful U2F token registration to a website. Specifically the value *Counter* is crucial to generate the value Keyhandle as shown in Fig. 10. The value of *Counter* signifies the number of entries in the U2F database U2F_db as explained in Section 6.3. As described in Fig. 10, counter i affects the generation of the session key K_i and subsequently the generation of the Keyhandle Kh_i. Thus, the key K_i is a function of the *Counter*. The pair (Kh_i, i) corresponding to a username u is maintained in both the server and the U2F token databases. At the time of authentication the server sends the pair (Kh_i, i) to the U2F token but before initiating any computations, the U2F token first checks for the valid entry of the (Kh_i, i) pair in its database. It proceeds only if the entries match.

Problem with invalid (Kh_i, i) pair: At the time of authentication, an attacker can provide any randomly chosen value of *Counter* i which will generate a specific session key K_i. The key K_i is fixed for this i but unknown to the attacker. Note that the attacker can not succeed in computing or guessing the Keyhandle Kh_i except with a negligible probability. If the Keyhandle was not being checked (as in the original protocol), an attacker can make multiple attempts with any counter i with randomly chosen (incorrect) Keyhandle Kh'_i and collect power traces to extract information about the session key K_i. This attack will yield information about the correct key K_i as this key depends only on the *Counter* i.

To prevent such a malicious attempt, it is required to allow only the legitimate (Kh_i, i) pairs to initiate authentication requests. Therefore, we recommend to keep all the registered (Kh_i, i) pairs in a database inside the U2F token. Since the U2F token is assumed to be a secure device [13], this database can be considered safe. When an invalid pair is provided to the token, the authentication request is refused. Hence, our proposed modification prevents the collection of traces by an attacker not having valid credentials. The problem of an attacker deliberately trying to overflow the database is not an issue with the U2F token and is discussed while describing technique 1 in Section 5.

6.2 Role of 'ctr'

The use of 'ctr' is proposed in original specifications of the U2F protocol [13, 14]. As explained in the specifications, 'ctr' can be a global or a local (specific to a website) variable. It gets incremented after each successful execution of authentication phase of the U2F protocol. Without loss of generality, we consider 'ctr' as a global variable. After each successful authentication request, the value of the 'ctr' gets incremented and the server keeps the updated value in its database. If the received value of 'ctr' at Step8 of Fig. 11 is less than the last recorded 'ctr' at the server database, it shows a possible cloning of the U2F token. Thus the use of the counter 'ctr' may help detect cloning of the U2F token. As mentioned in [13, 14], it is not a strong solution to detect cloning due to synchronization issues, false detection etc.

6.3 Role of U2F_db

The database at U2F token is represented as U2F_db. It contains the value generated at registration phase corresponding to each registration. For each registration, the U2F_db keeps a tuple containing site-specific keypairs (PK_u, SK_u) and the Keyhandle Kh_i corresponding to *Counter* i, for instance the tuple (PK_u, SK_u, Kh_i, i) as shown in Fig. 10. The value of *Counter* shows the total number of entries in the U2F_db. The use of database can not be optional to provide a secure authentication against side channel attack. To make the protocol side channel resistant, it is required to allow only the legitimate computations inside the U2F token. Therefore, valid entries are verified from the U2F_db before initiating any computation. At our modified authentication phase, information leakage is possible due to illegitimate computation providing wrong counter and Keyhandle pair (Kh_i, i) as explained in Section 6.1. This can ne prevented by verifying the existence of the pair (Kh_i, i) in the U2F_db before initiating the computation.

6.4 Role of S_db

The database at the server is represented as S_db. It contains the values generated at registration and authentication phases corresponding to each registered username. For each username u, a tuple including hash of the password with salt $H(p \parallel s)$, public key PK_u for u, Keyhandle Kh_i corresponding to *Counter* i, the certificate Acert, the signature $\mathcal{S}_M$ and the value of 'ctr' to record number of successful authentication corresponding to the U2F token is kept in the database. The tuple $(u, H(p \parallel s), PK_u$, Kh_i, i, Acert, $\mathcal{S}_M$, ctr) is as described in Fig. 11. At the time of authentication, if the server receives a 'ctr' from the U2F token which is less than the value of 'ctr' in S_db, it marks the U2F token as compromised. Thus, this helps detect a cloning attack even though it is not a perfect solution due to synchronization issues, false detection etc.

7 Evaluation

We evaluate our proposed modified U2F protocol considering different scenarios under the modified registration and authentication phases. We evaluate the protocol by considering different combination of values that can be tampered and their effect on the security of the overall system.

7.1 Modified Registration Phase

In this phase, the values forwarded by the browser to the U2F token at Step3 in Fig. 10 contain AppId, ChannelId and random challenge R. Only the value AppId is useful for the computations of Keyhandle which is used in the authentication phase.

Scenario-I (AppId, R, ChannelId) This is the case of authentic request for registration. Valid Kh_i would be registered when corresponding *Counter* is i. Multiple registration requests with legitimate inputs may reveal the signing key SK_T as discussed in Section 4 through side channel, however a single signature is safe.

Scenario-II (AppId$'$, R, ChannelId) This is the case when the authentic AppId is replaced with a tampered AppId$'$ at the time of registration request. The Keyhanlde at the U2F token would contain AppId$'$ and not AppId. Using this malicious approach, a user would get authenticated for fake AppId$'$ instead of the real AppId, without her even noticing it. When the real user attempts to access AppId, it leads to denial of service since the token has been registered for AppId$'$.

7.2 Modified Authentication Phase

We evaluate possible scenarios by tampering the values (AppId, Kh_i, i) corresponding to *Counter* i. These values are provided by the browser to the U2F token at Step4 of Fig. 11 and the verification in Step5 is performed using them.

Scenario-I (AppId, Kh_i, i): This is the case when valid inputs corresponding to *Counter* i are provided to the U2F token. As the inputs are not tampered, validation at Step5 and beyond succeeds. This is the scenario of successful authentication of the U2F token.

Scenario-II (AppId, Kh_i, i'): This is the case when *Counter* i corresponding to the Kh_i is tampered to a fake value i'. In this scenario, if the check for entries at U2F_db is not performed, incorrect computation corresponding to i' produces $K_{i'}$ and subsequently affects the computation of $Kh_{i'}$. The detection is possible with the verification of Kh_i having legitimate AppId as explained in Fig. 11. When multiple requests with i' and different legitimate (AppId, Keyhandle) pairs are provided, then some information about the key $K_{i'}$ may be leaked. By compromising $K_{i'}$, an attacker can reveal all the K_is where $i > i'$. Since the function $f_K(\cdot)$, as shown in Fig. 3, is invertible and K is the disclosed secret K_i, it also discloses the value SK_u (see Fig. 11). The signature at each authentication phase is performed with this signing key SK_u whereas at registration phase the signing key is SK_T. Therefore, it is easy to simulate functionality of the U2F token including SK_u without the possession of the token at the time of authentication. However, it is not possible to simulate the registration process as registration needs the signing key SK_T. The attack against the authentication phase described in this scenario can be thwarted by verifying the tuple (AppId, Kh_i, i) with the database U2F_db.

Scenario-III (AppId, Kh'_i, i): This is the case when Kh_i corresponding to *Counter* i is tampered to a value $Kh_{i'}$. In this scenario, if the check for entries at U2F_db is not performed, computations corresponding to i produce K_i and then Kh_i. The U2F token then verifies the received Kh'_i with the computed Kh_i. The mismatch in values signals authentication failure by the token. Therefore, the modification or tampering of Kh_i results in denial of service (DoS) attack after performing unnecessary computations.

Apart from the DoS attack, this scenario is similar to Scenario-II above. Multiple tampered Kh'_i corresponding to a fixed i may disclose the key K_i through side-channel attack. If the function $f_K(\cdot)$, as shown in Fig. 3, is invertible it further discloses SK_u. If the modification of Kh_i can be detected early then such side channel disclosure can be prevented or limited. Verifying the input tuple with the valid entries of the U2F database is required to disallow tempered inputs.

Scenario-IV (AppId, Kh'_i, i'): This is the case when both the Keyhandle and the *Counter* values are tampered. Without the check for valid entries, both the scenarios described in II and III are possible. Verification of inputs with the U2F database is required in this case as well.

Scenario-V (AppId$'$, Kh_i, i): This is the case when AppId corresponding to valid registered Kh_i, i is tampered to another value AppId$'$. This modification fails to satisfy the verification in Step5 of Fig 11. Hence, this scenario causes a DoS attack since the registration data contains AppId which does not match with the tampered AppID$'$. Thus storing the tuple (AppId, Kh_i, i) at the U2F database is required to prevent the DoS attack.

Scenario-VI (AppId$'$, Kh_i, i'): This is the case when *Counter* i corresponding to the Keyhandle Kh_i is tampered to another value i' and AppId is modified to AppId$'$. In this scenario, if the check for entries at U2F_db is not performed, incorrect computations corresponding to i' produce $K_{i'}$ and subsequent computations produce $Kh_{i'}$. This leads to denial of service to a valid U2F token.

Moreover, these unnecessary computations leak information about $K_{i'}$ via side channels. Multiple such attempts with valid Keyhandle but incorrect i' may compromise $K_{i'}$ and an attacker can reveal all K_is where $i > i'$. This may also disclose the value of SK_u as explained in Scenario-II earlier. Hence, software simulation of the U2F authentication phase is feasible. Therefore, verification of the inputs with U2F database is required to prevent this scenario.

Scenario-VII (AppId$'$, Kh'_i, i): This is the case when Kh_i corresponding to *Counter* i is tampered to value Kh'_i and AppId to AppId$'$. In this scenario, if the check for entries at U2F_db is not performed, computation corresponding to i produces K_i and subsequently Kh_i. The detection is possible with the check that $Kh_i \neq Kh_{i'}$. This is a case of DoS attack and causing unnecessary computations. As explained earlier in Scenario-II, these unnecessary computations may reveal keys K_i and SK_u. Verification with database is needed to prevent this case.

Scenario-VIII (AppId$'$, Kh'_i, i'): This is the case when all the three values, the Keyhandle, the *Counter* and the AppId are tampered. Without the check for valid inputs, both the Scenarios II and III explained earlier are possible. Further, repeated inputs with incorrect i' may leak information about $K_{i'}$ and all K_is where $i > i'$. Similar to the Scenario-II, SK_u may be disclosed when function, as shown in Fig. 3, is ${f_{K_i}}^{-1}(.)$ and correct Kh_i corresponding to i is known. This allows an attacker to simulate future authentications by the token. Verification of the inputs with a database is required to prevent this case as well.

Scenario-IX (AppId, Kh_i, i, R, ChannelId): The random value R is fixed for a particular session but changes at each authentication request. When AppId, Kh_i, and i are genuine, then all the computations of Step5 in Fig. 11 are verified. Afterwards, signature $\mathcal{S}$ is computed utilizing R.

Since R is a one-time random input, an attacker with access to the U2F token can initiate multiple computations corresponding to the legitimate AppId, Kh_i and i values. The attacker can thus generate signatures on different inputs (as R varies) with the same secret key SK_u. Collection of power traces corresponding to different input-output pairs again leaks information about this key SK_u which may be extracted using side channel attacks.

This scenario describes an inherent limitation of the U2F protocol and the only way it can be prevented is to restrict the use of the token by an attacker with all valid inputs with multiple random values of R.

7.3 Limitations of U2F

At the authentication phase of the U2F protocol, a signature $\mathcal{S}$ is obtained with site-specific signing key SK_u as shown at Step5 of Fig. 11. The inputs to the signing algorithm are fixed except R which is randomly generated at each authentication request initiated by the token. As explained above, in Senario-IX, this changing R provides different input-output pairs computed under the same secret key SK_u to an attacker, who may utilize side channel attacks to learn this secret. For example, it is easy for an attacker to initiate a legitimate authentication request multiple times and compromise the key. Then it can simulate the legitimate authentication without possession of the token. This attack is not possible to prevent with existing U2F solution. The possible countermeasure is to restrict computations and release of data only to the genuine users of the token. One possibility is to take biometric input to first authenticate a legitimate user and then initiate further computations. Therefore, the scenario above shows a limitation of the U2F solution. Another limitation is transaction non-repudiation as mentioned in the specifications [13, 14] and explained in Section 3.

8 Possible Side Channel Attacks on Universal Authentication Framework Protocol and Mitigation

The Universal Authentication Framework (UAF) protocol is an authentication protocol proposed by FIDO alliance which supports biometric authentication to provide a unified and extensible authentication mechanism that supplants passwords [8]. It involves five entities as shown in Fig. 12. We describe these entities next. The first entity is the user of the protocol and the second entity is the UAF authenticator which is similar to a U2F token but also includes a database and a verifier to authenticate the biometric data of users. The next entity, Authenticator Specific Module (ASM) is the software to work as an interface between the UAF authenticator and the browser. The fourth entity, the user browser is called the FIDO client. The fifth entity includes both the web server and the FIDO server. The web server is also called the relying party. The FIDO server is an additional server which runs on the relying party's infrastructure. This FIDO server maintains a database which includes information coming from the UAF authenticator. Similar to the U2F protocol, the UAF protocol can be explained in three phases:

1. *The UAF Manufacturing Phase* that assigns device secret key DS_K and an asymmetric keypair (PK_M, SK_M) to the UAF authenticator, apart from performing additional steps to handle the biometric input.
2. *The UAF Registration Phase* that registers a user for future authentication.
3. *The UAF Authentication Phase* that authenticates the registered users.

Following is the description of the three phases of the UAF protocol.

8.1 UAF Manufacturing Phase

This phase is executed at the time of manufacturing of the UAF token as shown in Fig 13. In this phase the UAF token is provided with a randomly generated unique secret called Device Secret Key DS_K at Step1. At Step2, a public-private key pair (PK_M, SK_M) is provided to the token. The (PK_M, SK_M) keypair is not a one-time generation. Once randomly generated from a key generation function, the manufacturer provides the same (PK_M, SK_M) pair to all the tokens of a specific model manufactured by it. Note that the manufacturer can keep the same key pair for all the tokens manufactured by it (that is, it can treat all the tokens manufactured by it as the same model). At Step3, a certificate Acert issued by a trusted CA which includes PK_M (the public key of the manufacturer) is provided to the token. During the registration phase, the signature with SK_M is verified with PK_M which is extracted from Acert by the server. This verification proves the genuineness of the UAF token. Therefore, both DS_K and (PK_M, SK_M) are fixed for a token while the same (PK_M, SK_M) values are provided to multiple tokens which preserve the anonymity of the token. An integer counter 'ctr' is provided at Step4. This 'ctr' is initialized to value zero and gets incremented after each successful execution of the authentication phase.

8.2 UAF Registration Phase

The registration procedure is explained in Fig 14. Initially the username u and password p are communicated to the FIDO client at Step1 which is forwarded to the server at Step2. At Step3 the server verifies the values u and $H(p \parallel s)$ which is the hash of the password where s is the salt. On successful verification, the server triggers the UAF registration request to FIDO server at Step4. At Step5 the FIDO server generates a r bit random challenge R to differentiate each request. The FIDO server sends the UAF registration policy which

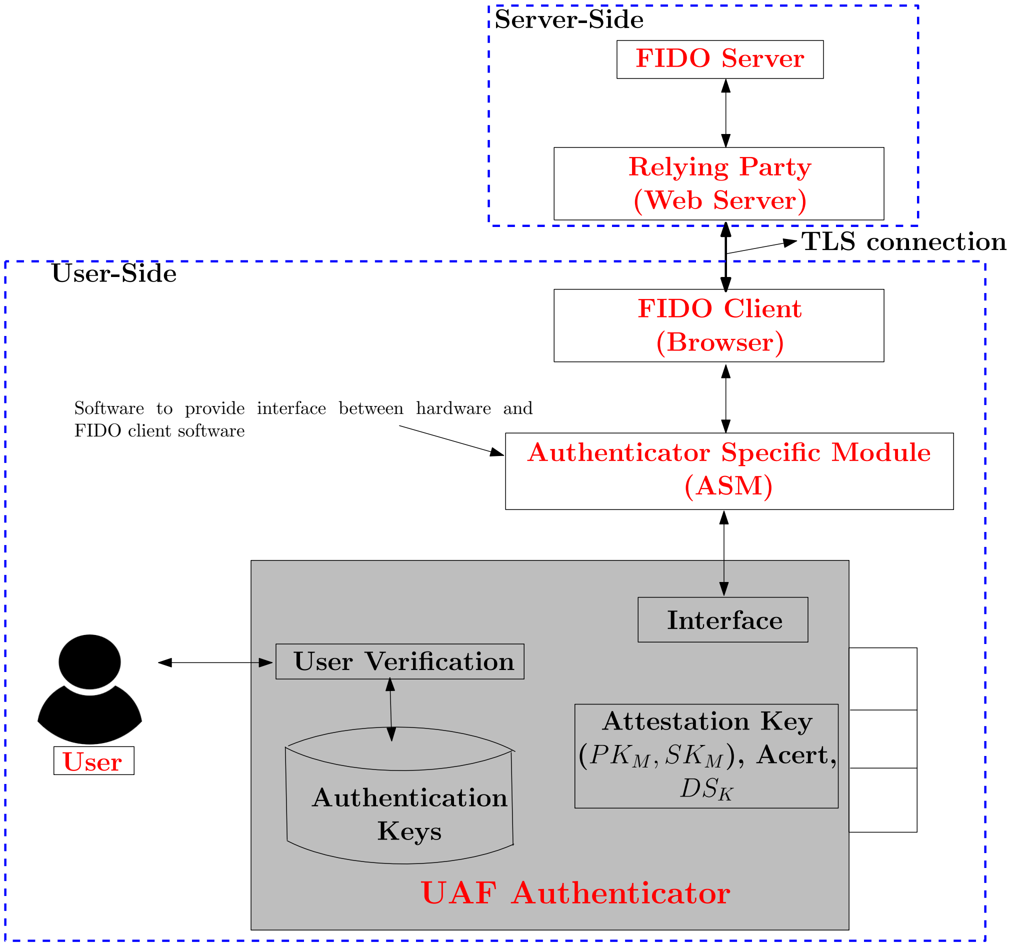

Server-Side
FIDO Server
Relying Party (Web Server)
TLS connection
User-Side
FIDO Client (Browser)
Software to provide interface between hardware and FIDO client software
Authenticator Specific Module (ASM)
Interface
User Verification
Attestation Key (PK_M, SK_M), Acert, DS_K
Authentication Keys
User
UAF Authenticator

Fig. 12: **UAF protocol entities.** UAF authenticator is the UAF token provided by manufacturer. The key DS_K, the asymmetric keypair(PK_M, SK_M) with attestation certificate Acert which includes PK_M are provided by the manufacturer. ASM is the software to provide interface between hardware token and client browser.

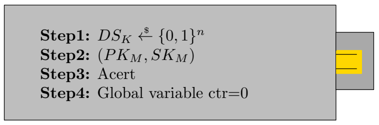

Step1: $DS_K \xleftarrow{\$} \{0,1\}^n$
Step2: (PK_M, SK_M)
Step3: Acert
Step4: Global variable ctr=0

Fig. 13: **UAF Manufacturing Phase.** The key DS_K is a randomly generated n-bit secret. The keypair (PK_M, SK_M) is the asymmetric keypair of the manufacturer shared with the token. Acert is the certificate signed by a trusted CA which includes PK_M. At the time of registration phase, a signature with SK_M is verified with PK_M from Acert by the server to prove the genuineness of the UAF token. The global variable ctr is initialized to zero and incremented after each successful authentication.

includes the authentication modes supported by it and the value R to the server at Step6 which is forwarded at Step7 to the FIDO client. At Step8 the FIDO client forwards the identity of the server as AppId and the value R alongwith the registration policy to the ASM. ASM generates an n-bit random value called the *kid* and computes a value called keyhandle access token (KT) which is the hash of the AppId and *kid*. The ASM stores the pair (*kid*, KT) in its database represented as A_db. At Step9, the ASM forwards the received values of Step8 and the pair (*kid*, KT) to the UAF token. At Step10 the UAF token requests user verification satisfying the UAF registration policy communicated by the FIDO server. On successful user verification, at Step11 the UAF token first generates a website specific public-private key pair represented as (PK_u, SK_u) by applying a random key generation function $\mathcal{K}(\cdot)$. The function $\mathcal{K}(\cdot)$ can be an openSSL key pair generation library which needs as input an elliptic curve such as the NIST standard P-256 curve. We define a key dependent function $f_K(\cdot)$ where K is the secret key. The UAF token computes the function with key DS_K and outputs f_{DS_K} (SK_u, KT, u) which is called the Keyhandle Kh. The above function can be instantiated with any block cipher such as AES in CBC mode or HMAC. It then computes a signature $\mathcal{S} \leftarrow \text{Sign}_{SK_M}(R$, AppId, ChannelId, AAID, *kid*, $PK_u)$ where Authenticator Attestation ID (AAID) is a unique identifier assigned to a model of the FIDO authenticators. At Step12, the token sends the values PK_u, C, AAId, *kid* and $\mathcal{S}$ along with ACert to the ASM. The ASM forwards the received values to the browser at Step13. The browser forwards the received values to the server at Step14. After receiving the values, the server forwards the values to the FIDO server at Step15. At Step16, the FIDO server first verifies the signature $\mathcal{S}$ with PK_M from the certificate Acert. This verification proves the genuineness of the UAF token. On verification, the FIDO server updates its database FIDO_db with the values (AAId, *kid*) corresponding to the username u and it shows the successful registration of the UAF token else the server rejects the registration request. The UAF protocol allows a user to register multiple tokens with the same account.

8.3 UAF Authentication Phase

The authentication procedure is explained in Fig 15. Once second factor authentication with UAF is registered, for example, when the values (AAId, PK_u, kid, Acert) corresponding the username u are registered with the FIDO server, the subsequent login to the website needs to verify the registered values by communicating with the UAF token through browser. The steps are as follows. After receiving username u at Step1, the browser forwards the value to the server at Step2. On receiving the value, the server requests UAF authentication to the FIDO server at Step3. The FIDO server checks its database to retrieve the values AAId and *kid*. At Step4 the FIDO server generates a r bit random challenge R and sends it along with AAId, *kid* and the policy to the server at Step5. The server appends its AppId and forwards the received values to the browser at Step6. The browser forwards the received information with ChannelId to the ASM at Step7. Checking the policy, the ASM selects the authenticator supported by the attached UAF token. It then computes the keyhandle access token KT from the received AppId and *kid* and a value C which is the hash of values (AppId, R, ChannelId) at Step8. At Step9, the ASM sends the values AAId, KT and C to the UAF token. On receiving the values, the UAF token requests the user verification at Step10. It first verifies the user with stored biometric template at Step11. On successful verification, it verifies Kh corresponding the received KT at Step12. Specificlly, it performs inverse computation ${f_{DS_K}}^{-1}(Kh) = (SK'_u, KT', u')$. It then compares the received KT with the computed KT'. If a successful match happens, the token increments a counter represented as 'ctr'. This 'ctr' can be a global or a local variable. If it is local variable then each AppId gets its own 'ctr' else a single 'ctr' is used across all registered AppId. Throughout the explanation we consider 'ctr' as a global integer variable. The value of the 'ctr' is incremented after each successful authentication by the UAF token. This value 'ctr' is introduced to detect cloning of the UAF token as explained in Section 2.3 for U2F. The UAF token signs the values (AAId, C, *ctr*) with SK_u which is represented as $\mathcal{S}$ and sends the values $\mathcal{S}$ and ctr to the ASM at Step13. The ASM then forwards the values to the browser at Step14 and similarly the browser forwards to the server at Step15. At Step16 the server sends the values to the FIDO server. Finally the FIDO server verifies the received signature $\mathcal{S}$ with stored key PK_u for the username u from FIDO_db. On successful verification, it keeps the 'ctr' value with the database and this shows the successful completion of the authentication process.

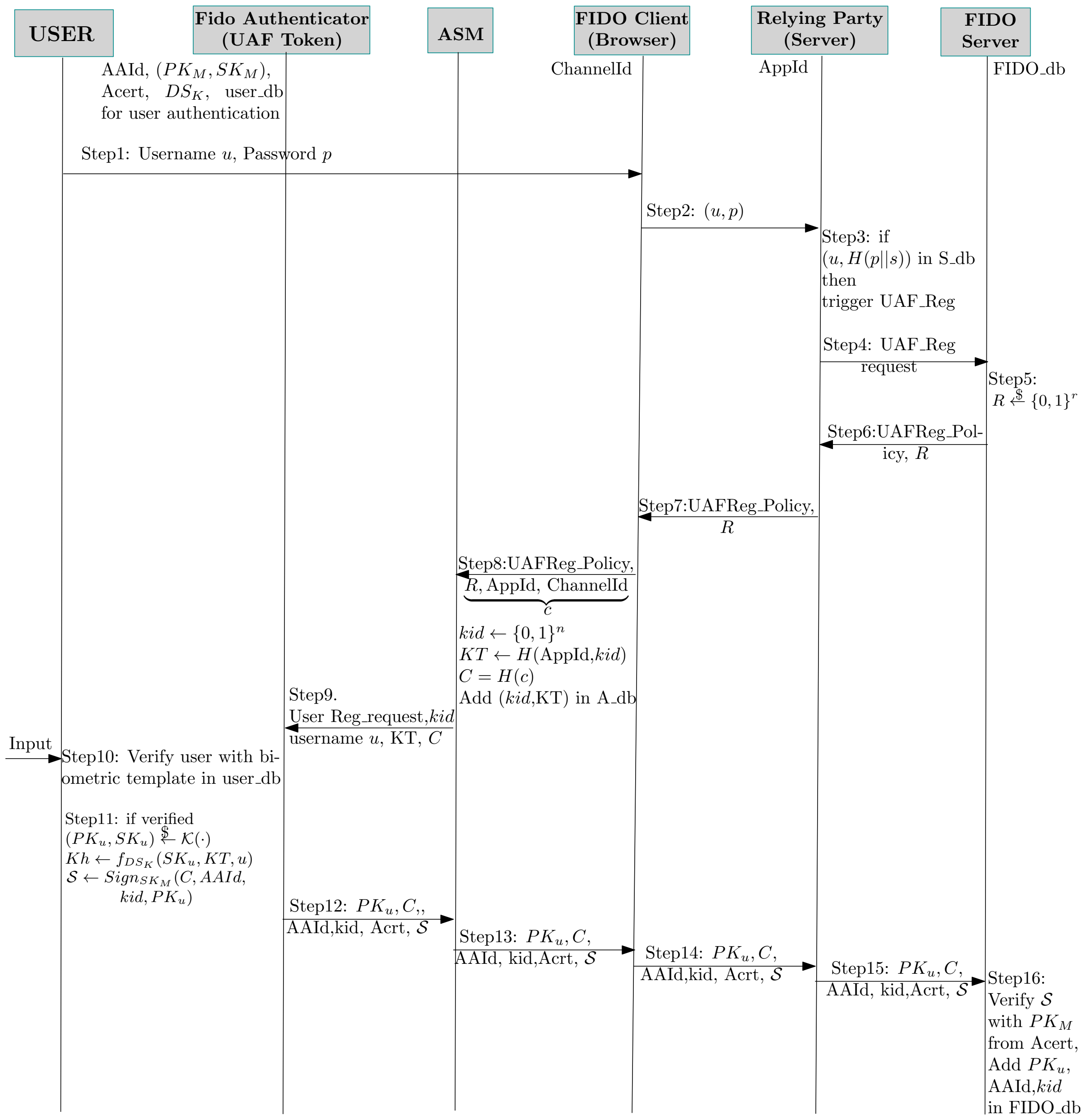

USER
Fido Authenticator (UAF Token)
ASM
FIDO Client (Browser)
Relying Party (Server)
FIDO Server
AAId, (PK_M, SK_M), Acert, DS_K, user_db for user authentication
ChannelId
AppId
FIDO_db
Step1: Username u, Password p
Step2: (u, p)
Step3: if $(u, H(p||s))$ in S_db then trigger UAF_Reg
Step4: UAF_Reg request
Step5: $R \xleftarrow{\$} \{0,1\}^r$
Step6:UAFReg_Policy, R
Step7:UAFReg_Policy, R
Step8:UAFReg_Policy, $\underbrace{R\text{, AppId, ChannelId}}_{c}$
$kid \leftarrow \{0,1\}^n$
$KT \leftarrow H(\text{AppId}, kid)$
$C = H(c)$
Add (kid,KT) in A_db
Step9. User Reg_request,kid username u, KT, C
Input
Step10: Verify user with biometric template in user_db
Step11: if verified
$(PK_u, SK_u) \xleftarrow{\$} \mathcal{K}(\cdot)$
$Kh \leftarrow f_{DS_K}(SK_u, KT, u)$
$\mathcal{S} \leftarrow Sign_{SK_M}(C, AAId, kid, PK_u)$
Step12: $PK_u, C,,$ AAId,kid, Acrt, $\mathcal{S}$
Step13: $PK_u, C,$ AAId, kid,Acrt, $\mathcal{S}$
Step14: $PK_u, C,$ AAId,kid, Acrt, $\mathcal{S}$
Step15: $PK_u, C,$ AAId, kid,Acrt, $\mathcal{S}$
Step16: Verify $\mathcal{S}$ with PK_M from Acert, Add PK_u, AAId,kid in FIDO_db

Fig. 14: **UAF Registration Phase.** The key DS_K, the asymmetric keypair(PK_M, SK_M) with attestation certificate Acert which includes PK_M and the counter 'ctr' are from the manufacturing phase. AppId is the URL of the server. ChannelId is the TLS ChannelId. FIDO_db is the FIDO server database. R is a r bit random number generated by FIDO server. $\mathcal{K}(\cdot)$ is a random key generation function which generates the keypair (PK_u, SK_u). $f_{DS_K}(\cdot)$ with key DS_K is the function to generate Keyhandle Kh, A_db is the database of ASM.

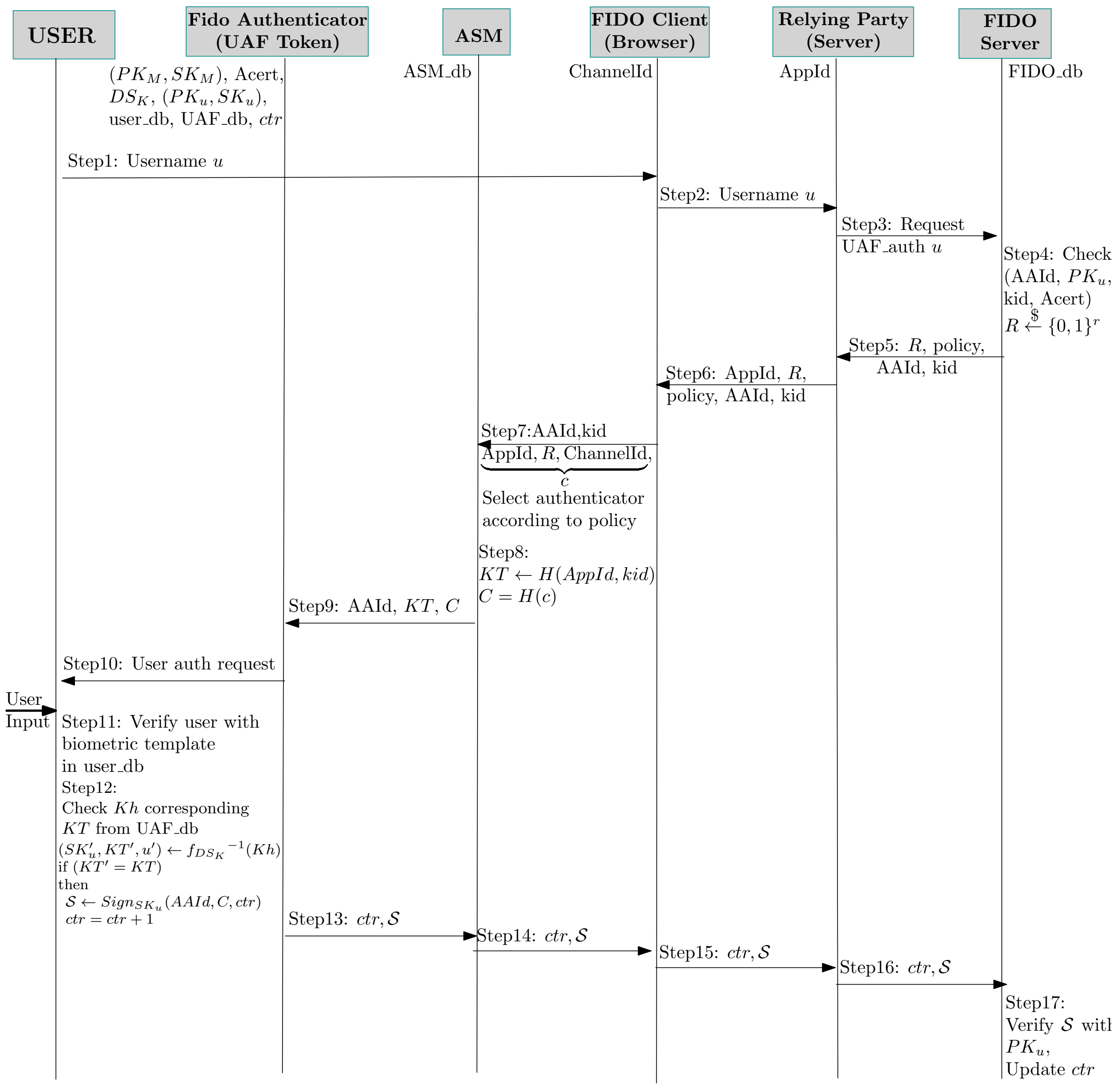

USER
Fido Authenticator (UAF Token)
ASM
FIDO Client (Browser)
Relying Party (Server)
FIDO Server
(PK_M, SK_M), Acert, DS_K, (PK_u, SK_u), user_db, UAF_db, ctr
ASM_db
ChannelId
AppId
FIDO_db
Step1: Username u
Step2: Username u
Step3: Request UAF_auth u
Step4: Check (AAId, PK_u, kid, Acert) $R \overset{\$}{\leftarrow} \{0,1\}^r$
Step5: R, policy, AAId, kid
Step6: AppId, R, policy, AAId, kid
Step7:AAId,kid $\underbrace{\text{AppId}, R, \text{ChannelId},}_{c}$
Select authenticator according to policy
Step8: $KT \leftarrow H(AppId, kid)$ $C = H(c)$
Step9: AAId, KT, C
Step10: User auth request
User Input
Step11: Verify user with biometric template in user_db
Step12: Check Kh corresponding KT from UAF_db
$(SK'_u, KT', u') \leftarrow f_{DS_K}{}^{-1}(Kh)$
if $(KT' = KT)$
then
$\mathcal{S} \leftarrow Sign_{SK_u}(AAId, C, ctr)$
$ctr = ctr + 1$
Step13: $ctr, \mathcal{S}$
Step14: $ctr, \mathcal{S}$
Step15: $ctr, \mathcal{S}$
Step16: $ctr, \mathcal{S}$
Step17: Verify $\mathcal{S}$ witl PK_u, Update ctr

Fig. 15: **UAF Authentication Phase.** The key DS_K, the asymmetric keypair(PK_M, SK_M) with attestation certificate Acert which includes PK_M and counter 'ctr' are provided at the manufacturing phase. The FIDO server database FIDO_db contains all the required values generated at the registration phase corresponding to a registered username. R is a r bit random number generated by the server. AppId is the URL of the server. ChannelId is the TLS ChannelId. The value Kh is the Keyhandle. 'ctr' is the counter used to record the number of authentication requests successfully satisfied by the UAF token.

8.4 Attack on UAF

The UAF authenticator, similar to the U2F authenticator, is assigned a unique device secret key DS_K and an asymmetric keypair (PK_M, SK_M) with attestation certificate Acert at the Manufacturing phase by the manufacturer. The key DS_K is used to compute Keyhandle and can be compromised by applying side channel attacks following the explanation of the Section 4.1. Similarly, the attestation private key SK_M, which is used as the signing key to perform signature during the UAF registration phase, can be compromised as explained in Section 4.2. We recommend the same countermeasure explained in Section 5 to protect the DS_K in case of UAF and to provide unique (PK_M, SK_M) pair to each UAF authenticator to mitigate the attack on SK_M. Providing side channel attack resistant solution is crucial for any hardware that can be easily accessible to an attacker. Therefore, prevention or mitigation of SCA is of utmost importance considering UAF as well.

9 Conclusions

In this work, we observe that a side channel attack is possible on the U2F protocol which may compromise the device secret key DS_k and attestation private key SK_M. Both of these keys are assigned to the U2F token at the manufacturing phase of the U2F protocol. This side channel attack can completely break the second factor U2F solution. We suggest a countermeasure to protect the DS_k and suggest a minor modification in the protocol to mitigate the attack. We also present the detailed analysis of the U2F protocol including its security analysis. The side channel attacks can be applied to any such secure device that uses a single fixed key either for encryption or signature and hence the UAF protocol is also an easy target of this side channel attack. We provide a brief overview of the UAF protocol and show that a strategy similar to the case of U2F can be applied for it to prevent side channel attack on UAF as well.

To develop a signature scheme secure against side channel attacks for the security of U2F protocol is an open problem. Developing countermeasures to overcome the limitations of U2F as explained in Section 7.3 is another interesting problem to explore.

References

1. Jerome H. Saltzer. Protection and the control of information sharing in MULTICS. In Herbert Schorr, Alan J. Perlis, Peter Weiner, and W. Donald Frazer, editors, *Proceedings of the Fourth Symposium on Operating System Principles, SOSP 1973, Thomas J. Watson, Research Center, Yorktown Heights, New York, USA, October 15-17, 1973*. ACM, 1973.
2. Robert Morris and Ken Thompson. Password Security: A Case History, 1979. `http://cs-www.cs.yale.edu/homes/arvind/cs422/doc/unix-sec.pdf`.
3. Password Hashing Competition (PHC), 2014. `https://password-hashing.net/index.html`.
4. Donghoon Chang, Arpan Jati, Sweta Mishra, and Somitra Kumar Sanadhya. Cryptographic module based approach for password hashing schemes. In Stig Fr. Mjølsnes, editor, *Technology and Practice of Passwords - International Conference on Passwords, PASSWORDS'14, Trondheim, Norway, December 8-10, 2014, Revised Selected Papers*, volume 9393 of *Lecture Notes in Computer Science*, pages 39–57. Springer, 2014.
5. TOTP: Time-Based One-Time Password Algorithm. RFC: 6238 . `https://tools.ietf.org/html/rfc6238`, 2011. Online; accessed 24 May 2017.
6. DRAFT NIST Special Publication 800-63B Digital Identity Guidelines. `https://pages.nist.gov/800-63-3/sp800-63b.html`, 2017. Online; accessed 24 May 2017.
7. Florian Maury and Mickael Bergem. A first glance at the U2F protocol. SSTIC2016, 2016. Available at: `https://www.sstic.org/media/SSTIC2016/SSTIC-actes/a_first_glance_at_the_u2f_protocol/SSTIC2016-Article-a_first_glance_at_the_u2f_protocol-maury_bergem.pdf`.
8. FIDO UAF Protocol Specification. FIDO Alliance Implementation Draft, 02 February 2017. Available at: `https://fidoalliance.org/specs/fido-uaf-v1.1-id-20170202/fido-uaf-protocol-v1.1-id-20170202.pdf`.
9. William Stallings. *Cryptography and network security - principles and practice (3. ed.)*. Prentice Hall, 2003.
10. Paul C. Kocher. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In Neal Koblitz, editor, *Advances in Cryptology - CRYPTO '96, 16th Annual International Cryptology Conference, Santa Barbara, California, USA, August 18-22, 1996, Proceedings*, volume 1109 of *Lecture Notes in Computer Science*, pages 104–113. Springer, 1996.

11. Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In Michael J. Wiener, editor, *Advances in Cryptology - CRYPTO '99, 19th Annual International Cryptology Conference, Santa Barbara, California, USA, August 15-19, 1999, Proceedings*, volume 1666 of *Lecture Notes in Computer Science*, pages 388–397. Springer, 1999.
12. Jean-Jacques Quisquater and David Samyde. Electromagnetic analysis (EMA): measures and counter-measures for smart cards. In Isabelle Attali and Thomas P. Jensen, editors, *Smart Card Programming and Security, International Conference on Research in Smart Cards, E-smart 2001, Cannes, France, September 19-21, 2001, Proceedings*, volume 2140 of *Lecture Notes in Computer Science*, pages 200–210. Springer, 2001.
13. Universal 2nd Factor (U2F) Overview . FIDO Alliance Proposed Standard, 14 May 2015. Available at: `https://fidoalliance.org/specs/fido-u2f-overview-ps-20150514.pdf`.
14. Universal 2nd Factor (U2F) Overview . FIDO Alliance Proposed Standard, 15 September 2016. Available at: `https://fidoalliance.org/specs/fido-u2f-v1.1-id-20160915/fido-u2f-overview-v1.1-id-20160915.pdf`.
15. Suresh Chari, Josyula R. Rao, and Pankaj Rohatgi. Template attacks. In Burton S. Kaliski Jr., Çetin Kaya Koç, and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems - CHES 2002, 4th International Workshop, Redwood Shores, CA, USA, August 13-15, 2002, Revised Papers*, volume 2523 of *Lecture Notes in Computer Science*, pages 13–28. Springer, 2002.
16. Dakshi Agrawal, Josyula R. Rao, Pankaj Rohatgi, and Kai Schramm. Templates as master keys. In Josyula R. Rao and Berk Sunar, editors, *Cryptographic Hardware and Embedded Systems - CHES 2005, 7th International Workshop, Edinburgh, UK, August 29 - September 1, 2005, Proceedings*, volume 3659 of *Lecture Notes in Computer Science*, pages 15–29. Springer, 2005.
17. Michel Abdalla and Mihir Bellare. Increasing the lifetime of a key: A comparative analysis of the security of re-keying techniques. In Tatsuaki Okamoto, editor, *Advances in Cryptology - ASIACRYPT 2000, 6th International Conference on the Theory and Application of Cryptology and Information Security, Kyoto, Japan, December 3-7, 2000, Proceedings*, volume 1976 of *Lecture Notes in Computer Science*, pages 546–559. Springer, 2000.
18. Michael Dietz, Alexei Czeskis, Dirk Balfanz, and Dan S. Wallach. Origin-bound certificates: A fresh approach to strong client authentication for the web. In Tadayoshi Kohno, editor, *Proceedings of the 21th USENIX Security Symposium, Bellevue, WA, USA, August 8-10, 2012*, pages 317–331. USENIX Association, 2012.
19. R. Balfanz, D.and Hamilton. Transport Layer Security (TLS) Channel IDs, v01 (IETF Internet-Draf), 2013. Available at: `http://tools.ietf.org/html/draft-balfanz-tlschannelid`.
20. Wikipedia. Denial-of-service attack, wikipedia, the free encyclopedia. `https://en.wikipedia.org/w/index.php?title=Denial-of-service_attack&oldid=792146882`, 2017. Online; accessed 3 August 2017.
21. Nikos Leoutsarakos. What's wrong with FIDO?, 2015. Available at: `http://www.zeropasswords.com/pdfs/WHATisWRONG_FIDO.pdf`.
22. Wei Wei, Jinjiu Li, Longbing Cao, Yuming Ou, and Jiahang Chen. Effective detection of sophisticated online banking fraud on extremely imbalanced data. *World Wide Web*, 16(4), 2013.
23. Symantec Internet Security Threat Report trends for 2010. Technical report, Symantec, April 2011. Available at: `http://eval.symantec.com/mktginfo/enterprise/white_papers/b-whitepaper_internet_security_threat_report_xv_04-2010.en-us.pdf`.
24. Internet Security Threat Report 2013. Technical report, Symantec, April 2013. Available at: `http://www.symantec.com/content/en/us/enterprise/other_resources/b-istr_main_report_v18_2012_21291018.en-us.pdf`.
25. Wikipedia. Non-repudiation — wikipedia, the free encyclopedia. `https://en.wikipedia.org/w/index.php?title=Non-repudiation&oldid=755811340`, 2016. Online; accessed 7 June 2017.
26. Jerry Felix and Chris Hauck. System Security: A Hacker's Perspective, 1987.
27. Wikipedia. Phishing, wikipedia, the free encyclopedia. `https://en.wikipedia.org/w/index.php?title=Phishing&oldid=775126727`, 2017. Online; accessed 24 April 2017.
28. Computer Security Division (Information Technology Lab). Guide to Malware Incident Prevention and Handling. NIST Special Publication 800-83 Revision 1, August, 2015.
29. Mauro Conti, Nicola Dragoni, and Viktor Lesyk. A survey of man in the middle attacks. *IEEE Communications Surveys and Tutorials*, 18(3), 2016.
30. Steven M. Bellovin and Michael Merritt. Encrypted key exchange: password-based protocols secure against dictionary attacks. In *1992 IEEE Computer Society Symposium on Research in Security and Privacy, Oakland, CA, USA, May 4-6, 1992*, pages 72–84. IEEE Computer Society, 1992.
31. Nikolaos Karapanos and Srdjan Capkun. On the effective prevention of TLS man-in-the-middle attacks in web applications. In Kevin Fu and Jaeyeon Jung, editors, *Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014.*, pages 671–686. USENIX Association, 2014.
32. Kamin Whitehouse, Chris Karlof, and David E. Culler. A practical evaluation of radio signal strength for ranging-based localization. *Mobile Computing and Communications Review*, 11, 2007.

33. Wikipedia. Side-channel attack, wikipedia, the free encyclopedia. `https://en.wikipedia.org/w/index.php?title=Side-channel_attack&oldid=770588486`, 2017. Online; accessed 21 April 2017.
34. Tali Garsiel and Paul Irish. How browsers work: Behind the scenes of modern web browsers. `https://www.html5rocks.com/en/tutorials/internals/howbrowserswork/`, 2011. Online; accessed 4 June 2017.
35. Yongbin Zhou and Dengguo Feng. Side-channel attacks: Ten years after its publication and the impacts on cryptographic module security testing. *IACR Cryptology ePrint Archive*, 2005:388, 2005.
36. Junfeng Fan, Xu Guo, Elke De Mulder, Patrick Schaumont, Bart Preneel, and Ingrid Verbauwhede. State-of-the-art of secure ECC implementations: A survey on known side-channel attacks and countermeasures. In Jim Plusquellic and Ken Mai, editors, *HOST 2010, Proceedings of the 2010 IEEE International Symposium on Hardware-Oriented Security and Trust (HOST), 13-14 June 2010, Anaheim Convention Center, California, USA*, pages 76–87. IEEE Computer Society, 2010.
37. P. Kocher. Leak Resistant Cryptographic Indexed Key Update, US Patent 6539092. `http://www.google.co.in/patents/US6539092`, 2003.
38. Wikipedia. Pseudorandom number generator, wikipedia, the free encyclopedia. `https://en.wikipedia.org/w/index.php?title=Pseudorandom_number_generator&oldid=781388201`, 2017. Online; accessed 4 June 2017.
39. Wikipedia. Advanced encryption standard — wikipedia, the free encyclopedia. `https://en.wikipedia.org/w/index.php?title=Advanced_Encryption_Standard&oldid=783641380`, 2017. Online; accessed 8 June 2017.