Christian Paul Kollmann, BSc

Integrating Universal Second Factor Authentication into CrySIL

An Approach for Extensible and Flexible Second Factor Authentication

MASTER'S THESIS

to achieve the university degree of

Diplom-Ingenieur

Master's degree programme: Computer Science

submitted to

Graz University of Technology

Supervisor

O.Univ.-Prof. Dipl-Ing. Dr.techn. Reinhard Posch

Institute of Applied Information Processing and Communications

Advisor

Dipl.-Ing. Florian Reimair

Graz, May 2015

Affidavit

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources. The text document uploaded to TUGRAZonline is identical to the present master's thesis.

SIGNATURINFORMATIONEN	**Unterzeichner**	Christian Paul Kollmann
	Datum/Zeit-UTC	2016-05-07T14:40:19+02:00
	Prüfinformation	Informationen zur Prüfung der elektronischen Signatur finden Sie unter: https://www.signaturpruefung.gv.at
Hinweis	This document is signed with a qualified electronic signature. According to § 4 art. 1 of the Signature Act it in principle is legally equivalent to a handwritten signature.	

Graz, _____ _____

Date Signature

Abstract

The simple combination of username and password is frequently used to authenticate users to services. However, using the secret password alone does not meet high security standards for authentication, according to NIST. Two-factor authentication (2FA) is an accepted way to increase the security of user verification. 2FA adds a second form of authentication to the process, typically proof of possession of a physical object or a personal characteristic of the user, e.g. a fingerprint. Despite offering benefits, today's methods for 2FA also exhibit deficiencies: Most methods lack adequate user experience and protection against man-in-the-middle attacks. The FIDO Alliance created an approach called Universal Second Factor (U2F) to address some of the shortcomings of existing methods for 2FA. U2F is an open standard, relying on special USB tokens connected to the user's device completing a challenge–response protocol. However, this idea of USB tokens does not extend to mobile scenarios, e.g. when using smartphones. Furthermore, the user cannot easily use existing cryptographic devices with U2F implementations.

Our approach provides an extensible and flexible method for 2FA. We build upon the broad adoption of U2F and integrate it into the CrySIL system. CrySIL is a solution for remote key storage providing cryptographic operations to heterogeneous platforms. We adapt the U2F implementation in the Google Chrome and Chromium browsers and develop a credential provider for Microsoft Windows. Both solutions enable forwarding authentication challenges to a CrySIL instance. The modular architecture of CrySIL system makes it possible to implement modules to support various authenticator devices, including genuine U2F tokens using NFC transport, existing smart cards, and similar systems such as electronic identity cards. We also succeed in fully virtualizing the U2F token by employing a module relying on a pure software implementation. Even more valuable, we can use any Android device running a CrySIL instance as a U2F token. We enable the user to utilize existing devices as a strong second factor, retaining the benefits and security properties of the U2F standard. Websites, which already support U2F, accept our system as a form of 2FA. Our approach provides solutions for challenges present in common approaches and offers many advantages compared to existing solutions.

Kurzfassung

Die Kombination aus Name und Passwort ist weit verbreitet um Benutzer gegenüber einem Dienst zu authentifizieren. Ein geheimes Passwort alleine entspricht laut NIST jedoch nicht hohen Sicherheitsanforderungen. Zwei-Faktor-Authentifizierung (2FA) ist eine bewährte Methode um die Sicherheit zu erhöhen. Dazu wird während der Authentifizierung ein zweiter Faktor abgefragt, etwa der Besitz eines Objektes oder eine physische Eigenschaft des Benutzers, bspw. ein Fingerabdruck. Aktuelle Methoden für 2FA weisen einige Nachteile auf, bspw. geringen Schutz gegen Man-in-the-Middle-Angriffe oder mangelhafte Benutzerfreundlichkeit. Die FIDO Alliance hat Universal Second Factor (U2F) entwickelt, um Defizite bestehender Methoden für 2FA zu eliminieren. U2F ist ein offener Standard der spezielle USB-Token nutzt, welche ein kryptographisches Protokoll zur Authentifizierung absolvieren. Diese Token können jedoch nicht in mobilen Umgebungen eingesetzt werden; ebenso wenig können bestehende kryptographische Geräte auf einfache Weise im U2F-Ablauf verwendet werden.

Unser Ansatz bietet eine erweiterbare und flexible Methode für 2FA. Dazu integrieren wir eine Implementierung des U2F-Standards in das CrySIL-System. CrySIL ist eine Lösung für eine zentrale Schlüssel-Verwaltung welche kryptographische Operationen für heterogene Umgebungen anbietet. Wir modifizieren die U2F-Implementierung in Google Chrome (bzw. Chromium) und entwickeln ein Plugin zur Anmeldung an ein Microsoft-Windows-System. Beide Lösungen erlauben es uns, die Authentifizierung an eine CrySIL-Instanz weiter zu leiten. Das modulare CrySIL-System ermöglicht es uns, verschiedene Module zu entwickeln, um eine Vielzahl von kryptographischen Geräten zu unterstützen: U2F-Token mit einer NFC-Schnittstelle, bestehende Smartcards, oder auch österreichische Bürgerkarten. Zusätzlich kann jedes Android-Gerät mit einer CrySIL-Instanz als U2F-Token verwendet werden. Mit unserer Lösung kann der Benutzer bestehende Geräte als sicheren zweiten Faktor zur Authentifizierung verwenden, während alle Sicherheitseigenschaften des U2F-Standards beibehalten werden. Webseiten die U2F bereits unterstützen akzeptieren unser System. Unser Ansatz bietet Lösungen für einige Herausforderungen für 2FA-Systeme und weist Vorteile im direkten Vergleich mit bekannten Methoden auf.

Contents

List of Figures

List of Listings

Acknowledgments

I am grateful to Florian Reimair for his marvelous supervision. I also need to thank Florian and Bernd Prünster for our fruitful discussions.

I thank Christoph, Michael, Patrick, Richard, and Stefan for reading through parts of this thesis and providing me with excellent feedback.

My deepest thanks, however, go to Verena for her invaluable support.

1 Introduction

Users regularly use the combination of a username and password to authenticate to a variety of services, e.g. operating systems or web applications. The username is usually publicly known, but users should keep their passwords secret. However, most users are not up to the challenge of creating and remembering secure passwords for a myriad of services. Taneski, Hericko, and Brumen [43] describe that passwords are often shared among different services and are either weak and therefore easily cracked by a dictionary attack or easy to figure out by an attacker using social engineering. Mirante and Cappos [29] show that stolen databases are containing plenty of passwords, often insufficiently hashed, representing a viable source for attackers to break into user accounts. NIST specifies four security levels of authentication, showing that using passwords as the single factor does not meet high-security standards [10]. Hardware-based cryptographic tokens must be used to achieve compliance with the highest level. This NIST recommendation is widely adopted by government agencies in the US and companies worldwide.

Two-factor authentication (2FA) is a common way to add security to the process of user verification. It combines the secret password known to the user (first factor) with some other form of authentication (second factor). O'Gorman [34] describes and compares popular choices for the second factor: Tokens to prove possession of a physical object and biometrics as a physical characteristic of the user. An every-day example of 2FA is the use of a bank card (proof of possession) together with the PIN (proof of knowledge) to withdraw money from a cash dispenser. This example and similar methods do not scale well to the ubiquitous use of authentication to web applications in the browser nowadays. Web service providers may use another form of 2FA such as one-time passwords (OTP). These short codes (usually around 6 to 8 characters) may either be sent to the mobile phone

of the user or generated by a special device such as an RSA SecurID token[1]. Austria's mobile phone signature[2], one implementation of the Austrian citizen card concept defined by Leitold, Hollosi, and Posch [25], uses this method. The web service grants access to the private signature key when the user provides his password and the OTP sent to the cell phone of the user.

Common methods for 2FA increase the security in the authentication process but exhibit some limitations. OTP requires the user to copy manually codes from the device generating the codes to the device performing the login. When using mobile phones, there may be coverage issues and delays in receiving the codes. Special hardware tokens often work for one particular site only. Methods based on a challenge–response protocol, e.g. cryptographic smart cards, require special reader hardware on the client side. This requirement also holds for electronic identity (eID) systems based on smart cards, sometimes called citizen cards. Additionally, these eID cards by definition identify the user and are controlled by the government. Users may be unwilling to use them on a website where a pseudonym otherwise identifies them. All these aspects place an unnecessary burden on the user and impede user acceptance.

Research shows some drawbacks of common 2FA solutions: Petsas et al. [35] illustrate that, despite the clear value added to account security when using a second factor, no more than 6.4 % of Google users activate 2FA. Weir et al. [45] state that users of e-banking solutions perceive one-factor methods as secure and convenient options and demonstrate a reluctance to use two-factor methods for authentication. Herley and van Oorschot [16] even claim that passwords are here to stay since they are the best fit for many scenarios and suggest supporting passwords better instead of trying to replace them. Even though using 2FA increases the security of login procedures, attacks are still possible. Schneier [39] explains that established 2FA methods, in general, are not immune to man-in-the-middle and phishing attacks. To put it in a nutshell, today's methods for 2FA lack adequate user experience and resilience to sophisticated attacks and thus, are not widespread among average users.

[1] https://www.rsa.com/en-us/products-services/identity-access-management/securid/hardware-tokens

[2] https://www.handy-signatur.at/hs2/

Srinivas et al. [40] define the Universal Second Factor (U2F) as an open authentication standard, targeting some of the shortcomings of existing 2FA solutions. The FIDO Alliance, an industry consortium formed to develop and standardize online authentication methods, backs development of the specification. The standard enables online services to add strong 2FA to existing user login procedures. Furthermore, it aims at unifying 2FA by using an open protocol and simple hardware tokens. Design goals of the standard include ease of use and cost efficiency, trying to make second-factor authentication viable for the average user on the Web. One authenticator device can be utilized for an infinite number of services since the token creates new credentials for every service.

The emerging U2F standard currently gains industry support and can be used as a second factor for authentication on several websites, including those provided by Google, Dropbox, and Github. Native support on the client side is being implemented in browsers, starting with Google Chrome, or rather its open source variant Chromium. Yubico, among other companies, produces compatible hardware in the form of a FIDO U2F Security Key[3], allowing for driver-less USB operation. To sum it up, a broad number of users can use implementations of the standard throughout the Web as of today.

1.1 Challenge

Existing U2F tokens embed a secure element in a small form factor as simple USB key fobs. Registration of new tokens and authentication of users is handled by a challenge–response protocol between relying party and the hardware token. The client software running on the user's computer mediates between those two. One goal of U2F is to strengthen user authentication on the Web; usually a browser plays the role of the client software. The client takes measures to prevent man-in-the-middle attacks on the authentication process. The user experience is simplified, requiring the user to insert merely the USB token and press the only button on it to prove possession of the device and confirm the operation. This procedure works quite well on desktop devices and is reasonably fast when the user has his token at hand.

[3] https://www.yubico.com/applications/fido/

However, this idea does not extend to scenarios involving mobile devices such as smartphones and tablets. Nearly all of them apparently lack a standard USB port and thus the user cannot easily connect the token to the smartphone. Although some U2F tokens offer an NFC interface, no mobile browser supports this method of operation as of February 2016. This limitation requires the website to fall back on some other method for authentication. That method has to be configured by the user in advance and typically relies on a different second factor. Obviously, this additional configuration increases the complexity and reduces ease-of-use for the user.

Moreover, only certified hardware tokens can be used for U2F, further diminishing the potential user base. Existing smart cards or similar cryptographic devices such as eID cards cannot be easily used in the process. Therefore, the user has to carry around a particular U2F token all the time to be able to use it in the authentication process. Overall, the U2F process clearly lacks flexibility on the client side with the usage of hardware tokens as they exist today.

All things considered, an ideal solution for 2FA has to stand many challenges. First, the hassle of competing standards and methods should be avoided, for both, users and service providers. Second, the user needs to have the option to use his authenticator device no matter on what service the account is registered. Therefore, service providers need to be able to implement the solution easily. Third, the user should be able to utilize existing cryptographic devices and not need to buy and carry an additional device. Nevertheless, the security of the authenticator device has to rely on proven cryptographic standards, ideally backed by a secure hardware element. Fourth, the setting of the login procedure should not affect the user experience, i.e. it should not matter if the user logs in to his account using a desktop browser or a mobile device. The user should be able to use the same token in both scenarios. Finally, the whole process of confirming possession of the second factor should be as effortless as possible. A solution meeting all these requirements would not only increase security in existing authentication processes but would most probably also be widely accepted by users.

1.2 Contribution

Our contribution is an approach as a step towards an ideal 2FA solution as described before. Our proposal builds upon the existing implementations of U2F clients and the support from service providers to reach a large number of users. We take a step forward by increasing the flexibility of the standard in two aspects: We can use different, existing authenticator devices and show that U2F is not only suitable for the Web, as demonstrated by the early implementations, but also for local systems, i.e. the Microsoft Windows login.

The foundation for our work is the integration of the U2F standard into the well-tried cryptographic service interoperability layer (CrySIL). Reimair, Teufl, and Zefferer [37] describe CrySIL as an approach standing the challenges of key management and utilizing cryptographic functions within heterogeneous application deployment scenarios. It offers a centralized, secure key storage and cryptographic engine to empower access to the keys everywhere at any time. Clients can execute cryptographic operations on the central instance utilizing a defined protocol. The modular architecture of CrySIL supports the combination of various front-ends receiving operation requests and back-ends executing cryptographic operations. These back-ends usually use hardware devices to conduct the operations.

We show that our approach enables various applications for secure second-factor authentication. We adapt the U2F client implementation in Google Chrome, and its open source variant Chromium, to forward the authentication challenges to a CrySIL instance. We also implement a custom credential provider for Microsoft Windows 10 to add a second factor to the login procedure on a local system. The architecture of CrySIL permits using a variety of devices and appropriate modules to act as the authenticator device in the U2F process. We can fully virtualize the U2F token as the cryptographic requests are not necessarily handled by an actual U2F hardware device. The CrySIL instance can even be provided by a cloud service, supporting universal access from any client. The client implementation in the Google Chrome browser makes it possible to apply our approach in real-world scenarios on several websites.

Additionally, we enable the user to use her existing mobile device as the second factor in an authentication process. To offer this possibility, we use a MoCrySIL instance on an Android device. Reimair et al. [38] describe MoCrySIL as a solution to carry your cryptographic keys in your pocket by running a CrySIL instance on a mobile device. We rely on a technique called WebVPN to connect the U2F client to the MoCrySIL instance securely. Reimair et al. demonstrate this WebVPN scheme to solve the challenge of reaching the mobile device despite ever-changing IP addresses and NAT behind routers. Within the scheme, a relay service mediates between a CrySIL client and the MoCrySIL instance. The service utilizes WebSocket connections and push notifications and provides end-to-end security between the two communication partners. When using his mobile device for authentication, the user can choose to handle the cryptographic operations with the hardware-backed key store provided by the Android system[4] or with an external token. That token can be connected over NFC and be either a compatible smart card or a genuine U2F token. In the latter case, the relying party in U2F can not tell the difference between the user connecting the token directly and utilizing our solution.

In addition to the Web use case described before, we facilitate a solution for 2FA on a local Microsoft Windows system. We empower the user to use smart cards and similar systems such as eID cards to provide the cryptographic services. To support the usage of these devices, we implement a suitable module for CrySIL. This module enables any CrySIL instance to execute commands on any compatible smart card. Integrated into our approach, the smart card will take the part of the hardware token in the U2F process. This concept empowers the user to use existing cryptographic devices for secure 2FA. The interoperability features of CrySIL make it possible to use these authenticator devices for all U2F clients, including the browser scenario described earlier.

The advantages of our solution compared to a traditional U2F approach are as follows: We enable the use of existing cryptographic devices, including smart cards and similar eID cards. The user can use these devices as the secure second factor in applications supporting U2F, including the Chrome browser. Our solution also makes it possible to use an Android smartphone as the authenticator device. The system can be extended to support even more devices, given the extensibility of

[4] https://developer.android.com/training/articles/keystore.html

the CrySIL approach. Additionally, we show how to utilize U2F in local scenarios such as the Windows login.

Only a few limitations arise when using our approach: When using a smart card, the same key is used for all authentication challenges. Reusing the key contradicts the concept of one key per relying party of U2F, but does not influence the strong cryptographic properties. It, however, compromises the privacy of the user if he uses the same device for different accounts or across cooperating relying parties. Relying parties could then connect the accounts because they share one public key. When using the Android key store, a self-signed certificate is used as the attestation certificate for registration of a new token in U2F. This certificate prevents the relying party from identifying the manufacturer of the token, which contradicts a convention of U2F.

1.3 Structure

In Chapter 2 we describe the fundamentals of our work, including the main building blocks U2F and CrySIL. In Chapter 3 we survey the literature to find existing solutions for an ideal 2FA method. In Chapter 4 we describe our approach in detail and explain supported use cases in real-world applications. In Chapter 5 we outline the software components we have implemented, including modules for the CrySIL system. In Chapter 6 we perform a security analysis of our system and compare our approach to existing solutions. In Chapter 7 we conclude with thoughts on open questions and further research opportunities.

2 Preliminaries

We base our approach for 2FA on two well-tried systems: U2F provides the authentication standard and some existing client implementations; CrySIL presents an open and flexible architecture where its implementation offers a centralized key store and an API for cryptographic services. In this chapter, we introduce the U2F standard in general, including details about the protocol for registration of tokens and authentication of accounts. Additionally, we show how U2F takes measures against man-in-the-middle attacks. We also explain concepts central to the U2F standard, such as attestation certificates and generation of credentials. We illustrate the architecture of the CrySIL system and explain how that system can be extended with custom modules. We will combine both systems so that a CrySIL instance can act as the authenticator device in a U2F setting. Moreover, we provide background information on the login system that Microsoft uses for its Windows operating systems.

In Section 2.1 we discuss the U2F standard, including the core idea, servers, clients, and the protocol in detail. In Section 2.2 we describe the CrySIL system, which provides the software framework for our approach. In Section 2.3 we examine the Microsoft Windows login system including the extensible credential providers.

2.1 Universal Second Factor Authentication

U2F is an open standard maintained by the FIDO Alliance [40]. Several industry companies are members of the FIDO Alliance, including Yubico, Google, and NXP. The goal of U2F is to provide an open and flexible method for secure second-factor authentication with the focus put on Web use cases. It is designed to augment existing authentication procedures which rely on usernames and passwords

only. The standard specifies the use of authenticator devices (also simply called tokens) completing a cryptographic challenge–response protocol to prove possession of the device. That device may be hardware-backed or may be realized in software, where the first option is preferable. Various websites have adopted the specification already and offer U2F as a method for 2FA, including Google, Dropbox, and Github. Nevertheless, native client support is implemented in the Google Chrome browser (and its open-source variant Chromium) only as of February 2016. Yubico and other companies produce special USB tokens compatible with the U2F standard. These authenticator devices are relatively cheap compared to other security tokens but support the U2F protocol only.

The standard addresses some of the shortcomings of common second-factor authentication methods. The authentication process for the user is greatly simplified: In addition to the traditional combination of username and password, the user only has to plug her USB token into the computer and confirm the operation by pressing the only button on the token. In general, the relying party (e.g. a website) communicates via the client (e.g. the browser) with the authenticator (e.g. a hardware token) to complete a challenge–response protocol. A single token supports an unlimited number of websites and relying parties. Nevertheless, the token generates new credentials for each relying party. To deal with a large number of credentials, manufacturers may decide to provide enough tamper-resistant storage or to export private keys in a secure manner.

Advantages of U2F include the use of standard cryptography, namely the use of private–public key pairs on an elliptic curve and matching digital signatures. The standard is designed to provide the user with freedom of choice for authenticator devices. Users can use the same token for several relying parties instead of one device per service. Several manufacturers produce low-cost devices for U2F, significantly lowering the financial barrier for users. Clients can implement additional checks in the protocol to mitigate common man-in-the-middle attacks. Lang et al. [24] describe some design goals of U2F, including ease of use for end-users and developers as well as increased privacy, since the credentials used for authentication are specific to one relying party and do not leak information about the user.

Nevertheless, some shortcomings exist in the usual setting of U2F. The user still needs to carry an additional, though small, device to use it in the authentica-

tion process. This requirement introduces additional hassle for the user, and such a token can be easily lost. The problem of a lost token is in part mitigated by the general recommendation to use two hardware keys and some other fallback solution implemented in software by the relying party. Another drawback is that existing cryptographic devices, e.g. smart cards, cannot be easily used with implementations of the standard. This limitation prevents the user from using devices he already owns, such as electronic identity cards or smartphones with cryptographic functionality and forces him to carry another device. Moreover, existing implementations are not suitable for the mobile use case, as no mobile browser supports U2F as of February 2016.

2.1.1 Protocol Messages

The basic operation mode of U2F for registration and authentication is a challenge–response protocol. In U2F terms, the relying party, e.g. a website, sends a random challenge to the client, e.g. the browser, which forwards it to the authenticator device (or token), e.g. a Yubico security key. The client verifies the validity of the challenge by checking the application identifier provided by the relying party. This verification ensures that no third party has injected a malicious authentication request. The client forwards the request to the hardware device. The device performs the cryptographic operations and sends the result back to the client. The client, in turn, sends the response to the relying party, which verifies the calculations. The main use case of U2F is authentication on the Web though other software can implement the protocol and use it for secure second-factor authentication. In the Web scenario, only websites accessed over TLS are allowed to send U2F requests to the browser.

All cryptographic operations involved in U2F are based on public key cryptographic. More explicit elliptic curve cryptography is used, based on the `secp256r1` curve defined in [1], also known as P-256 defined in [5]. That means that any private key used for signature generation is simply a multiplier, used together with the generator of the curve to calculate the public key. The public key is then a point on the elliptic curve. The public key representation used in the U2F protocol consists of two 32 byte values of the `x` and `y` coordinate of the point on the elliptic curve.

For calculating the signature over the challenges from the relying party, ECDSA is used, established in [21]. The signature itself is represented as an ASN.1 structure, defined in [18], containing two 32-byte values. All message digests used throughout the protocol are calculated using SHA256, described in [12], yielding a 32-byte long digest. The attestation certificate sent in the registration response is encoded using DER, defined in [19].

The two following subsections describe the details of the U2F protocol for registration and authentication, laying out the format of all messages between clients and relying parties. Balfanz, Birgisson, and Lang [2] define the JavaScript API used for the U2F protocol, whereas Balfanz and Ehrensvard [3] designate the format of the raw messages used in the protocol. We use both documents as the source for the description. Since U2F was primarily designed for Web applications, the URL-safe variant of Base64 encoding is used [22]. In this variant, all "-" are replaced with "+", all "/" are replaced with "_", and the padding character "=" and all newline characters are stripped from the encoded text. In all steps described in the following, the version of the U2F protocol is included in the messages and has the fixed value `U2F_V2`. We are not considering older versions of the protocol in our approach and this thesis.

Registration

When the user activates second-factor authentication for his account, the relying party starts the registration workflow of U2F. Therein, the authenticator token creates new credentials—a private–public cryptographic key pair—bound to the relying party. The token exports the public key and a handle identifying the new key pair in subsequent authentication requests.

Figure 2.1 shows the message flow for registration in a U2F setting. Details for this message flow are as follows:

1. The relying party (or server, S) generates a random challenge (r) and includes its application identifier (a) in the request to the client (C). On the Web, this application identifier is simply the base URL of the website.

$$S \rightarrow C : (a, r)$$

2. The client verifies the authenticity of the application identifier. The client builds a data structure (d) which may include the TLS channel identification for MITM protection.

$$C : \text{generate } d$$

3. The client includes that data structure in the request and forwards it together with the challenge from the relying party to the token (T).

$$C \to T : (a, r, d)$$

4. The token always generates a fresh private–public key pair (k, P) to be used for this application identifier only. The token calculates a signature (with the private key matching the public key of the attestation certificate) over the combination of application identifier, random challenge, data structure, public key (P), and key handle (h).

$$T : \text{generate } k, P, h, \quad s = \text{sign}(a, r, d, P, h)$$

5. The token sends back that signature together with a key handle identifying the new private–public key pair, the public key of the pair and an attestation certificate (t) to the client.

$$T \to C : (P, h, t, s)$$

6. The client sends back the response from the token to the relying party, together with the challenge and the client data structure.

$$C \to S : (r, d, P, h, t, s)$$

7. The relying party checks the signature and verifies the other data, including the attestation certificate.

$$S : \text{verify } s, t$$

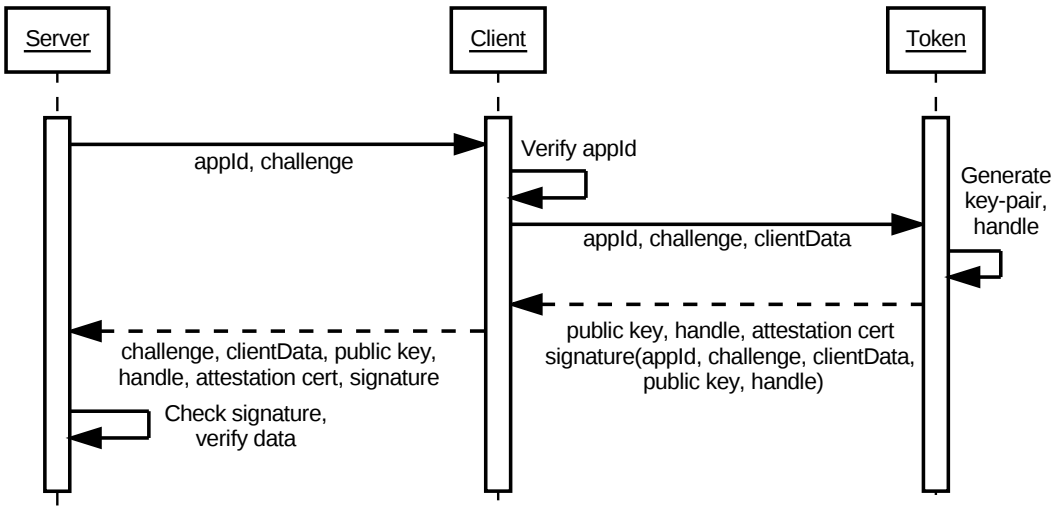


Figure 2.1: Sequence diagram showing a registration operation in the U2F process. Server, client, and token complete a challenge–response protocol. Names refer to data fields in the definition of the standard.

Listing 2.1 describes the format of the registration challenge sent from the relying party to the client. Listing 2.2 illustrates the conversion of that first message on the client. Listing 2.3 presents the answer from the token to the client and relying party. The structure `clientData` used in the messages are the bytes of the UTF8-encoded JSON representation of the structure stated. The field `cid_pubkey` therein is optional and may include the TLS channel identification for MITM protection. The magic number `0x05` in the registration data field in the answer is the value of the reserved byte, used for legacy reasons. The magic number `0x00` in the signature input in the answer is a byte reserved for future use. The public key has a length of exactly 65 bytes and is an uncompressed representation of a curve point on the P-256 NIST elliptic curve. The key handle has variable length but is limited to 255 bytes, since the length of the key handle has to be stated in one byte preceding the handle. The attestation certificate also has variable length, but the length is determined by the DER encoding used. All hashes are calculated using SHA256, yielding a 32-byte long digest.

```
{ appId: "https://example.com",
  challenge: Base64(32 byte random),
  version: "U2F_V2"
}
```

Listing 2.1: Registration challenge sent from the relying party to the client. Message is shown in the JSON format defined by the standard.

```
{ appIdHash: Base64(SHA256(<appId>)),
  challengeHash: Base64(SHA256(clientData)),
  version: "U2F_V2"
}
clientData = {
  origin: <appId>,
  challenge: <challenge>,
  typ: "navigator.id.finishEnrollment",
  cid_pubkey: {
    kty: "EC", crv: "P-256",
    x: Base64(coordinate), y: Base64(coordinate)
  } /* optional */
}
```

Listing 2.2: Converted registration challenge on the client, suitable for the token. Message is shown in the JSON format defined by the standard. Names in angle brackets refer to contents from the previous message.

```
{ clientData: <clientData>,
  registrationData: Base64(0x05, <publicKey>, <keyHandleLength>,
    <keyHandle>, <attestationCertificate>, <signature>)
}
/* signature over (0x00, <appIdHash>, <challengeHash>, <keyHandle>,
   <publicKey>) */
```

Listing 2.3: Registration response from the token. Message is shown in the JSON format defined by the standard. Names in angle brackets refer to contents from previous messages or data generated by the token.

Authentication

When the user has activated second-factor authentication for his account and wants to log in, the relying party initiates the authentication workflow of U2F. There, the relying party sends the key handle from the registration process to the token. The token then either looks up the key pair or creates the private key directly from the handle, as we shall discuss in Section 2.1.5.

Figure 2.2 shows the message flow for authentication in a U2F setting. Details for this message flow are as follows:

1. The relying party (or server, S) generates a random challenge (r) and includes its application identifier (a) and the key handle (h) of the token registered in the request to the client (C).

$$S \to C : (a, r, h)$$

2. The client verifies the authenticity of the application identifier and creates a data structure (d) equivalent to the one in the registration protocol flow.

$$C : \text{generate } d$$

3. The client adds that structure to the challenge and sends it to the token (T).

$$C \to T : (a, r, d, h)$$

4. The token uses the handle to look up or calculate the appropriate private key (k). It uses that key to compute a signature over the application identifier, random challenge, data structure, and a counter value (c).

$$T : \text{look up } k, P, \quad s = \text{sign}(a, r, d, c)$$

5. The token sends back that signature together with the counter value in plain to the client.

$$T \to C : (c, s)$$

6. The client sends back the data received from the token to the relying party, together with the challenge and data structure.

$$C \to S : (r, d, c, s)$$

7. The relying party verifies the signature and checks the other data, including the counter value.

S : verify s, c

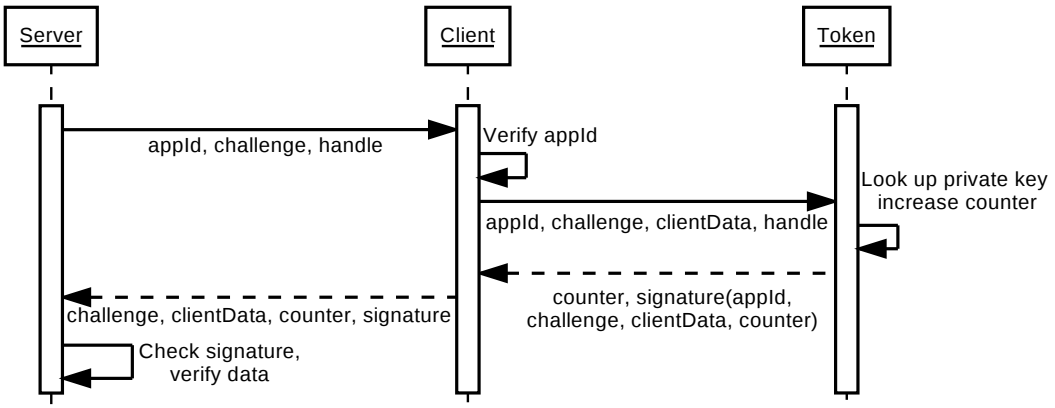


Figure 2.2: Sequence diagram showing authentication in a U2F process. Server, client, and token complete a challenge–response protocol. Names refer to the data fields in the definition of the standard.

Listing 2.4 describes the format of the authentication challenge from the relying party to the client. Listing 2.5 illustrates the conversion of that first message on the client. Listing 2.6 presents the answer from the token to the client and relying party. The counter value kept internally by the token is exactly represented in four bytes using big endian encoding (most-significant byte values first). The structure `clientData` used in the messages are the bytes of the UTF8-encoded JSON representation of the structure stated. The field `cid_pubkey` therein is optional and may include the TLS channel identification for MITM protection. The magic number `0x01` in the signature input is the value of the user-presence byte, indicating the user has actively approved this request. The U2F standard mandates a form of user consent.

```
{ appId: "https://example.com",
  challenge: Base64(32 byte random),
  keyHandle: Base64(key handle),
  version: "U2F_V2"
}
```

Listing 2.4: Authentication challenge sent from the relying party to the client. Message is shown in the JSON format defined by the standard.

```
{ appIdHash: Base64(SHA256(<appId>)),
  challengeHash: Base64(SHA256(clientData)),
  keyHandle: <keyHandle>,
  version: "U2F_V2"
}
clientData = {
  origin: <appId>,
  challenge: <challenge>,
  typ: "navigator.id.getAssertion",
  cid_pubkey: {
    kty: "EC", crv: "P-256",
    x: Base64(coordinate), y: Base64(coordinate)
  } /* optional */
}
```

Listing 2.5: Converted authentication challenge on the client, suitable for the token. Message is shown in the JSON format defined by the standard. Names in angle brackets refer to contents from the previous message.

```
{ challenge: <challenge>,
  keyHandle: <keyHandle>,
  clientData: <clientData>,
  signatureData: Base64(0x01, <counter>, <signature>)
}
signature over (<appIdHash>, 0x01, <counter>, <challengeHash>)
```

Listing 2.6: Authentication response from the token. Message is shown in the JSON format defined by the standard. Names in angle brackets refer to contents from previous messages or data generated by the token.

2.1.2 Man-in-the-Middle Protection

Schneier [39] states that common 2FA methods offer no protection against man-in-the-middle (MITM) and phishing attacks. The attacker can always set up a fake website and trick the user into entering his credentials including the second factor. Those credentials are obviously valid for a session initiated by the attacker to the real website. U2F actively addresses this threat of MITM attacks.

Upon registration, a U2F token is required to generate a new private–public key pair unique to the origin (application identifier) of the relying party. Consequently, the key handle exported is tangled with the origin internally. If the attacker tries to send an authentication request to a U2F device using a false origin, e.g. while performing a phishing attack, the device will detect the mismatch between key handle and origin and will abort the authentication. Additionally, the authenticity of the origin stated by the relying party has to be checked by the client.

Balfanz and Hamilton [4] define the channel ID extension to TLS. This standard establishes a long-lived cryptographic channel between client and server that persists across multiple connections and sessions. The U2F specification recommends that the client use this extension to detect more sophisticated attacks. If the server uses this TLS extension, the browser will insert the public key of the current connection into the client data object sent to the token. The value of this public key, along with the origin and key handle, is signed by the token and sent back to the relying party. The server can then verify that the client data signed by the token contains the actual origin and correct channel ID of the TLS connection. Since an attacker performing a man-in-the-middle attack needs to establish two separate TLS connections, one to the user and one to the server, a mismatch in the channel ID will be detected by the origin site.

This protection against MITM attacks will not work if either the browser does not support the channel ID extension or if the attacker presents a valid TLS certificate for the attacked site. Another example of MITM attacks where the protections by the protocol will not help is the following: The user first registers a new account without a second factor on a website. Subsequently, when the user adds a U2F token to her account, the attacker intercepts this request and does not forward it to the actual website. On ensuing logins on the website forged by the attacker, the

user may use her U2F token, but the attacker can simply strip the second factor and forward a simple login request to the legitimate website.

Popov et al. [36] define a new standard to replace channel ID, called token binding. As of February 2016, the proposed standard is in draft status. Nevertheless, the MITM protection in U2F will not be affected by this future change. The browser will simply use the token information instead of the channel ID and proceed as before.

2.1.3 Attestation Certificates

Authenticator devices present an attestation certificate in the response to a registration challenge of a relying party. This attestation certificate can be used by the relying party to identify the authenticator device class. Relying parties may even prevent the user from registering authenticators not matching an expected attestation certificate or an expected issuer of the certificate. The U2F certification process for tokens makes sure that this certificate does not contain information identifying a specific device. Otherwise, this would permit collaborating relying parties, or one single relying party, to connect accounts using the same token. The current version of the U2F specification does not define the content of the attestation certificate. Future versions may specify the format of the content to enable the relying party to identify features of the device, e.g. the form of user consent or additional hardware features.

2.1.4 Counter Value

In every authentication response in the U2F protocol, the token includes a counter value. This value is transmitted in plain text as well as included in the signature to prevent tampering by the client between the token and relying party. The authenticator device stores this counter value and must increase it on every signature operation. The standard does not specify whether to increase this value globally for all key handles or to keep one counter per application identifier. Furthermore, the increment value of the counter is not specified, meaning consecutive values may well vary by a value greater than one.

Using this strict monotonically increasing counter value enables relying parties to detect some attacks on the authenticator device. An example of a detectable attack is the following: The user authenticates to a relying party, sending counter value 4 in the process. An attacker can clone the device afterward, which includes the counter value of 4. When the user authenticates to the same relying party two times again, the device will send the counter value 6. If the attacker tries to login later on that relying party using his cloned device, the device will send the counter value 5. The relying party can detect this mismatch and deny authentication.

This example, however, has several limitations: If the authenticator device includes a secure element, cloning of the whole device should not be possible. Nevertheless, software-based implementations of authenticator devices are permitted by the U2F standard. Also, if the attacker uses the cloned device before the user does, the attack can not be detected. The counter value is limited to 4 bytes in the message definitions of the protocol, meaning it will roll over, from the maximum value to the starting value 0, eventually. The relying party needs to handle this case thoughtfully. However, 4 bytes limit the number of operations to over 4 billion before reaching the point of rollover, which may never be achieved during the reasonable lifetime of a token.

2.1.5 Key Generation

The U2F standard mandates that the authenticator device has to create a new key pair for each registration command. The public key and a key handle to identify this key pair are exported to the relying party. If the token was to store all private keys and matching key handles, it would require a practically unlimited amount of secure storage. Thus, the standard does not necessitate the key handle to be strictly an index to a table containing all private keys.

Yubico implements this requirement of creating a new key pair for each registration despite refraining from including huge amounts of secure storage [33, 46]. On Yubico tokens, the private key is derived from the application identifier and a nonce, using a single device secret. An authentication operation includes the key handle; with that information, the device can restore the private key.

The detailed key derivation process is as follows:

1. The PRNG (pseudo random number generator) on the token generates a nonce (n).

$$n = \text{PRNG}()$$

2. An HMAC function uses the application identifier (a) and the nonce as input, and the device secret as the key (s) to generate a private key (k).

$$k = \text{HMAC}_s(a, n)$$

3. A second HMAC function uses the application identifier and the private key as input, and again the device secret as the key to calculate an intermediate value (x).

$$x = \text{HMAC}_s(a, k)$$

4. The token exports the last output of the HMAC concatenated with the nonce as the key handle (h) to the relying party.

$$h = n \,||\, x$$

5. The token multiplies the generator point (G) on the elliptic curve with the private key and exports the resulting public key (P) to the relying party.

$$P = k * G$$

For the keyed-hash function, HMAC-SHA256 is used [14]. For authentication, the relying party sends its application identifier and the key handle to the token. The token can extract the nonce from the key handle and recalculate the private key. This private key is then used to calculate the signature in the authentication process. The token will detect any modification to the key handle by comparing the intermediate value (x) from the handle to the value of its internal calculation. Security of this process seems to rely mostly on the unpredictability of the random number generator used. However, no thorough security analysis is available as of today.

2.2 Cryptographic Service Interoperability Layer

Reimair, Teufl, and Zefferer [37] define CrySIL as an approach standing the challenges of key management and utilizing cryptographic functions within heterogeneous application deployment scenarios. It offers a centralized, secure key storage and cryptographic engine to empower access to the cryptographic keys everywhere at any time. Figure 2.3 shows the structure of a CrySIL instance, containing several modules. These modules are defined as building blocks so that the modules can be implemented on any platform, and an instance can run on any device. This flexible architecture of CrySIL makes it easy to extend the system to support new use cases.

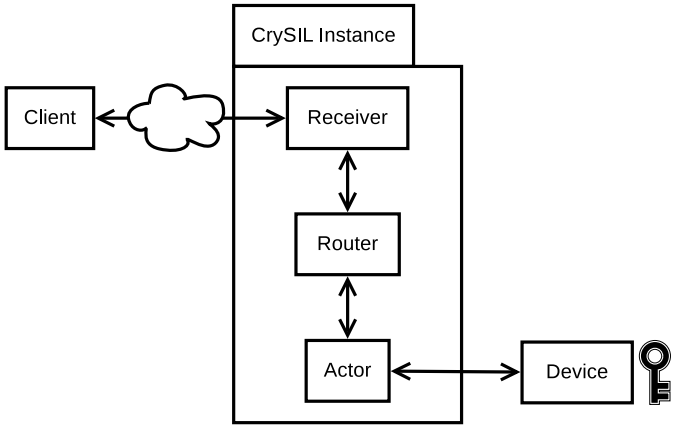


Figure 2.3: CrySIL instance with the main building blocks: Receiver, router and actor. Arrows show communications paths among the components. The cloud indicates a possible remote connection between client and instance.

Reimair et al. [38] describe MoCrySIL as a solution to carry your cryptographic keys in your pocket by running a CrySIL instance on a mobile device. The authors describe the WebVPN scheme to solve the challenge of reaching the mobile device despite ever-changing IP addresses and NAT behind routers. The scheme acts as a relay service utilizing WebSocket connections and push notifications while providing end-to-end security between the two communication partners. MoCrySIL executes cryptographic commands on the hardware-backed key store provided by

the Android system[1]. The prototypical implementation presented by the authors shows the successful integration of an Android device into the CrySIL landscape. Figure 2.4 shows the structure of a MoCrySIL instance, containing the same modules as a CrySIL instance. Between client and instance, the WebVPN relay service is added.

A TLS tunnel between the two communication partners is established to prevent this relay service from manipulating the messages between client and instance. This tunnel provides end-to-end encryption from the client to the MoCrySIL instance. The modular concept of CrySIL enables the use of the modules providing this security feature in all other CrySIL scenarios.

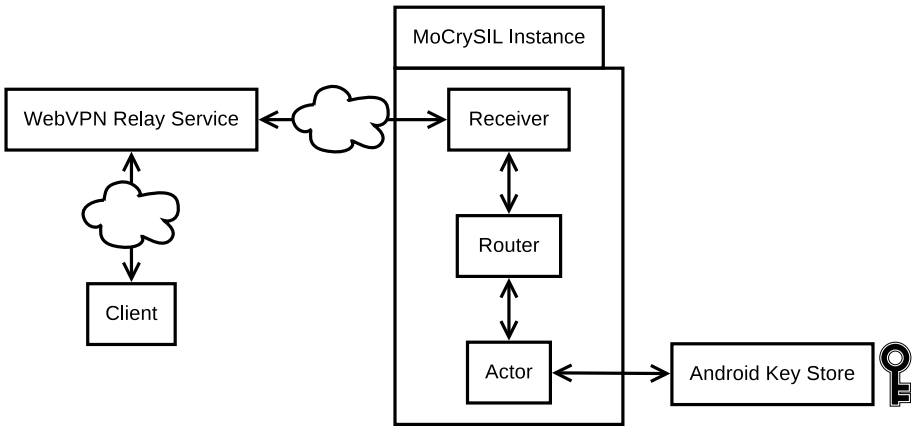


Figure 2.4: MoCrySIL instance with the main building blocks: WebVPN, receiver, router and actor. Arrows show communications paths among the components. Clouds indicate remote connections.

2.3 Microsoft Windows Credential Provider

The module responsible for handling interactive login to a local Microsoft Windows operating system is called Winlogon. Microsoft [28] describes how that module can be extended by implementing a custom credential provider. Credential providers describe the interface available to the user when logging into a Windows system. Each credential provider registered with the system can show tiles,

[1] https://developer.android.com/training/articles/keystore.html

each one representing one form of authentication. The user can usually choose between different providers and authentication methods. Credential providers included in Windows 10 involve the ordinary password credential, a variant with a shorter PIN and Windows Hello, which can authenticate the user using biometrics[2]. The credential provider framework enables convenient extensions of the Windows login system to support any form of authentication.

2.4 Summary

In this chapter, we have outlined the standards and technologies which build the foundation for our approach. U2F provides an open standard for secure second-factor authentication. We explained the sequence of steps for both basic operations, which are registration and authentication. We analyzed how the standard addresses limitations in existing approaches for 2FA, e.g. man-in-the-middle attacks. We described the CrySIL system, which exhibits a flexible and extensible architecture for cryptographic operations. Finally, we showed how the Windows login system permits extending its features by implementing custom credentials providers. In the next chapter, we shall survey existing solutions for second-factor authentication.

[2] http://windows.microsoft.com/en-us/windows-10/getstarted-what-is-hello

3 Related Work

In this chapter, we survey existing approaches to increase security in authentication processes. We limit our review to approaches for adding a strong second factor to existing user verification methods. We concentrate on authentication on the Web, as this is one of the most prevalent use cases nowadays. In this Web setting, the usual form of authentication is the username together with the password. Some of the methods we present, try to make use of the ubiquitous mobile phone for authentication. Other methods are suggesting novel approaches for 2FA, e.g. by combining biometrics and possession of cryptographic devices. Furthermore, we take a look at the adoption of the U2F standard on the client and server side. We shall see that some commercial frameworks already implement the standard in their portfolio of authentication methods.

In Section 3.1 we survey existing solutions for second-factor authentication. In Section 3.2 we examine some novel forms of authentication. In Section 3.3 we analyze the current state of U2F adoption.

3.1 Second Factor Authentication

The use of one-time passwords (OTP) on the Web is relatively wide-spread. To use them as a strong second factor, the OTP is commonly based on possession of a device, e.g. a particular hardware token or an application on a mobile phone. Generation of the codes may rely on time-synchronization between client and server or on a secret seed and an algorithm to generate a chain of passwords.

Examples for OTP methods are HOTP and TOTP. HOTP (HMAC-based OTP [30]) builds on cryptographic keyed hash functions to compute subsequent passwords. TOTP (Time-based OTP [31]) is established by adding the current timestamp to

that calculation. Both methods depend on a key exchange before calculating the first password. However, even systems using special tokens generating the OTP are vulnerable to attacks, as shown in an attack against the RSA SecurID system[1], where attackers were able to get hold of the seeds of the tokens.

OTP used for authorization in banking applications throughout Europe are called TAN (transaction authentication numbers). These short codes are often transferred out-of-band to the user, e.g. through text messages to a mobile phone, or distributed by mail in advance. Mulliner et al. [32] argue that SMS-based TAN cannot be considered secure anymore. The authors state attacks against cellular networks and specialized trojans for mobile phones as the primary reasons for this conclusion.

Dmitrienko et al. [13] show exhaustive attacks against various 2FA schemes employed by websites today, such as SMS-based TAN. The authors also show that attacks against the popular Google Authenticator application can succeed. The attacks rely on cross-platform infection to gain control over both endpoints, the PC and the mobile device. Some attacks can be mitigated by redesigning how 2FA is integrated into the services. The authors also call for more secure mobile 2FA solutions, which should leverage trusted execution environments on the device and protect against man-in-the-middle attacks.

Van Rijswijk and van Dijk [44] introduce an authentication framework based on a smartphone application scanning QR codes displayed on websites. That code contains a challenge which will be answered by the smartphone after entering the correct PIN. The challenge represents an OTP, which will be sent to the authentication endpoint of the website, thus sparing the user from entering the code manually.

Everts, Hoepman, and Siljee [15] implement a system using smartphones to sign-in users to websites using a PC or laptop. The approach supports both, usernames and passwords and an authentication based on public-key cryptography. The system enables a seamless switch from passwords to the more secure user verification later on. In the first phase, the smartphone application acts as a password

[1] https://arstechnica.com/security/2011/06/rsa-finally-comes-clean-securid-is-compromised/

manager. This solution obviously requires integration from service providers to offer this service to end-users.

Suoranta, Andrade, and Aura [42] propose a scheme to enable strong authentication based on hardware security modules found in mobile phones. The authors integrate their solution into a single sign-on (SSO) system to help users being tired of typing passwords. The targets for implementation are the Shibboleth identity provider and Nokia phones, but the modular solution supports substituting these parts. Advantages are that neither the browser nor the client application relying on the SSO provider needs to be adapted. A disadvantage of the approach is that the user needs to compare manually session identifiers to link the session and authentication to each other.

Czeskis et al. [11] offer a system called PhoneAuth that uses personal devices to provide a strong cryptographic authentication. That factor can be added to common authentication with passwords on Web services. The phone and the browser on the computer communicate over a Bluetooth channel to perform the authentication. The implementation consists of adaptions of the browser, a particular application on the phone and support on the website provider. Security of the scheme is based on public-key cryptography, but no details about the exact implementation on the phone are given. We believe that this work of Czeskis et al. is one predecessor to the U2F standard.

3.2 Alternative Approaches

The usual categorization for authentication methods is based on knowledge, possession, and inherence (biometrics). Brainard et al. [9] propose to use somebody you know as a fourth-factor authentication. The authors introduce their concept of vouching as a secure form of emergency authentication, e.g. when the user cannot access the first or second factor for authentication. Even a prototype system based on SecurID hardware tokens is described. Nevertheless, the use cases of this approach are severely restricted.

Jin, Ling, and Goh [20] propose a novel authentication approach by combining biometrics with random tokens generated by smart cards. This concept relieves the user of the burden to remember passwords completely. However, to roll out this solution, the user needs a smart card reader and a fingerprint reader to perform a login.

Sun et al. [41] define an interesting system called TouchIn that enables a 2FA method by using a multi-touch mobile device. The knowledge factor is represented by a simple drawing on the touchscreen (the curve password), whereas the biometrics factor is represented by the characteristics of the input, such as pressure and acceleration. Unfortunately, the security analysis only covers cases where the attacker can observe the drawing pattern and no sophisticated man-in-the-middle attacks on the authentication process itself.

Zwattendorfer and Tauber [47] describe how to achieve secure cloud authentication using electronic identities issued by governments. The authors show how to use the European STORK ID framework for secure and reliable authentication at applications offered by cloud service providers. This approach not only enables authentication but also for identification. However, this idea yields some privacy issues. Users often use pseudonyms for accounts on websites. In this case, they may not want to be identified by their name when logging in to this account. Furthermore, this approach would permit cooperating service providers to track users across different services.

Machani et al. [27] define UAF (the universal authentication framework), another standard developed by the FIDO Alliance. It aims at replacing passwords altogether by local authentication specific to a device. That device is authenticated to the relying party, and the user authenticates himself to that device only, e.g. by providing a fingerprint or entering a PIN. The relying party in this process can state a policy to restrict the local authentication methods valid for logging in. UAF also includes the opportunity to show the user data related to the transaction. The standard itself looks promising to provide the flexibility needed in times of fast-changing user behavior and device support.

The FIDO Alliance itself expects the standards UAF and U2F to further evolve and eventually harmonize[2]. Few products implement a UAF authenticator, most notably the fingerprint sensor on some Samsung Galaxy smartphones and fingerprint readers on Lenovo ThinkPads[3]. PayPal offers a fingerprint authentication on several Samsung mobile devices, which is based on UAF[4]. Nevertheless, the every-day user of the Web cannot use UAF to log in to any website, yet.

3.3 U2F Adoption

Lang et al. [24] show that employing a two-factor authentication solution based on U2F leads to an increased level of security and user satisfaction. The authors base their findings on a large-scale deployment within Google and on public web applications offered by Google. They show that the time spent authenticating using U2F keys (called Security Keys by the authors) is lower compared to traditional OTP methods (received either via an application or SMS). Also, no authentication failures occurred when using U2F. Such failures may occur when using traditional OTP methods and the user incorrectly the code from the mobile phone to her laptop used for the log-in.

More important for the adoption of U2F is support on all three sides, i.e. from relying parties, from client software vendors, and from hardware manufacturers. Various companies produce U2F tokens, including Yubico[5], Hypersecu[6] and an iris identity authenticator by EyeLock[7]. NXP[8] and Infineon[9] provide reference implementations of hardware tokens.

Several implementations for U2F are available as open source software. These offerings include libraries written by members of the FIDO Alliance to be integrated

[2] https://fidoalliance.org/about/faq/
[3] https://fidoalliance.org/assets/downloads/FIDO-U2F-UAF-Tutorial-v1.pdf
[4] https://www.paypal-pages.com/samsunggalaxys5/us/index.html
[5] https://www.yubico.com/products/yubikey-hardware/
[6] https://www.hypersecu.com/products/hyperfido
[7] https://www.eyelock.com/index.php/in-the-news/news/1/415/
[8] http://www.claritycommunications.com/pdf/NXP-FIDO.pdf
[9] https://www.infineon.com/fido

into existing server solutions[10]. Few websites offer support for U2F as a second-factor method, most notably Dropbox[11], Github[12], and Google[13], as of today.

Client support for U2F is implemented in the Google Chrome desktop browser[14] and its open-source variant Chromium. Support for the protocol was first implemented as an after-market extension for Chrome but was later moved into the crypto-token extension integrated directly in the browser code. Mozilla has planned to support U2F in the Firefox browser. As recently as February 2016, the implementation is underway[15]. There also exists an experimental third-party add-on trying to add U2F support to Firefox[16].

Microsoft has announced to implement support for U2F in Windows 10[17] and the Edge browser[18] but has not yet completed its implementation as of February 2016. Recently the focus of the Edge developer team has shifted towards implementing an upcoming FIDO 2.0 Web API standard[19] to be integrated in Microsoft Passport. Microsoft Passport is a two-factor authentication solution for Microsoft accounts[20], which can be used on local Windows installations.

Several authentication frameworks integrate U2F as an authentication method. One of them is Transakt U2F from Entersekt[21] where the smartphone can be utilized as a U2F token when installing an application. According to information provided by Entersekt, the private key used for U2F authentication is stored on an application server hosted by Amazon AWS, wrapped with a key stored on the phone. Since that product is still in its beta phase, no source code or detailed security analysis is available. The same holds for the SurePassID authentication

[10] https://github.com/showcases/universal-2nd-factor

[11] https://blogs.dropbox.com/dropbox/2015/08/u2f-security-keys/

[12] https://github.com/blog/2071-github-supports-universal-2nd-factor-authentication

[13] https://support.google.com/accounts/answer/6103523

[14] https://googleonlinesecurity.blogspot.com/2014/10/strengthening-2-step-verification-with.html

[15] https://bugzilla.mozilla.org/show_bug.cgi?id=1065729

[16] https://addons.mozilla.org/firefox/addon/u2f-support-add-on/

[17] https://blogs.windows.com/business/2015/02/13/microsoft-announces-fido-support-coming-to-windows-10/

[18] https://dev.windows.com/en-us/microsoft-edge/platform/status/fidou2f

[19] https://wpdev.uservoice.com/forums/257854-microsoft-edge-developer/suggestions/6830216-u2f-support-2-factor

[20] https://technet.microsoft.com/en-us/library/dn985839.aspx

[21] http://blog.entersekt.com/google-and-fido-u2f

server[22] which apparently enables various multi-factor authentication methods and tokens, including U2F, for existing server applications. Another open source example is privacyIDEA[23], which permits the integration of several two-factor authentication methods including U2F into existing applications.

The Ledger U2F applet[24] provides a Java Card implementation of the U2F standard. It may be installed on compatible Fidesmo[25] smart cards, which come with an NFC interface for use on mobile devices. Together with the Google Authenticator app for Android, this could enable a mobile use case for U2F authentication in the Chrome browser for Android[26] in the future.

3.4 Summary

In this chapter, we showed that some established authentication frameworks already integrate the U2F standard. Nearly all of the commercial applications are closed source and are not inter-operable with other solutions. Most of the solutions require the user to trust a third party with their credentials. This fact reduces the chance of broad adoption by both, service providers and end-users. There are also still usability constraints with many authentication solutions, complicating their use for the average user. As an example, all methods relying on an OTP require the user to manually copy the OTP from one device to another. Some promising ideas to strengthen user authentication on the Web exist, but they lack support from websites and clients. Overall, existing approaches do not demonstrate enough flexibility and extensibility to be used by a majority of users. In the next chapter, we shall outline our approach proposed to address shortcomings of methods described in this chapter.

[22] http://www.surepassid.com/why-we-are-unique/fido-authentication-support/
[23] https://www.privacyidea.org/
[24] https://github.com/LedgerHQ/ledger-u2f-javacard
[25] http://www.fidesmo.com/store
[26] http://about.fidesmo.com/nfc-u2f-android/

4 Approach

In the previous chapter, we showed that existing solutions for 2FA offer several limitations. Our goal is to provide an extensible and flexible approach for 2FA that is usable in various authentication use cases among different services. To achieve this goal, we combine the rising U2F standard with the open CrySIL system. The freedom of CrySIL to assemble custom instances by combining several software modules provides the flexibility needed to support numerous use cases. We implement modules for the CrySIL architecture to receive U2F messages and other modules to support miscellaneous cryptographic devices. We modify the U2F client in the Chrome browser to insert a CrySIL node. This node makes it possible to forward authentication requests from websites to any CrySIL instance. In addition to the browser scenario, we enable the use of a second factor during a Windows login procedure. There, we extend the usual combination of username and password for authentication on a local system with a U2F token. Our approach is entirely transparent to the relying party in the U2F process, as it can not differentiate between an actual U2F token and our system.

In Section 4.1 we describe our approach in general terms. In Section 4.2 we lay out applications and use cases supported by our solution. In Section 4.3 we state advantages of our approach compared to existing solutions and address limitations of our idea. In Section 4.4 we discuss specifics of the U2F protocol, namely attestation certificates and the realization thereof in our approach.

4.1 General Idea

Existing solutions for second-factor authentication exhibit a few limitations. Some of them are constrained in the applicability on the server side, limiting the use of a particular solution to one relying party. Others require the user to buy and carry special tokens, again usable only for a restricted set of service providers. No solution empowers the user to easily use his existing cryptographic devices for 2FA among different services. In general, support of open solutions is decidedly limited.

A CrySIL instance implements a centralized key storage and offers an interface for client applications to perform cryptographic operations. Those operations offered include common tasks such as encrypting and signing data. An instance also provides advanced procedures such as generating and exporting wrapped keys. The cryptographic operations performed by an authenticator device in U2F boil down to creating a new asymmetric key pair and signing challenges with it. Aligning those requirements and offerings, we think it is feasible to use a CrySIL instance as the authenticator device (token) in the U2F system.

Registering a new token for an existing account on a service generates a registration command in U2F. The token is then supposed to create a new asymmetric key pair, provide an attestation certificate and calculate the signature over the challenge. We will use the commands provided by CrySIL, namely generating a wrapped key and signing data, to fulfill this requirement.

When the user wants to log in and has a token registered with her account, the relying party requests an authentication in U2F. To answer this request, the token needs to calculate the signature over the challenge and a counter value. The relying party includes a key handle in its message to the token that enables the token to identify the key pair which was generated in the registration process described earlier. We will again use existing commands of CrySIL first to generate the key from the handle given and second to sign the data with this key. Additionally, we will implement functionality to store the counter securely and include it in the signature data.

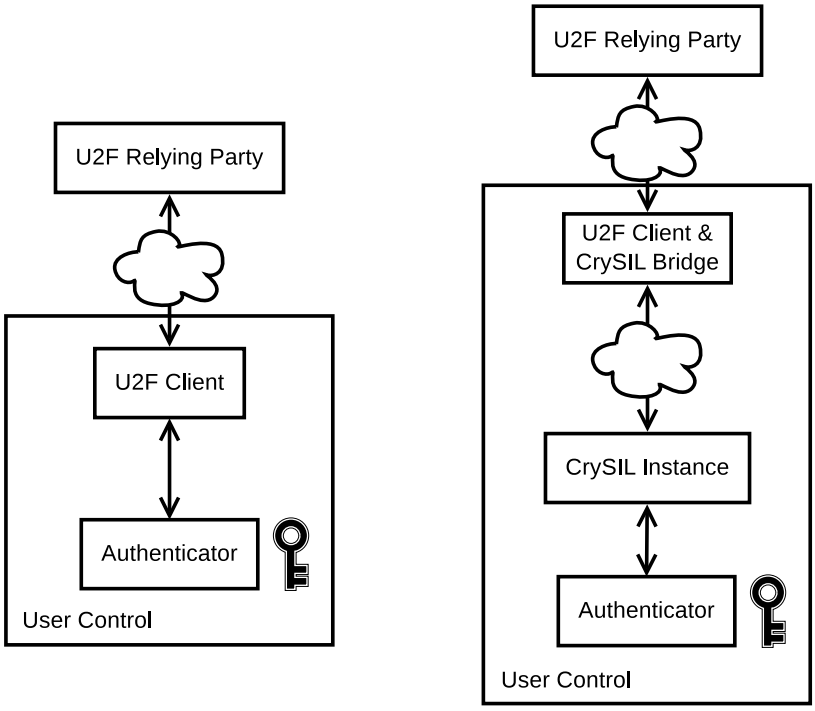


Figure 4.1: Left: General U2F approach with relying party, client, and authenticator. Right: Our approach with relying party, client including the CrySIL bridge, CrySIL instance, and authenticator. In both cases, the user has control over the client and authenticator. In our approach, the user additionally controls the CrySIL instance. The clouds in the picture denote remote connections between relying party and client, and between client and CrySIL instance.

Note that this is a simplified view of the whole process; we describe the details of the U2F protocol in Section 2.1.1 and the implementation details of our solution in Section 5.1.1. Figure 4.1 explains the differences of a common U2F setting to our approach. In the former scenario, the user has control over the U2F client in his browser and the authenticator device. In our approach, the user additionally has control over the CrySIL instance, and the U2F client is extended with a CrySIL bridge. This bridge is a realization of a CrySIL node and contains a receiver compatible with the U2F protocol and a forwarder. This forwarder sends commands to a CrySIL instance over HTTPS. The connection between the relying party and the client in the U2F setting may be a remote connection, just as the connection between the U2F client and the CrySIL instance in our approach. Figure 4.2

shows the message flow between the U2F client, the CrySIL bridge therein and the CrySIL instance in our approach. As described before, two CrySIL messages are generated for one U2F command.

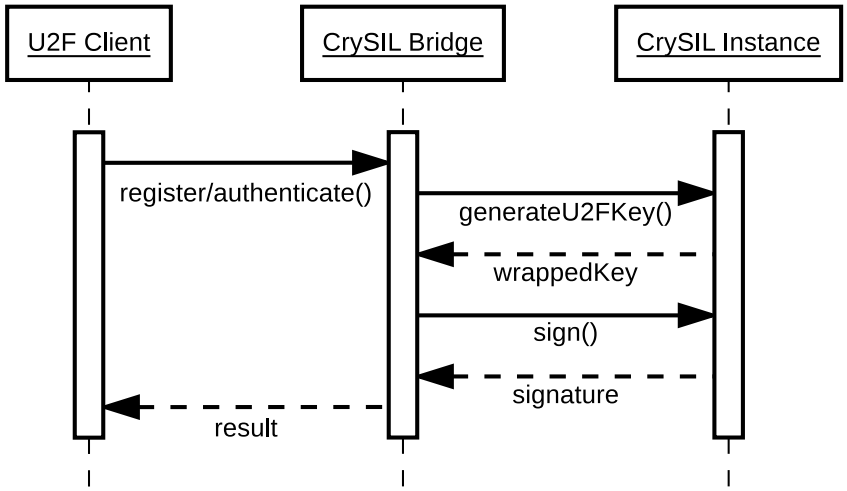


Figure 4.2: Sequence diagram showing the messages among the U2F client, the CrySIL bridge, and the CrySIL instance to execute one U2F command. The CrySIL bridge generates two CrySIL messages for one U2F request. This holds for both commands in U2F, registration and authentication.

We provide actors for the CrySIL system to execute the commands needed on a variety of devices suitable for second-factor authentication. These authenticator devices include existing smart cards and eID cards. One example for such an identity card is an Austrian health insurance card with activated citizen card functionality. Furthermore, we can use the Android key store as an authenticator device when using MoCrySIL. That solution for Android also enables using cryptographic devices connected over NFC, such as dual-interface smart cards and even U2F tokens from Yubico. Furthermore, the existent actor providing a software implementation of cryptographic commands, based on a file key store, is extended to be compatible with our solution.

4.2 Applications

One of the most prevalent use cases of authentication today is a user logging in to a website in his browser. The U2F standard explicitly focuses on that scenario. As of February 2016, only one web browser supports U2F authentication; that is Chromium or Google Chrome. To show the applicability of our solution in real-world scenarios, we adapt the U2F client implemented in the browser. The implementation of the client comprises a front-end receiving commands from relying parties and a back-end handling those commands. The browser can use our implementation of a CrySIL bridge as such a back-end. The front-end, used by developers of websites to send challenges to U2F tokens, is not altered. This separation allows our approach to retain compatibility to all websites supporting U2F, which includes Dropbox, Github, and Google. Our custom back-end forwards all U2F requests to a CrySIL instance configured by the user. Overall, our approach essentially replaces the token in the U2F process with a CrySIL instance.

We can apply the same concept to any other U2F client, not necessarily in a browser. We implement a credential provider for Microsoft Windows to showcase this possibility. We increase the security of the local login process of a Windows system by adding a second factor to the account password commonly used. The implementation involves a relying party and client for U2F as well as the CrySIL bridge. Again, any CrySIL instance can be configured by the user to be used in the authentication process. In contrast to the solution involving Chromium, the Windows credential provider additionally plays the role of a relying party. Therefore, it has to verify actively the data provided as the response by the U2F token.

The adaptability of our solution enables a mix-and-match of clients and authenticator devices to support various use cases. Any of the two clients, the Chromium browser, and the Windows login provider can talk to any CrySIL instance. That instance can pass on the requests to any device supported: Smart card, electronic identity card, Android key store, software implementation, or a genuine U2F token over NFC. Therefore, the user can use her smartphone as a secure second factor when accessing her Google account. Another option is to use her eID card as the second factor when logging in to a local Windows system.

4.3 Advantages

We extend the flexibility of the U2F process with our approach: We enable the use of existing cryptographic devices, including smart cards and similar eID cards. Given the modular concept of the CrySIL approach, one can also implement actors to support even more devices. Additionally, we show how to employ U2F in local scenarios, in contrast to the focus on Web authentication of early implementations.

Our approach keeps the benefits of U2F over traditional solutions for 2FA. We retain the property of resilience to man-in-the-middle attacks and other critical security aspects. We also maintain the user-friendliness and simplicity of U2F for the user, only requiring the user to confirm each operation on the authenticator device. Moreover, when using our approach based on an Android device, the user is relieved of connecting a particular hardware token to the device performing the login. Therefore, the user does not have to carry around yet another device for 2FA. We further keep the broad support from service providers for U2F, meaning that a significant number of users on the Web can use our approach.

In the realization of our approach, we build on the existing structure of the CrySIL system. As one consequence, the communication of the client with the CrySIL instance (acting as the authenticator device) requires an active network connection. This is no constraint for authentication on the Web, but may be of concern for a local login on a Microsoft Windows system. Furthermore, when using an Android smartphone as the authenticator device, there may be coverage or power issues.

4.4 Attestation Certificates

U2F tokens send an attestation certificate in the response to a registration challenge. The relying party can use this certificate to identify the device class and type. It may deny the user to use this specific device based on the information provided therein. Our approach includes these attestation certificates, with characteristics peculiar to the actor utilized by the CrySIL instance. In the registration

step, the key pair designated by the attestation certificate needs to be used to calculate the signature for the response message.

When using a smart card, the actor in our implementation uses a certificate stored on the card as the attestation certificate. This certificate is then used in the response to all registration commands. Relying parties can track this specific authenticator device using the attestation certificate. Though, this does not necessarily leak personal information about the user, depending on the content of the certificate.

When using an Austrian eID card, i.e. a smart card containing an implementation of the Austrian citizen card concept, a certificate for the key pair of the card is stored on the card. The subject of this certificate identifies the owner of the card by its full name. To prevent leaking this personal information to relying parties in our approach, we use a self-signed certificate instead. The actor for the eID cards creates such a certificate on every registration command. Therefore, this implementation avoids the identification of the user by the attestation certificate. Nevertheless, all attestation certificates contain the same public key, i.e. the one stored on the card.

When using the Android key store, or a software implementation as the actor for the CrySIL instance, again a self-signed certificate is exported as the attestation certificate. This implementation of self-signed certificates also prevents the relying party from gaining any information about the device class used. This feature may restrict the applicability of our solution if the relying party will only accept devices with an attestation certificate signed by certain issuers. However, this behavior of relying parties would also contradict the notion of freedom of choice in the U2F standard.

For smart cards and eID cards, the same key is used every time to answer registration commands. As a consequence, relying parties can track the authenticator device if the user connects it to different accounts. Nevertheless, a fresh key handle is created for every registration request. This procedure ensures that key handles from one relying party cannot be used to trick the card into generating signatures for another relying party.

Smart cards and similar systems such as identity cards also present another constraint: They offer no storage to save the counter required by the U2F protocol

securely. This counter needs to be included in every authentication response and needs to be increased at least globally on each operation, as we described in Section 2.1.4. As long as the CrySIL instance is running on the same computer, the counter can be stored in software and cached to disk. However, this obviously adds the possibility of an attacker modifying this value and causing the relying party to probably invalidate this authenticator device because of the mismatch in the counter values.

4.5 Summary

In this chapter, we described the general idea of our approach towards an extensible and flexible solution for 2FA. We explained how to combine the U2F standard with the CrySIL scheme. We empower the user to use a diverse array of cryptographic devices as a secure second factor by profiting from the flexibility of CrySIL. We also showed how to deploy our approach in a variety of real-world applications, ranging from authentication on the Web to local Windows login. We keep the advantages of U2F and even provide some more benefits. We addressed limitations concerning attestation certificates emerging in certain cases of our approach. In the next chapter, we shall outline the implementation of our approach, in particular, the modules developed for CrySIL enabling support for various scenarios.

5 Implementation

In the preceding chapter, we described the general idea of our approach for an extensible and flexible method for 2FA. To assemble our approach, we implement several modules for the CrySIL system. In this chapter, we explain the design decisions we took in the course of our implementation, concerning placement of the bridge and handling the concept of user presence. This bridge element will be included in the U2F clients and enables forwarding the authentication challenges to a CrySIL instance. We also describe the design of the actors in general and provide implementation details on the various modules. The implementation of the actors for the MoCrySIL instance running on an Android devices gives the user the chance to handle authentication requests with different actors and hardware tokens. Moreover, we demonstrate how to integrate our approach into the existing U2F client in the Chromium and Google Chrome browsers. To achieve this, we enhance the existing extension handling the U2F commands in the browsers. To enable another showcase for our approach, we implement a credential provider for Microsoft Windows. This credential provider enhances the local login process for the user with a second factor.

In Section 5.1 we explore the modules implemented for CrySIL. In Section 5.2 we analyze the extension for the Chromium and Google Chrome browsers. In Section 5.3 we examine the custom credential provider implemented for Microsoft Windows. In Appendix B we show screenshots of the clients in the browser, the Windows applications, and the Android application.

5.1 CrySIL Modules

A CrySIL instance is composed of several software modules: Receivers, a router, and actors. For the implementation of our approach, we develop receivers which understand U2F commands and actors compatible to various authenticator devices. The router module mediates between the receivers and actors in an instance. We extend existing U2F clients by including a CrySIL bridge. This CrySIL bridge comprises a receiver and forwarder module to send all authentication challenges from the U2F client to a CrySIL instance. We embed this bridge into existing software that implements a U2F client, e.g. a browser. Additionally, we implement several actors, each one capable of executing commands on a specific hardware device. U2F commands sent to the CrySIL instance are then executed on these devices. Our approach can either use CrySIL instances with existing actors adapted to the needs of our U2F integration or instances running one of the newly developed actors. Generally speaking, any receiver can be matched with any actor to build a custom CrySIL instance suitable for the use case in question.

We implement the following CrySIL receivers for our approach:

- A general U2F receiver as a Java module, converting incoming U2F messages into CrySIL commands.
- A receiver in JavaScript, to supplement the U2F extension for Chromium and Google Chrome.
- A receiver in C++, to extend the credential provider for Microsoft Windows.

Together with a forwarder, such a receiver constitutes a bridge between U2F and CrySIL. The forwarder sends the commands over an HTTPS connection to a CrySIL instance.

We implement the following CrySIL actors for our approach:

- An actor supporting smart cards.
- An actor supporting electronic identity cards.
- An actor running on Android supporting U2F tokens and smart cards over NFC.

In addition to the actors described preceding, we adapt the following existing actors:

- The actor using the IAIK-JCE[1] security provider for Java, backed by a file key store, executing cryptographic commands in software.
- The actor using the Bouncycastle[2] library on an Android device, using the Android key store.

5.1.1 Design Decisions

We discussed in Section 4.1 that the CrySIL bridge converts each U2F command received into two CrySIL commands which will be forwarded to a CrySIL instance. The two CrySIL commands are used first to create a wrapped key and in the second step to use this key to calculate a signature value. The relying party in U2F stores a key handle to identify the key from the registration response in subsequent authentication requests. CrySIL offers a method to generate and export wrapped keys in containers based on the cryptographic message syntax (CMS, [17]). However, this container can not be used as the key handle for U2F because the length of a handle is limited to 255 bytes in the standard. Therefore, instead of relying on the CMS format, we implement different strategies for key generation specific to the U2F use case. One of the strategies resembles the key generation algorithm used by Yubico, which we described in Section 2.1.5. As a result of this decision to implement a new key generation functionality, existing actors for the CrySIL system have to be adapted to be employed in U2F use cases.

We could implement the receiver bridging the commands between the U2F format and the CrySIL format on either side of the connection between the U2F client and the CrySIL instance. Figure 5.1 shows these alternatives. We have decided to implement it on the side of the U2F client. This choice bears the advantage that we can handle authentication for cryptographic devices on the client as intended by the CrySIL protocol. At first glance, it seems counter-intuitive to require authentication for a second factor in an authentication protocol. Nevertheless, this

[1] https://jce.iaik.tugraz.at/sic/Products/Core_Crypto_Toolkits/JCA_JCE
[2] https://www.bouncycastle.org/

step is needed to support authenticator devices which require a PIN to be entered for every command executed. One example of such a device is an Austrian health insurance card with activated citizen card functionality. This implementation choice enables the CrySIL node in the U2F client to prompt the user for that PIN. Also, other authentication methods supported by the CrySIL system could be implemented that way.

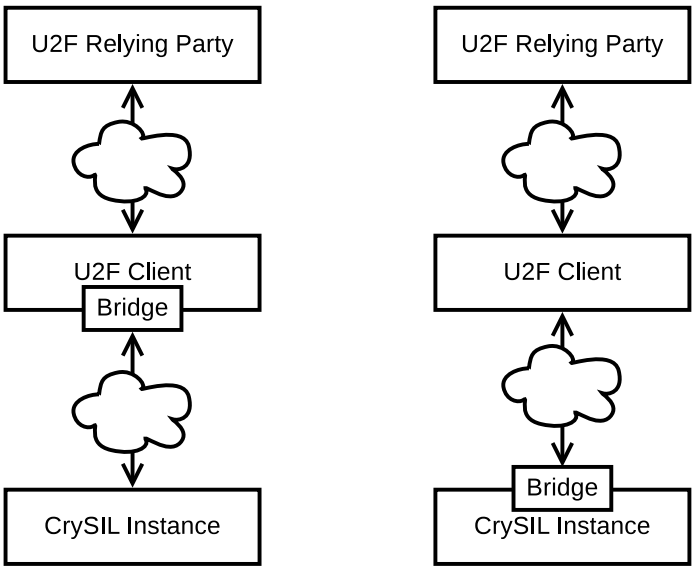


Figure 5.1: Alternatives for the placement of the bridge converting messages between the U2F and CrySIL formats. Left: Implementation of the bridge in the U2F client. Right: Implementation of the bridge as a module in the CrySIL instance. We chose to implement the alternative on the left. Clouds indicate remote connections.

As a consequence of this placement of the bridging element, a single U2F command is converted into two CrySIL commands on the client. Therefore, we need to consider another subtlety of the U2F protocol: The signature input in the authentication step includes the counter value of the authenticator device. We described this mechanism in detail in Section 2.1.4. However, this counter value is known only to the actor of the CrySIL instance. Since the bridge on the client side assembles the signature input and creates the command requesting the signature value, it cannot access the counter value. To solve this challenge, we introduce a new header type for CrySIL commands that includes the counter value. The

bridge sends this header, but with an empty value for the counter. If the actor executes a command containing such a header, it exports the counter value in the response. The client can then add this counter value to the response to the U2F relying party to conform to the standard.

The U2F standard places importance on the concept of user presence: The user has to confirm explicitly every operation on an authenticator device. We achieve a similar behavior in our approach in several ways, depending on the actor and cryptographic device used. When using an Android device in a MoCrySIL setting, the user has to choose actively which actor should handle the operation. This actor can either be the software implementation using the Android key store or the actor communicating with a second device using NFC. When using a CrySIL instance with an eID card, every signing operation on the card requires a PIN to be entered by the user. Our approach handles this by sending a CrySIL authentication challenge to the client, as described before. When using a smart card, and a PIN is set for the key pair in use, the same principle applies. However, other CrySIL actors may choose not to require the user's consent for every operation.

5.1.2 CrySIL U2F Bridge

In the preceding section, we discussed the placement of the CrySIL bridge. We embed this element into the U2F client to handle authentication requests from actors of the CrySIL instance, e.g. for smart cards. Therefore, we need to implement this bridge in the programming language of existing U2F clients we want to support. We also need to adapt these clients to forward all authentication requests to our bridge element.

Figure 5.2 shows the message flow between a U2F client, the CrySIL bridge and the CrySIL instance for a registration command. A relying party in the U2F setting issues this command when the user wants to register a new hardware token for his account. The CrySIL bridge receives all data from the U2F client, including application identifier and client data structure. First, the bridge sends a command to request the generation of a wrapped key to the CrySIL instance. The bridge uses the key from the response to this request for the next step. In that second step, the bridge requests a cryptographic signature over various data. The bridge

combines the responses from both CrySIL requests to create and send a response to the U2F client.

Figure 5.3 shows a similar message flow between the software elements for an authentication command. A relying party sends this command to request proof of possession of the hardware token from the user. The bridge creates the same two commands as in the previous case of registration: It requests a wrapped key from the instance, and uses that key to request a cryptographic signature. In contrast to the registration flow, the wrapped key is calculated from the handle and not newly generated. Also, the CrySIL instance has to insert a counter value into the signature data. Furthermore, the CrySIL bridge has to include this counter value in its response to the U2F client.

5.1.3 Smart Card Actor

To use smart cards in our approach, we implement an actor using the PKCS#11 library[3] and the PKCS#11 security provider[4] from the Institute of Applied Information Processing and Communications (IAIK). This library offers a programming interface to communicate with cryptographic devices, including smart cards and hardware security modules. Figure 5.4 presents the building blocks of the implementation of this actor. This figure also shows that the device running the CrySIL instance with this actor needs an additional smart card reader to communicate with the card. Such readers are usually connected to a USB interface.

[3] https://jce.iaik.tugraz.at/sic/Products/Core_Crypto_Toolkits/PKCS_11_Wrapper
[4] https://jce.iaik.tugraz.at/sic/Products/Core_Crypto_Toolkits/PKCS_11_Provider

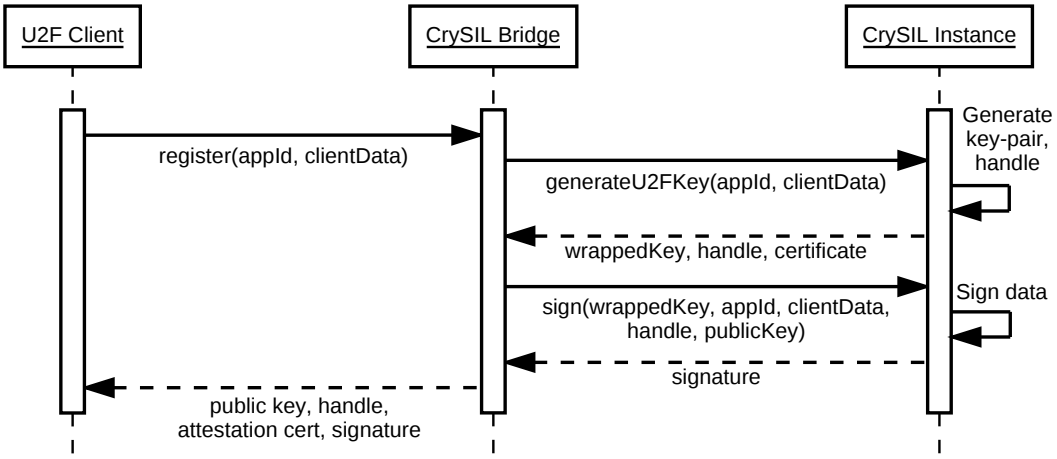


Figure 5.2: Flow of messages between a U2F client, a CrySIL bridge embedded therein, and a CrySIL instance to handle a U2F registration request from a relying party.

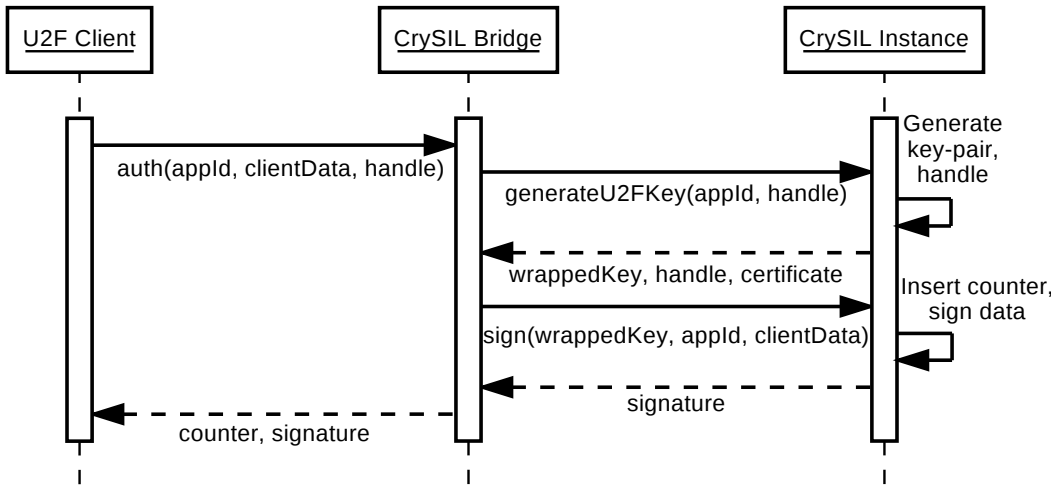


Figure 5.3: Flow of messages between a U2F client, a CrySIL bridge embedded therein, and a CrySIL instance to handle a U2F authentication request from a relying party.

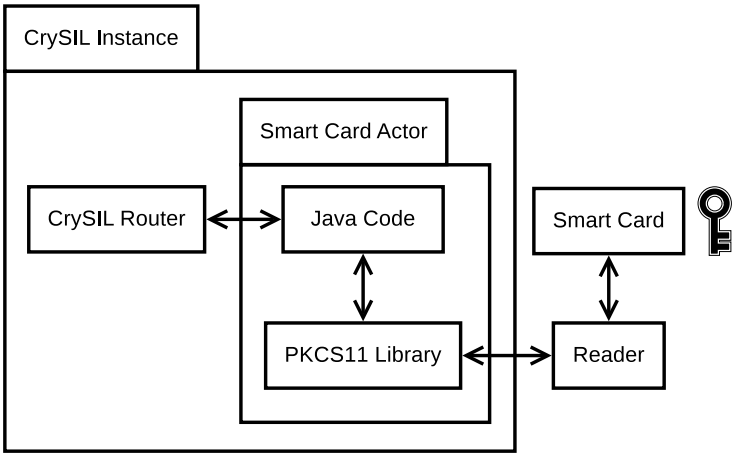


Figure 5.4: Building blocks of the smart card actor inside a CrySIL instance: The CrySIL router sends commands to the Java implementation, which forwards them with help of the PKCS#11 library to the smart card reader. That reader is usually connected to an USB interface.

The card needs to offer an ECC key pair to be usable in our approach. We use this key pair to handle all U2F operations. The elliptic curve used needs to be P-256 (or `secp256r1` equivalently) as required by the U2F protocol. Additionally, a certificate has to be stored on the card, which will be used as the attestation certificate in U2F registration responses. The private key matching the public key in that certificate also needs to be available for signing the registration response. Additionally, an RSA key pair on the card will be used to generate and verify key handles. We use this key pair to emulate the device secret in the key generation process, as it generates deterministic cryptographic signatures. In the registration step, we do not generate a new key pair on the card, but only a new key handle that gets exported to the relying party. On subsequent authentication steps, we can verify the authenticity of the key handle the relying party sends with the challenge. Any PIN required for smart card operations will be handled by the authentication mechanism of the CrySIL protocol, as we described in Section 5.1.1.

5.1.4 Electronic Identity Card Actor

The implementation of the actor for electronic identity cards is similar to the actor for smart cards. Figure 5.5 shows the building blocks of this actor for eID cards. Again, an external card reader is necessary to use the cards.

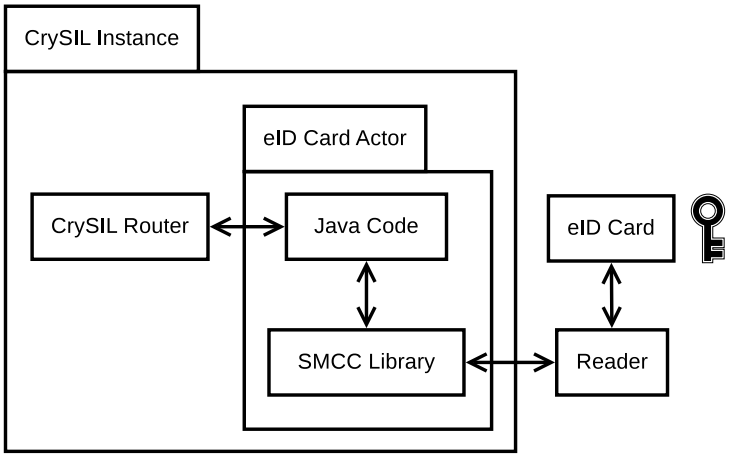


Figure 5.5: Building blocks of the electronic identity card actor inside a CrySIL instance: The CrySIL router sends commands to the Java implementation, which forwards them with help of the SMCC library to the card reader. That reader is usually connected to an USB interface.

The implementation of the module uses the smart card communication library (SMCC) from the MOCCA[5] project. MOCCA is a modular, open source citizen card environment, initiated by EGIZ[6], Austria's eGovernment innovation center. The library exposes a programming interface to perform certain operations on the cards, including signature creation. The range of cards compatible with the library includes Austrian health insurance cards with activated citizen card functionality and similar eID cards from other European countries. For application in our approach, the eID card has to provide an ECC key pair on the curve P-256 that can be used to create cryptographic signatures. In the case of the Austrian cards, this key pair is also used for qualified electronic signatures. A second key pair based on RSA needs to be available to generate and verify key handles, as we described in the preceding subsection for the actor for smart cards. These key

[5] https://joinup.ec.europa.eu/site/mocca/
[6] https://www.egiz.gv.at/

handles get exported to relying parties in the U2F registration phase. The certificate for the ECC key pair on the card usually identifies the owner. To prevent leak of this personal information in our approach, we do not use this certificate directly as the attestation certificate for U2F purposes. Instead, the actor creates a new self-signed certificate for the ECC key pair. This certificate will be signed by the RSA key pair on the card.

5.1.5 Android Implementation

When running a MoCrySIL instance on an Android device, the instance uses the actor based on the Android key store. The Android key store supports hardware-backed keys since the release of Android 4.3 in mid-2013[7]. Depending on the implementation of a trusted execution environment on the device, the key store then uses either a Secure Element, a Trusted Platform Module (TPM), or an ARM TrustZone. The capabilities of the key store were extended even further in Android 6[8], released by the end of 2015. The security of our approach relies on a device providing such a hardware-backed key store. The private key material will then never leave the secure zone on the device and cannot be extracted.

In our implementation, we use the Android key store to store a single RSA key pair. This key pair will be used to generate and verify key handles, resembling the function of a device secret on a U2F token. The approach for key generation is similar to the one used by Yubico, described in Section 2.1.5. The private key is derived from the application identifier and a nonce to create a new key pair for registering on a relying party. Thus, this actor fulfills the requirement of the U2F standard to generate a fresh key pair for every registration command. An attestation certificate will be created on-the-fly and signed with the same key pair. We also use the Android key store to securely store the counter value we need for the authentication step in the U2F protocol. Because the key store can only store cryptographic keys and certificates for those keys, we exploit the serial number of a certificate to save the counter value.

[7] https://developer.android.com/about/versions/android-4.3.html#Security
[8] https://source.android.com/security/keystore/

Additionally, we implement an actor using the NFC interface of the Android device. This actor enables the user to tap an external cryptographic token to the Android device and let the CrySIL instance execute the commands on that token. Our implementation supports two different devices, namely the YubiKey NEO[9] and the SmartCard-HSM Dual Interface Card[10]. For both devices, the actor directly sends the correct APDU (application protocol data unit) messages over the NFC interface. When using a YubiKey, the built-in U2F functionality of the token is used, and the actor does not alter the responses in any way. When using a smart card, an approach very similar to the one adopted in the smart card actor is used. Overall, this NFC actor is a viable alternative to retain strong security properties in case the key store of the Android device is not hardware-backed.

When interacting with a MoCrySIL instance, instead of contacting the instance directly, the forwarder in the U2F client contacts a WebVPN relay service first. This relay service then communicates with the Android device and transmits messages between the two communication partners. To prevent the relay service from eavesdropping on the conversation, we use end-to-end encryption as described by Reimair et al. [38]. We use the appropriate CrySIL modules to establish a TLS channel between the bridge in the U2F client and the MoCrySIL instance, wherein the latter is authenticated. The modular architecture of CrySIL allows us to use these modules in all implementations, whether a WebVPN relay service is present. Figure 5.6 shows this setup, including the relay service and the two actors.

On every incoming request, the user has to actively choose one of the actors available to handle the request. To implement this feature, we adapt the router in the MoCrySIL implementation. This interaction ensures that the user is aware of any authentication request happening and complies with the concept of user presence in the U2F standard.

[9] `https://www.yubico.com/products/yubikey-hardware/yubikey-neo/`
[10] `http://www.smartcard-hsm.com/`

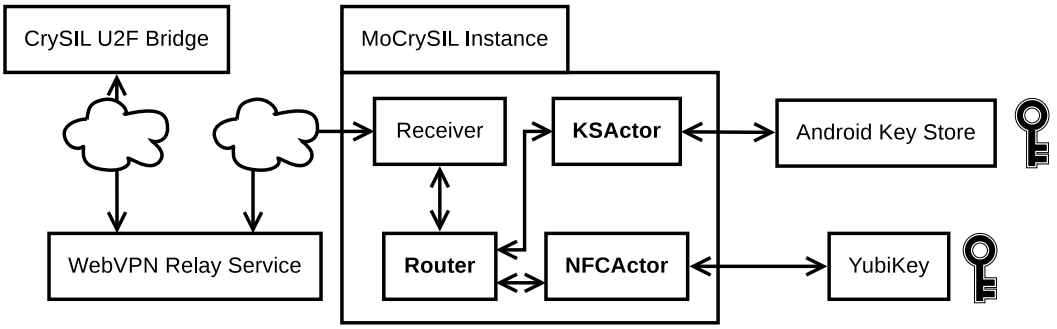


Figure 5.6: MoCrySIL instance adapted to our approach, showing our implementation of two actors and the adaptation of the router (in bold). Arrows show communications paths among the components. Clouds indicate remote connections.

5.2 Chromium Extension

U2F clearly focuses on user authentication on the Web. Several websites implement U2F as a method for second-factor authentication, and the Chromium browser (respectively Google Chrome) can be used as a client. Support for the protocol was first implemented as a third-party extension for Chromium. Older versions of the extension have provided an interface for handling U2F requests with a custom helper written in another extension. This support has been dropped by the developers when they moved the extension into the main source code of the browser. As our approach relies on this functionality to include a custom helper, we have ported that support into the current source code of the Chromium browser. As a consequence, the source code of the browser has to be patched and recompiled to use the CrySIL bridge, and therefore our approach, with U2F. Figure 5.7 shows an overview of the software components of our approach in this scenario.

5.2.1 Browser Integration

We implement an additional Chromium extension that serves as a bottom-half helper to the crypto-token extension. The developers of that extension make a distinction of bottom-half and top-half in their code. This separation enables other developers to extend the back-end (bottom-half) functionality while keeping the

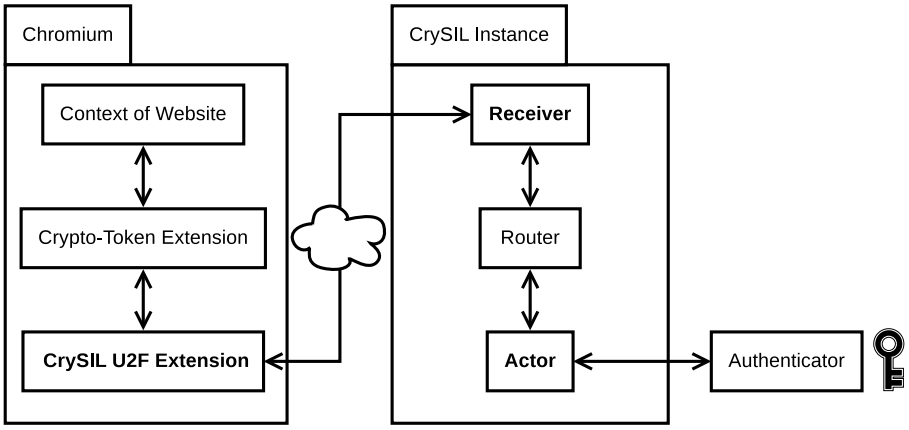


Figure 5.7: Setup of our approach in a Web use case: We add our CrySIL U2F extension to the Chromium (or Google Chrome) browser. Consequently, all U2F requests are forwarded to the CrySIL instance. We developed the components in bold specifically for our approach. The cloud indicates a remote connection between browser and CrySIL instance.

front-end (top-half) for website developers unchanged. When our extension is loaded, e.g. when the browser starts, it will register itself with the crypto-token extension. Our extension sends a message over the message passing API[11] to the crypto-token extension to perform this registration. The crypto-token extension will accept this request if an internal whitelist contains the identifier of the inquiring extension. To ensure this acceptance, we modified the code of the crypto-token extension and added the identifier of our extension.

Websites include either a custom JavaScript file to handle U2F authentication or use the scripts provided by the crypto-token extension. Either way, the website will use the message passing API to send any U2F-related request to the crypto-token extension. The crypto-token extension verifies every incoming request, e.g. it checks the origin of the website sending the request. It also creates the client data structure and may include the TLS channel identification there, if supported by the website. If the checks succeed, the extension uses the messaging API to forward the extended request to the bottom-half helper, i.e. our extension.

Figure 5.8 gives an overview of the software modules implemented in our extension. When our extension receives a request from the crypto-token extension in

[11] `https://developer.chrome.com/extensions/messaging`

the message listener, it passes it on to the receiver module. The converter module then converts the message from the U2F format into the format suitable for CrySIL. We use external open-source libraries to perform operations such as calculating message digests. The adapter module encapsulates these libraries for use in the other modules. The converter module creates two CrySIL commands: One to request the generation of a wrapped key and one to request the creation of a signature. The forwarder module sends these converted commands to the CrySIL instance that the user has configured. The authentication module may display a JavaScript dialog to prompt the user for authentication if requested by the CrySIL instance. The response is then converted back to the format suitable for the crypto-token extension and sent back.

In the conversion step from the U2F messages into CrySIL commands, we use external open-source libraries to perform operations such as calculating message digests. We do not use any functionality that is exclusively offered in Google Chrome. As a result, our extension works in both browsers, the open source Chromium browser and Google Chrome.

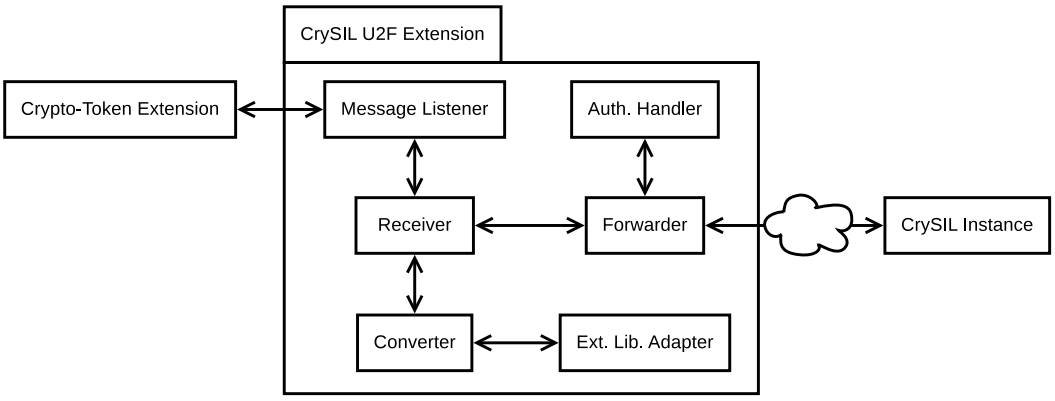


Figure 5.8: The software modules and communication paths of our extension for the Chromium and Google Chrome browsers. The message listener communicates with the existing crypto-token extension of the browser. The receiver gets this messages, and passes it on to the converter to create CrySIL commands. The adapter for external libraries calls third-party JavaScript modules for cryptographic operations. The forwarder sends the commands to the external CrySIL instance. The authentication handler processes any authentication challenges from the CrySIL instance. The cloud indicates a remote connection to the CrySIL instance, all other connections are internal.

5.2.2 Message Flow

Figure 5.9 shows the message flow among the software components described before. The process starts, when the user either tries to log in to a website (authentication in U2F) or is already logged in and wants to register a new token (registration in U2F). The sequence of messages is as follows:

1. The website fulfills the role of the relying party in U2F. It sends the proper request to the browser.
2. The browser, more precisely the crypto-token extension, takes the part of the client in U2F and receives the request from the relying party. It constructs the client data structure.
3. The crypto-token extension forwards the incoming request to our extension, that has been registered as a bottom-half helper in advance.
4. The CrySIL bridge in our extension converts the commands into a format suitable for a CrySIL instance.
5. The forwarder module sends the commands to the CrySIL instance configured by the user beforehand.
6. The CrySIL instance fulfills the role of the authenticator device in U2F and executes the cryptographic commands.
7. The CrySIL instance sends the results back to the CrySIL bridge in our browser extension.
8. Our extension passes the result on to the crypto-token extension
9. The website receives the result of the command from the browser, validates it, and either confirms the login or registers the new token.

In the step involving the CrySIL instance, an authentication for the smart card used as the authenticator device may be required. This verification is handled with a CrySIL authentication challenge, as we described preceding. Therefore, the user may be prompted to enter the PIN for the card by a simple JavaScript dialog in the browser. In Appendix A we list the format of all messages exchanged between the components.

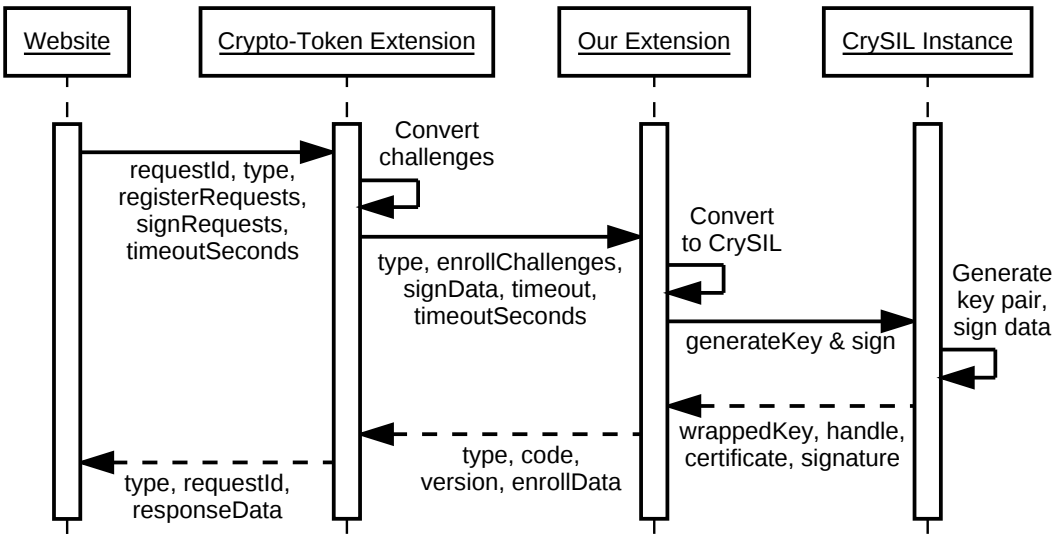


Figure 5.9: Message flow between a website issuing a U2F command, the crypto-token extension in the browser, the extension of our approach, and a CrySIL instance. For the first two connections, the message passing API in the browser is used. The last connection is established over HTTPS and used to transmit two commands.

5.3 Windows Credential Provider

There exist various credential providers for Microsoft Windows, presenting alternatives to the usual password authentication for login. However, no open-source implementation offers 2FA with support for U2F tokens. Therefore, we cannot adapt an existing solution to integrate our approach into the login system of Windows. To showcase our approach for local authentication, we implement a new credential provider. It extends the usual login (only requiring a password) with a second factor, provided by our approach. It implements functionality of a U2F client as well as functionality of a relying party. The second part is necessary because we need to generate authentication challenges and to verify the signature values returned by the CrySIL instance. Figure 5.10 shows the building blocks of our implementation for Windows in detail.

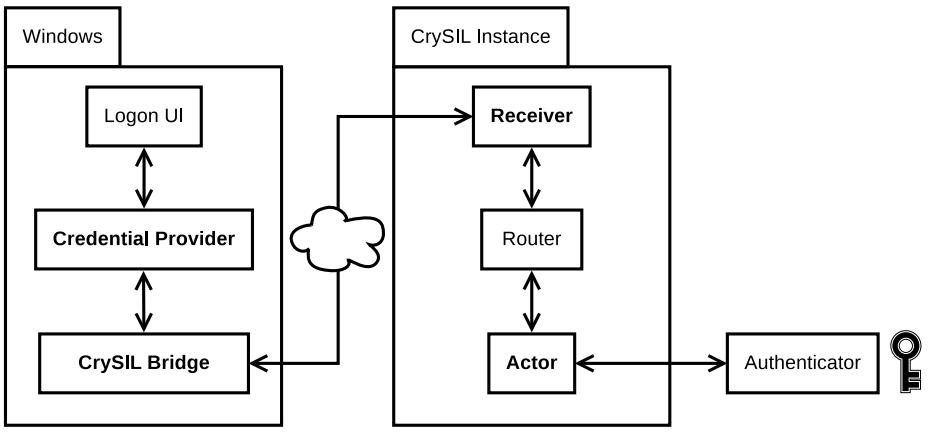


Figure 5.10: Setup of our approach for Windows: Our credential provider extends the Windows login system to support U2F functionality. The bridge forwards the authentication requests to a CrySIL instance. We developed the components in bold specifically for our approach. The cloud indicates a remote connection between Windows system and CrySIL instance.

5.3.1 Login Integration

The entry point for our credential provider is the Logon UI component from the operating system. This component queries all registered credential providers on the system to get all available credentials. Each credential of a provider is represented as one tile on the login screen. The Logon UI displays these tiles and supplies the credentials the user enters back to the credential provider. Our implementation provides one such tile for each user account. This tile shows the usual password field and information about the registered authenticator device.

Figure 5.11 shows the software modules of our credential provider. Logon UI calls our credential provider when the user selects the corresponding tile on the login screen. The credential module does not create CrySIL commands directly, to keep the software design similar to our browser extension. Instead, it creates a U2F request that is passed on to the CrySIL receiver. This module converts the incoming commands analogous to the module in the browser extension described before. We use Microsoft's Cryptography API: Next Generation (CNG)[12] to validate signatures received from the client and other cryptographic tasks.

[12] https://msdn.microsoft.com/en-us/library/windows/desktop/aa376210.aspx

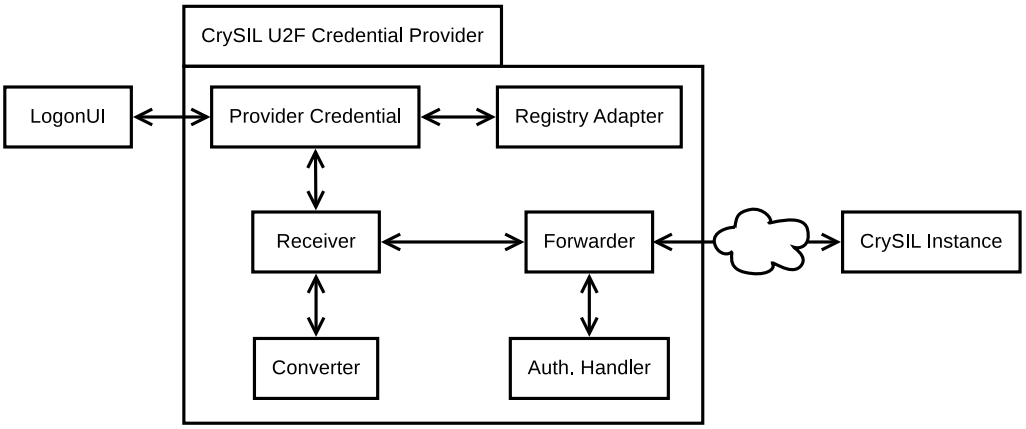


Figure 5.11: The software modules and communication paths of our credential provider for Microsoft Windows. The Logon UI of the system communicates with the credential module of our provider. This module reads information from the registry. Then, it generates a U2F command that is handled by the receiver module. This module passes it on to the converter to create CrySIL commands. The forwarder sends the commands to the external CrySIL instance. The authentication handler processes any authentication challenges from that instance. The cloud indicates a remote connection to the CrySIL instance, all other connections are internal.

5.3.2 Work Flow

Our approach for Windows includes two distinct components. The credential provider itself creates only U2F authentication commands to verify the possession of the registered token during login. The second application is used to register a new authenticator device for the login activities. Therefore, it issues only U2F registration commands. This registration application also enables modifying the connection information for the CrySIL instance that the credential provider will use during login. The Windows registry stores all configuration values along with the registered public key and key handle of the authenticator device. Figure 5.12 shows the interaction of the credential provider, the configuration app and the Windows registry to store the information.

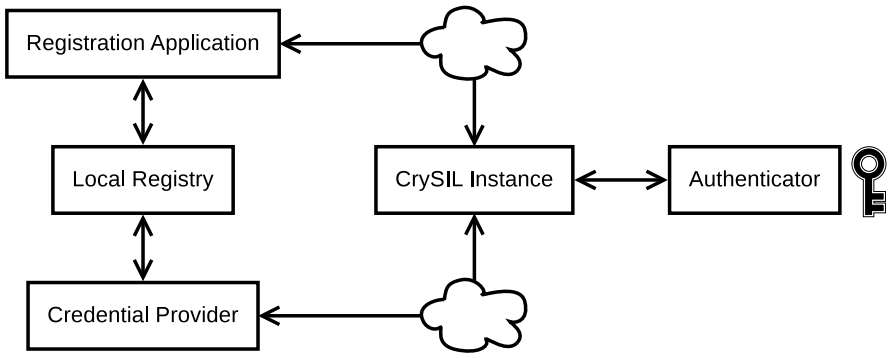


Figure 5.12: Setup of the Windows approach, showing the application needed for registration of tokens and the login provider for authentication. Arrows show communications paths among the components. Clouds indicate remote connections.

The sequence of registering an authenticator device and then logging in with it is as follows:

1. The user enters connection details in the registration application, which stores the values in the Windows registry.
2. The user registers a new authenticator device with this application. The registration application fulfills the role of the relying party in the U2F process. The CrySIL bridge included in the application plays the role of the client in U2F and uses the forwarder module to send the commands to the CrySIL instance defined by the user.
3. The CrySIL instance operates as the authenticator device in U2F and executes the registration commands.
4. The registration application verifies the result and stores the key handle and public key in the Windows registry.
5. The user logs out of the system and selects the tile of our CrySIL U2F credential provider on the user interface of the log-in screen.
6. The credential provider looks up the information about the registered authenticator device in the Windows registry.
7. The user enters his password, as usual, and confirms the login.
8. The credential provider takes the role of the relying party in U2F to authenticate the token registered to the user. The CrySIL bridge included in the credential provider acts as the client in U2F and sends the commands to the

CrySIL instance.

9. The CrySIL instance again operates as the authenticator device in U2F and handles the authentication commands.
10. The credential provider validates the result from the instance and either permits or denies access to the system. The validation of the password is not handled by our credential provider but by the Windows security authority module.

The CrySIL instance may issue an authentication challenge for the authenticator device, e.g. a smart card. The client handles this challenge by showing a simple Windows dialog, prompting the user to enter the PIN for the card.

5.4 Summary

In this chapter, we described the implementation of our approach. We demonstrated the integration of our approach in the Chromium and Google Chrome browsers and the Microsoft Windows login system. To enable the use of our approach on the Web, we augmented the browser with a custom extension, enhancing the existing U2F client. To support the Windows use case, we implemented a custom credential provider to extend the login system. Both clients forward commands to a CrySIL instance over HTTPS to execute the cryptographic commands. We presented the various actors we implemented for the CrySIL system. These actors enable assembling CrySIL instances performing cryptographic operations on a variety of devices. Devices include a smart card, a traditional U2F token, and eID cards. When running our approach on an Android device, the MoCrySIL instance can use the hardware-backed key store provided by the Android system. In the next chapter, we shall evaluate our approach and perform a security analysis.

6 Evaluation

In the previous chapter, we illustrated the implementation of our approach. In this chapter, we evaluate our approach and the security properties of it. To start, we discuss the cryptography used by U2F implementations and address concerns about the security of the standard in general. After that, we classify and categorize our approach using established frameworks to compare it to other solutions for 2FA. We show that our approach has some advantages over existing solutions since it enables the uncomplicated use of existing cryptographic devices in the authentication process. The U2F idea shows several beneficial properties when compared with established solutions for 2FA. Our approach keeps these properties intact. Furthermore, we perform a thorough security analysis of our approach based on an existing analysis of the U2F standard. Our approach increases the attack surface, as it introduces more components in the picture of user authentication. Nevertheless, the security analysis shows that, under certain assumptions, the additional threats can be mitigated. Only few residual risks remain in certain use cases of our approach.

In Section 6.1 we address properties of the U2F standard concerning the security of the implementation. In Section 6.2 we classify our approach and compare it with other solutions for 2FA. In Section 6.3 we compare our approach with existing solutions based on an established framework. In Section 6.4 we perform the detailed security analysis of our approach.

6.1 U2F Considerations

U2F uses wide-spread standards for cryptography, mainly public-key cryptography based on the P-256 elliptic curve, ECDSA to calculate signatures, and SHA256

to provide message digests. On the one hand, NIST classifies both, ECDSA based on a curve with public keys of 256-bit length and SHA256, as acceptable, meaning "the algorithm and key length are safe to use; no security risk is currently known" [6]. So ECDSA is acceptable for digital signature verification and generation, whereas SHA256 is acceptable for all hash function applications. On the other hand, NSA's Suite B[1] indicates that the curve P-384, resulting in public keys with 384-bit length, should be used for all levels of classified information. Before August 2015 and the announcement of the NSA to transition away from the advocated algorithms, the curve P-256 was included in the recommendation. Koblitz and Menezes [23] speculate about the reasons for this announcement. There is the rumor about the NSA having influenced parameters of several cryptographic standards. As a result, Bernstein and Lange [7] deem the curve P-256 as not safe.

Besides the cryptography standards, there are other security concerns about U2F in general. As an example, a relying party could match two accounts using the same token. To accomplish this, it needs to send the same key handle to tokens of different accounts. If those tokens accept the same key handle, it must be the same physical token. This matching is possible because a token in U2F has no concept of a user, but only the concept of an origin. A token generates key handles so that they are bound to the origin, but they do not store additional information. Nevertheless, while performing such a procedure, the relying party will have to take many error responses into account. Furthermore, users should get suspicious about the huge number of authentication challenges they must complete.

Authenticator devices used in U2F settings send an attestation certificate in the response to a registration challenge. A class of tokens, e.g. all Yubico Security Keys usually share one such certificate. Relying parties can use this information to prevent users from using certain tokens. Our approach primarily sends self-signed certificates as the attestation certificate. This fact may lead to the unexpected behavior of relying parties denying access when the user uses an implementation of our approach. However, our tests have shown, that Google, Dropbox, and Github do not deny access based on the attestation certificate.

[1] `https://www.nsa.gov/ia/programs/suiteb_cryptography/`

6.2 Classification

Van Rijswijk and van Dijk [44] define a classification of authentication solutions based on six categories. The categories are hardware independence, software independence, security, cost, compliance to open standards, and ease of use. Table 6.1 shows the classification of our approach compared to the classification of existing solutions given by van Rijswijk and van Dijk. For each of the categories, one solution can achieve up to five points, where "++" indicates the highest score, and "--" indicates the lowest score.

Compared with existing methods, our approach prevails in several categories. Hardware independence is given since several existing devices, e.g. smart cards and Android devices can be used in our approach. Software independence is indifferent since we need support in the client software and a CrySIL instance, but our software runs on different platforms. We show the security features later in this chapter. Cost is negligible since the user usually does not need to buy and carry new devices. Due to the use of the open U2F standard and the published CrySIL architecture, the open standards compliance is positive. Ease of use is also favorable since the user only needs to confirm the authentication operations with a simple tap.

	HW *Independence*	SW *Independence*	*Security*	*Cost*	*Open Standards*	*Ease-of-use*
Passwords	++	++	--	++	=	+/-
OTP over SMS	+	=	-	-	--	-
OTP apps	+	+/=	+	+/=	+/=	=
PKI token	--	--	++	--	=	+
Our approach	+	=	++	+	++	++

Table 6.1: Classification of our approach compared to existing authentication solutions. The categorization and ratings of existing solutions were developed by van Rijswijk and van Dijk [44].

6.3 Comparative Evaluation

Bonneau et al. [8] define a framework for the evaluation of authentication schemes on the Web. The authors include a set of 25 benefits in the categories of usability, deployability and security. Bonneau et al. conclude that no existing scheme comes close to providing all desired benefits. Lang et al. [24] perform an evaluation of the U2F standard according to that scheme. The evaluation shows that U2F offers some advantages when compared with existing hardware tokens and phone-based solutions such as OTP over SMS. U2F is partially *memorywise-effortless* because the user does not need to remember passwords. It is *efficient-to-use* and operates with *infrequent-errors* as shown in the deployment at Google. U2F offers security benefits because it is *resilient-to-phishing* and requires *no-trusted-third-party*.

We base the evaluation of our approach on the ratings for the U2F standard from Lang et al. Table 6.2 shows the results of our evaluation, with the evaluation of existing schemes given for comparison. Our approach keeps the relevant properties of U2F. In addition to that, our approach offers *quasi-nothing-to-carry* since existing devices the user is probably already carrying, e.g. mobile phones, can be used. This benefit also translates to *negligible-cost-per-user* since usually no additional devices need to be purchased. All security properties of U2F are retained, as we show in the security analysis later in this chapter. We also assign the property of *physically-effortless* to our approach, since when using an Android device for authentication, the user does only need to tap the screen. There is no need to connect a token directly to the user's computer, as it is the case with traditional security tokens. The property of *no-trusted-third-party* still applies to our approach, as the CrySIL instance is a third party in the setting of the authentication process. Nevertheless, it cannot compromise the prover's security or privacy since all key material is stored on the authenticator devices.

	Usability								Deployability						Security										
Scheme	*Memorywise-Effortless*	*Scalable-for-Users*	*Nothing-to-Carry*	*Physically-Effortless*	*Easy-to-Learn*	*Efficient-to-Use*	*Infrequent-Errors*	*Easy-Recovery-from-Loss*	*Accessible*	*Negligible-Cost-per-user*	*Server-Compatible*	*Browser-Compatible*	*Mature*	*Non-Proprietary*	*Resilient-to-Physical-Observation*	*Resilient-to-Targeted-Impersonation*	*Resilient-to-Throttled-Guessing*	*Resilient-to-Unthrottled-Guessing*	*Resilient-to-Internal-Observation*	*Resilient-to-Leaks-from-Other-Verifiers*	*Resilient-to-Phishing*	*Resilient-to-Theft*	*No-Trusted-Third-Party*	*Requiring-Explicit-Consent*	*Unlinkable*
Passwords			●		●	●	○	●	●	●	●	●	●	●		○						●	●	●	●
OTP over SMS	●	●	○		●		○	○	○			●	●	●	●	●	●	●	●	●		○		●	●
OTP with app			○		●	○	○	○	○			●	●		○	○	●	●		●		●	●	●	●
Default U2F	○	○			●	●	●		●	○		○	●	●	●	●	●	●	●	●	●	●	●	●	●
Our approach	○	○	○	●	●	●	●		●	●		○	●	●	●	●	●	●	●	●	●	●	●	●	●

● = offers the benefit; ○ = almost offers the benefit; *no circle* = does not offer the benefit.

Table 6.2: Comparative evaluation of our approach with existing methods for reference. The categorization was published by Bonneau et al. [8], whereas the assessment of the existing solutions was given by Lang et al. [24].

6.4 Security Analysis

Lindemann, Baghdasaryan, and Hill [26] provide a security reference of the two authentication standards from the FIDO Alliance, UAF and U2F. Based on this existing analysis, we perform a security analysis of our approach. We inherit some properties of the analysis, adapt them to our approach in particular, and extend other properties and requirements.

6.4.1 Model

Figure 6.1 shows the architecture of a traditional U2F solution. There, the relying party, client, and the authenticator device are the relevant components. Figure 6.2 shows the architecture of our approach and therefore the model we are evaluating. Compared with the traditional U2F architecture, our approach additionally involves a CrySIL bridge and a CrySIL instance. The U2F client is extended with the CrySIL bridge, and the CrySIL instance is placed between client and authenticator. Figure 6.3 shows that, when using a MoCrySIL instance, the architecture is even further extended, incorporating the WebVPN relay service as an additional element. This service is placed between the CrySIL bridge and the CrySIL instance. The environment of the relying party in our approach is typically unchanged when compared with other U2F applications. One exception is the case of the credential provider for Microsoft Windows, as it plays the role of the relying party as implemented for our approach.

As our flexible approach supports many use cases, depending on the actor employed, we have to consider several cases for the security analysis. The authenticator device is either a smart card, a similar system such as an eID card, a genuine U2F token, or the Android key store. The CrySIL instance contacted by the U2F client is either a CrySIL instance running on a computer or a MoCrySIL instance running on an Android device. The U2F client containing the CrySIL bridge may be incorporated in the extension for Google Chrome or the credential provider for Microsoft Windows. The relying party is either unmodified, when using the Chrome extension, or is part of our implementation for Windows. For the most of this analysis, the cases of using a smart card or an eID card will have the same

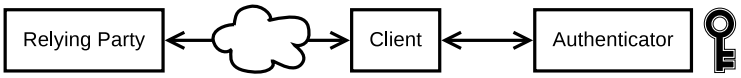


Figure 6.1: Architecture of a traditional U2F solution showing the relying party, the client, and the authenticator device (or token). The cloud indicates a remote connection between relying party and client.

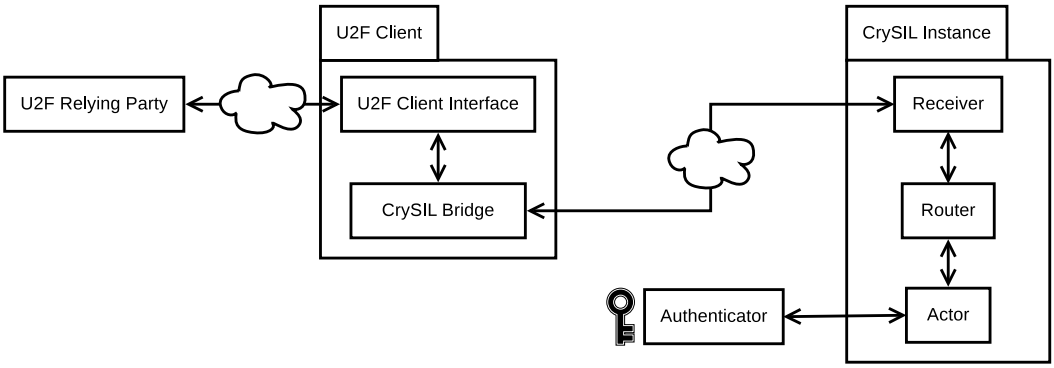


Figure 6.2: Architecture of our approach when using a CrySIL instance, extending the traditional U2F architecture. Clouds indicate remote connections.

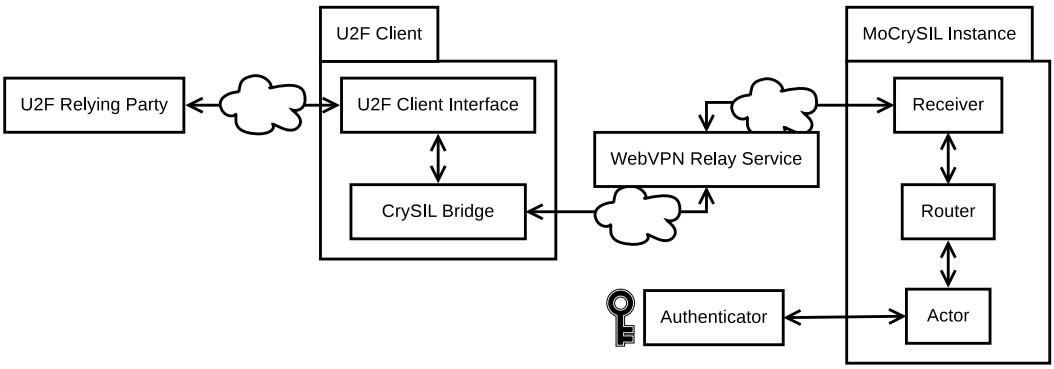


Figure 6.3: Architecture of our approach when using a MoCrySIL instance, extending the traditional U2F architecture. Compared to the case of using a CrySIL instance, the relay service for the WebVPN solution is added. Clouds indicate remote connections.

effect. Therefore, all consequences for smart cards will also apply to electronic identity cards unless stated otherwise.

6.4.2 Methodology

To start our security evaluation, we describe the methodology used throughout this section. First, we identify the assets of our approach, which include the cryptographic keys used for authentication. Second, we specify the assumptions under which we perform the security evaluation. Third, we declare the classification of attacks on our approach. Fourth, we state the security goals (also called objectives) of our approach. Fifth, we define the security measures taken in our approach. The implementation uses these measures to accomplish the security goals set before. Next, we list and describe the threats to our approach. We categorize the threats into the attack classes and analyze the impact on the assets and security goals. Finally, we identify the residual risks and give a conclusion of this analysis.

6.4.3 Assets

We determine the assets we want to protect as the following:

A-1 Cryptographic authentication key. The keys used in U2F are unique for the combination of relying party and authenticator and are generated on a registration request. Our approach may not create a new key for each registration, but use an existing key of the authenticator device, depending on the actor.

A-2 Cryptographic authentication key reference. The relying party stores the public key of the asymmetric cryptographic authentication key to verify the signatures from the authenticator.

A-3 Authenticator attestation key. Each authenticator stores a key to attest the authentication key and the type of the authenticator. These keys (and the

certificates for them) are typically shared among a device class of authenticators. Our approach may use self-signed attestation certificates, depending on the actor. Thus, there may be no explicit attestation key on the device, but only authentication keys.

A-4 Key handle of the cryptographic authentication key reference. The relying party stores the key handle of the authentication key sent by the authenticator. It may contain the encrypted private key, depending on the implementation of the token (and the actor in our approach).

6.4.4 Assumptions

We assume that the existing algorithms and systems we use for our approach have been evaluated elsewhere. Therefore, this security analysis relies on some assumptions. If any assumption listed is violated, the security goals defined in the following sections cannot be met.

The security assumptions are as following:

SA-1 The cryptographic algorithms (e.g. ECDSA for signature creation and SHA256 for message authentication) and parameters (the elliptic curve P-256) used by our system do not exhibit weaknesses that would render them useless for our purposes.

SA-2 Operating systems executing our applications enforce boundaries for privilege separation mechanisms. In particular, the Android smartphone running the MoCrySIL instance is not rooted and thus enforces sandboxing between applications.

SA-3 The applications and environment on the U2F client and the CrySIL instance act as trustworthy agents of the user.

SA-4 The applications and environment on the Windows system running our credential provider act as trustworthy agents of the relying party.

SA-5 The existing authenticator devices are secure. This assumption includes the smart cards, eID cards, and U2F tokens. Furthermore, these devices require either explicit user authentication (smart cards) or user presence

(U2F tokens) to perform cryptographic operations. Additionally, the Android key store used by the MoCrySIL instance is hardware-backed, e.g. the keys cannot be extracted from the secure zone.

SA-6 The implementation of the algorithms used in various parts of our approach is secure and correct. This assumption includes the third-party JavaScript libraries used in the browser extension and the implementation of the Windows cryptographic API used in the credential provider for Windows.

6.4.5 Attack Classes

We categorize the attacks into the following classes:

AC-1 Attacks that are automatable and lead to the attacker being able to impersonate the victim without actually involving the authenticator device.

AC-2 Same as [AC-1], but requiring the authenticator device for each authentication.

6.4.6 Security Goals

We will show that our approach keeps all the security goals originally defined for U2F intact. Therefore, we adopt the security goals declared by Lindemann, Baghdasaryan, and Hill as following:

SG-1 Strong authentication: The relying party can authenticate, that means recognize, the authenticator device of a user account with high cryptographic strength.

SG-2 Resilience against credential guessing: Protect against eavesdroppers trying to guess credentials of the authenticator device.

SG-3 Resilience against credential disclosure: Protect against phishing attacks, including attackers actively manipulating network traffic.

SG-4 Unlinkability: Relying parties should not be able to link authenticators to users, even when two or more relying parties cooperate.

SG-5 Resilience against verifier leaks: Protect against leaks from other relying parties, i.e. make sure the information known to one relying party can not be used to impersonate a user to another relying party.

SG-6 Resilience against authenticator leaks: Protect against leaks from authenticator tokens, i.e. make sure the information leaked by an authenticator can not be used to impersonate a user to another relying party.

SG-7 User presence: The user has to be present and explicitly confirm any operation before a relationship between a previously unknown relying party and the authenticator is established.

SG-8 Limited personal information: Make sure the amount of personal information leaked to the relying party is the absolute minimum needed for strong authentication.

SG-9 Attestable properties: Relying party has to be able to verify the authenticator device to assign an associated risk for that device.

SG-10 Resilience against forgery: Prevent attackers from being able to eavesdrop and modify communication to impersonate a user.

SG-11 Resilience against parallel session attacks: Prevent attackers from impersonating a user by creating a parallel session out of a wiretapped valid communication.

SG-12 Resilience against replay attacks: Prevent attackers from begin able to replay or forward an overheard session to impersonate the user.

6.4.7 Security Measures

The security measures following are based on the existing U2F security reference. We will show that our approach either keeps the security measures unimpaired or extends them where necessary.

SM-1 Key protection:

The authenticator device protects the authentication key against abuse. To achieve this measure, we rely on the security of the used smart cards and the Android key store. This assumption is covered by [SA-5]. When using the MoCrySIL solution,

the authenticator key is derived from a key stored in the hardware-backed Android key store. This key is then used by the MoCrySIL application to calculate signatures. Under security assumption [SA-2], the Android device is not rooted, and thus no other application can read this key from memory.

Supported goals: [SG-1] Strong authentication, [SG-2] Resilience against credential guessing, and [SG-3] Resilience against credential disclosure.

SM-2 Unique authentication keys:

On every incoming registration request, the authenticator device creates a new cryptographic authentication key. The implementation of the corresponding actor for the device ensures the implementation of this step. This measure cannot be achieved when using smart cards since our approach will then use a single cryptographic key pair for every request. We address this residual risk in [R-1].

Supported goals: [SG-4] Unlinkability, [SG-5] Resilience against verifier leaks, and [SG-8] Limited personal information.

SM-3 User presence:

U2F requires the user to confirm each signing operation on the authenticator token explicitly. Our approach enforces this user presence by requiring the user to either tap the NFC token to the Android smartphone, select the Android actor on the smartphone screen, or require authentication for the use of smart cards.

Supported goals: [SG-7] User presence.

SM-4 Signature counter:

U2F mandates that authenticators include an increasing counter in signature operations and responses. This feature enables relying parties to check for cloned authenticator tokens. Our approach emulates this behavior when using the actor for the Android key store. However, we have no way to store that counter value securely when using a smart card. We address this residual risk in [R-2].

Supported goals: [SG-1] Strong authentication, [SG-3] Resilience against credential disclosure and [SG-6] Resilience against authenticator leaks.

SM-5 Secure channels:

U2F requires the use of a TLS channel with server authentication between relying party and client. Our approach keeps this property when using the browser extension, as the U2F client in the browser is not directly modified. When using the credential provider for Windows, the relying party and client run in the same application, thus no secure channel is needed. We extend this property of U2F further: We use an end-to-end encrypted TLS channel between the CrySIL bridge in the U2F client and the CrySIL instance. This channel is particularly important when communicating with a MoCrySIL instance on Android device and using the WebVPN scheme.

Supported goals: [SG-7] User presence, [SG-10] Resilience against forgery, [SG-11] Resilience against parallel session attacks, and [SG-12] Resilience against replay attacks.

SM-6 Round trip integrity:

Our Windows registration application and the credential provider verify that data sent as the challenge and data received in the response are identical. This verification is not needed for the browser extension, as we only extend client functionality. In this setting, the website as the relying party will ensure the round trip integrity.

Supported goals: [SG-10] Resilience against forgery, [SG-11] Resilience against parallel session attacks, and [SG-12] Resilience against replay attacks.

SM-7 Cryptographically secure verifier database:

In a U2F implementation, the relying party stores only the public key of the authentication key and the key handle, both exported by the authenticator. This handle may represent an encrypted private key, dependent on the implementation of the token (and the actor in our approach). This measure is implemented in the Windows credential provider, as it stores these values in the Windows registry.

Supported goals: [SG-2] Resilience against credential guessing and [SG-5] Resilience against verifier leaks.

SM-8 Authenticator class attestation:

In the U2F concept, authenticator devices of the same batch of manufacturing share an attestation certificate. Our approach has several implementations of this idea, depending on the actor used: When using an eID card or the Android key store, a fresh self-signed certificate is used. In the case of smart cards, a certificate selected from the card is used. Only in the case of using a genuine U2F token, the intended attestation certificate is sent to the relying party. Altogether, this security measure may not be completely fulfilled by our approach. We address this fact as a residual risk in [R-3].

Supported goals: [SG-4] Unlinkability and [SG-9] Attestable properties.

SM-9 Channel binding:

Usage of the channel ID extension of TLS ensures the continuity of a secure channel between the relying party and the client in a U2F setting. U2F implementations in the browser employ this feature to prevent man-in-the-middle attacks. We do not modify this implementation in our browser extension. In the Windows use case, no TLS channel and thus no channel binding is necessary, as the relying party and client run on the same host.

Supported goals: [SG-1] Strong authentication, [SG-10] Resilience against forgery, [SG-11] Resilience against parallel session attacks, and [SG-12] Resilience against replay attacks.

Note that not all security measures from the existing security reference apply to U2F, as the document also covers the UAF standard. We do not need to consider the security measure of a trusted facet list, as this only applies to scenarios involving mobile applications as the relying party. Additionally, the trust store of certified authenticators is not implemented by any U2F server library. Table 6.3 shows which security goals are ensured by which security measures, and therefore gives an overview of the description list preceding.

	[SM-1] Key protection	[SM-2] Unique authentication keys	[SM-3] User presence	[SM-4] Signature counter	[SM-5] Secure channels	[SM-6] Round trip integrity	[SM-7] Cryptog. secure verifier database	[SM-8] Authenticator class attestation	[SM-9] Channel binding
[SG-1] Strong authentication	•			•					•
[SG-2] Resilience ag. credential guessing	•						•		
[SG-3] Resilience ag. credential disclosure	•			•					
[SG-4] Unlinkability		•						•	
[SG-5] Resilience ag. verifier leaks		•					•		
[SG-6] Resilience ag. authenticator leaks				•					
[SG-7] User presence			•		•				
[SG-8] Limited personal information		•							
[SG-9] Attestable properties								•	
[SG-10] Resilience ag. forgery					•	•			•
[SG-11] Resilience ag. parallel session att.					•	•			•
[SG-12] Resilience ag. replay attacks					•	•			•

Table 6.3: Mapping of security goals to security measures. A cell with a • indicates that a security goal is supported by a security measure.

6.4.8 Threats

In the following list, we identify threats to the U2F client, the CrySIL instance, the authenticator devices and our approach in general. We will assign an attack class to each threat, state which security goals are violated, and describe the consequences and possible mitigation procedures.

T-1 Mis-registration:

The attacker tricks the user into registering her token on a forged website. This means the user will log in to the forged website using her username and password, and then register a new token on this website.

Attack class: [AC-2] Automatable impersonation, requiring the authenticator device.

Violates: [SG-1] Strong authentication.

Consequence: The attacker can use the first factor (username and password) to register a token for the account of the user on the legitimate website.

Mitigations: This problem exists in the original U2F system as well and cannot be mitigated by a U2F implementation or by our approach. The relying party needs to be aware of this possibility.

T-2 U2F client or CrySIL instance corruption:

The attacker can execute code in the environment of the U2F client or the CrySIL instance.

Attack class: [AC-2] Automatable impersonation, requiring the authenticator device.

Violates: [SA-3] Clients act as trustworthy agents of the user.

Consequences: Violation of [SA-3], complete control over the authentication process. The attacker can forge all U2F requests, and forward genuine requests to an authenticator device under his control.

Mitigations: [SA-2] Operating system enforcing process boundaries. Also, the measures [SM-1] Key protection and [SM-3] User presence prevent the attacker from

gaining full access to the authenticator device. Nevertheless, a malicious implementation of a MoCrySIL instance can perform arbitrary signature operations with the keys stored in the Android key store. We address this residual risk in [R-4].

T-3 Physical user device attack:

The attacker can obtain physical access to the device running the CrySIL instance, but not the authenticator device itself. When using an Android device with a MoCrySIL instance, the user device is also the authenticator device. This option is approached in the next threat.

Attack class: [AC-2] Automatable impersonation, requiring the authenticator device.

Violates: [SA-3] Clients act as trustworthy agents of the user.

Consequences: Violation of [SA-3] by installing malicious software.

Mitigations: Measure [SM-1] Key protection mitigates this threat as the keys are not stored on the CrySIL instance, but the authenticator device. In the case of a MoCrySIL instance, the hardware-backed Android key store denies extraction of any keys, by assumption [SA-5] Secure authenticator devices.

T-4 Physical authenticator device attack:

The attacker can gain physical access to the authenticator device, e.g. by stealing it. The authenticator device may be either a smart card, an Android device, or a U2F token.

Attack class: [AC-1] Automatable impersonation.

Violates: [SA-3] Clients act as trustworthy agents of the user.

Consequences: Violation of [SA-3]. The attacker may be able to perform an offline attack, thereby impersonating the user and violating the security goal [SG-1] Strong authentication.

Mitigations: Partially by measure [SM-3] User presence: If the authenticator device is a smart card or an eID card, a PIN is required to perform cryptographic operations. In the case of a U2F token, a simple finger touch is sufficient, and no countermeasures against impersonation can be taken. We address this residual

risk in [R-4]. The Android device may have a screen lock, requiring the attacker to unlock the phone to gain access to the MoCrySIL instance. Physical attacks, trying to extract key material, are covered under assumption [SA-5] Secure authenticator devices.

T-5 Authenticator device corruption:

The attacker can foist a malicious smart card or NFC device on the user. It may be a cloned device prepared by the attacker. The user subsequently uses that malicious device to perform registration and authentication.

Attack class: [AC-1] Automatable impersonation.

Violates: [SG-1] Strong authentication in case of a manipulated algorithm, e.g. returning the same key pair for every operation, [SG-2] Resilience against credential guessing, e.g. the attacker knows the key creation algorithm and [SG-3] Resilience against credential disclosure, e.g. the key pairs created are known to the attacker.

Consequences: The attacker can use the cloned device to impersonate the user.

Mitigations: Partially by measure [SM-4] Signature counter, as the relying party can detect a cloned authenticator device. Relying parties may be able to detect malicious authenticator devices by their attestation certificate. However, a residual risk emerges and is covered under [R-4].

T-6 Windows registry read attack:

The attacker gains access to the Windows registry and reads the information stored about the authenticator devices registered with accounts on the system.

Attack class: None, as the attacker is not able to impersonate the user.

Violates: [SG-5] Resilience against verifier leaks, as the attacker may be able to factorize public keys, maybe [SG-4] Unlinkability and [SG-8] Limited personal information as described in the consequences.

Consequences: In the case of using a smart card as the authenticator device registered on the relying party, the attacker can read the (unique) public key of the card. This information can be used to link user accounts across relying parties and identify the user. This linking is not possible when using our approach with an Android device, as this actor will generate a new key pair for each registration.

Mitigations: Measure [SM-2] Unique authentication keys renders extracted key data useless for attacks on other relying parties, when not using a smart card. Measure [SM-7] Cryptographically secure verifier database ensures that the relying party stores no more than the public key and the key handle.

T-7 Windows registry modification attack:

The attacker gains write access to the Windows registry. She can then modify the information about key handles and public keys for the user accounts registered on the system.

Attack class: [AC-1] Automatable impersonation.

Violates: [SA-4] Server acts as trustworthy agent of the relying party.

Consequences: The attacker can inject her own public key and key handle, and modify the connection information to point the credential provider to a CrySIL instance under her control. Nevertheless, she would still require the password of the user account to be able to login to the Windows system.

Mitigations: The Windows registry is the only viable place to store information used by credential providers. Our approach relies on the integrity of the registry, under assumption [SA-4] Server acts as trustworthy agent of the relying party (which runs on the device of the user in this case).

T-8 Windows malware:

The attacker can execute software on the Windows machine and use the information stored in the Windows registry. This threat is similar to [T-6] Windows registry read attack.

Attack class: None, as the attacker is not able to impersonate the user.

Violates: [SG-5] Resilience against verifier leaks, maybe [SG-4] Unlinkability, and [SG-8] Limited personal information.

Consequences: The attacker can send random authentication requests to the CrySIL instance, possibly creating a denial of service attack.

Mitigations: Consequences for the user authentication to other relying parties are mitigated by measure [SM-2] Unique authentication keys. The consequence of a denial of service attack is mitigated by measure [SM-3] User presence.

T-9 Man-in-the-middle attack:

The attacker can perform a man-in-the-middle attack on the TLS connection between the CrySIL bridge (embedded in the U2F client) and the CrySIL instance talking to the authenticator device. Reimair et al. [38] state that the implementation of the TLS tunnel is somewhat susceptible to MITM attacks: The user needs to compare hash values to ensure the validity of certificates during the initial communication handshake.

Attack class: [AC-1] Automatable impersonation.

Violates: [SG-1] Strong authentication, [SG-3] Resilience against credential disclosure.

Consequences: The attacker can modify any request and response to and from the CrySIL instance. He can use the authenticator device as an oracle for signature creation. The attacker can intercept any authentication response from the client, therefore gaining access to the PIN entered by the user.

Mitigations: As described in the security analysis of the MoCrySIL system, this remains a residual risk, addressed in [R-5].

Table 6.4 gives an overview of the threats and shows which security goals are attacked by which threats.

	[T-1] Mis-registration	[T-2] U2F client or CrySIL instance corr.	[T-3] Physical user device attack	[T-4] Physical authenticator device attack	[T-5] Authenticator device corruption	[T-6] Windows registry read attack	[T-7] Windows registry modification attack	[T-8] Windows malware	[T-9] Man-in-the-middle attack
[SG-1] Strong authentication	•			•	•				•
[SG-2] Resilience ag. credential guessing					•				
[SG-3] Resilience ag. credential disclosure					•				•
[SG-4] Unlinkability						•		•	
[SG-5] Resilience ag. verifier leaks						•		•	
[SG-6] Resilience ag. authenticator leaks									
[SG-7] User presence									
[SG-8] Limited personal information						•		•	
[SG-9] Attestable properties									
[SG-10] Resilience ag. forgery									
[SG-11] Resilience ag. parallel session att.									
[SG-12] Resilience ag. replay attacks									

Table 6.4: Mapping of security goals to threats. A cell with a • indicates that a security goal is attacked by a threat.

6.4.9 Residual Risks

The following risks emerged from the threat analysis and can not be fully mitigated by our approach:

R-1 Linkability of user accounts (even shared among different relying parties) when using a smart card or an eID card. This risk arises from the fact that these cards can calculate signatures with their fixed private key only. Therefore, this key is used to handle all U2F operations. That means the same public portion of this key pair is sent to the relying party, even for different origins. This behavior violates goal [SG-4] Unlinkability.

R-2 When using a smart card or eID card, we have no means of storing the counter value needed for U2F authentication operations securely. Therefore, the counter is only stored in software as long as the CrySIL instance is running. As a consequence, relying parties may deny an authentication because the counter value is less than the value of a previous successful authentication process. That means our approach may fail to implement measure [SM-4] Signature counter correctly, thereby not fulfilling goals [SG-3] Resilience against credential disclosure and [SG-6] Resilience against authenticator leaks.

R-3 When using a smart card, the same certificate is returned as the attestation certificate for every registration command. When using an eID card, a self-signed certificate (for the same key pair every time) is returned. The actor for the Android key store returns a self-signed certificate for a new key pair on every occasion. Overall, these implementations violate [SG-4] Unlinkability and [SG-9] Attestable properties.

R-4 Authenticator device corruption can not be detected by software and therefore not mitigated by our approach. If the user is careless in handling his personal devices, an attacker can perform a targeted attack and slip malicious cryptographic devices on the user. This risk also covers the case of a malicious implementation of MoCrySIL instance installed on the user's Android device, as it can perform arbitrary operations with the keys in the Android key store. This risk may affect the security goals [SG-1] Strong authentication, [SG-7] User presence, and [SG-9] Attestable properties.

R-5 Man-in-the-middle attack against the TLS connection between CrySIL forwarder in the U2F client and MoCrySIL instance. This weakness may compromise the goals [SG-1] Strong authentication, [SG-3] Resilience against credential disclosure, and [SG-9] Resilience against forgery.

Table 6.5 gives an overview about the list preceding, showing which security goals are not entirely met by our approach.

	[R-1] Linkability of user accounts	[R-2] Storage of the counter value	[R-3] Same attestation certificate	[R-4] Authenticator device corruption	[R-5] Man-in-the-middle attack
[SG-1] Strong authentication				•	•
[SG-2] Resilience ag. credential guessing					
[SG-3] Resilience ag. credential disclosure		•			•
[SG-4] Unlinkability	•		•		
[SG-5] Resilience ag. verifier leaks					
[SG-6] Resilience ag. authenticator leaks		•			
[SG-7] User presence				•	
[SG-8] Limited personal information					
[SG-9] Attestable properties			•	•	
[SG-10] Resilience ag. forgery					•
[SG-11] Resilience ag. parallel session att.					
[SG-12] Resilience ag. replay attacks					

Table 6.5: Mapping of security goals to residual risks. A cell with a • indicates that a security goal can not be fully met, and therefore a risk remains.

6.4.10 Conclusion

This security analysis showed that our approach keeps the security properties of the U2F standard mainly intact. However, when using a smart card, or an Austrian eID card, a few risks remain, mainly because of the use of a single key pair for authentication on all relying parties. These risks are naturally not present in a traditional U2F model. However, in both cases, in our approach and the U2F approach, one risk remains: If the user is careless in handling the authenticator devices, all security measures may be worthless.

The use of one key pair for all cryptographic operations enables relying parties to track the device, and therefore the user, among different accounts and services. As a result, the user may be reluctant to use our approach with a smart card or eID card for authentication on the Web. Nevertheless, the user can benefit from the flexibility of our approach by using his own Android device. With this possibility, the user can avoid the risks mentioned before while building on the security features of U2F. We showed that in this application of our approach the only residual risk is a man-in-the-middle attack on the connection between browser and Android device. The user can minimize this risk when he makes extra effort while connecting to the Android device for the first time and carefully verifies the certificate of the CrySIL instance for the TLS tunnel.

Our approach embracing smart cards and similar systems is still suitable for local authentication, despite the risks involved. When using our credential provider for Microsoft Windows, the relying party in the process is under control of the user. Therefore, the linkability of user accounts (because of the one key pair used) may be negligible. The implementation of the credential provider could be modified not to verify the counter value provided by the authenticator device, and then even the risk of lacking secure storage for the counter value can be reduced.

6.5 Summary

In this chapter, we approached some concerns about the U2F standard and its implementations. We also provided a classification of our approach to compare it with existing solutions for 2FA. We were able to show that we keep the benefits of U2F. We even add desirable properties concerning usability and deployability to our approach. Therefore, our approach stands up well compared with other solutions. The security analysis showed that our approach keeps the desirable security features of U2F intact, whereas only a few residual risks remain. In the next chapter, we shall conclude this thesis and provide an outlook including further research opportunities.

7 Conclusions

The average user today needs to authenticate himself to many services, both online and offline. Second-factor authentication increases the security of these authentication processes. Several approaches for 2FA exist, each with distinctive advantages and limitations. Among those methods is U2F, which currently gains industry support and can be used on a variety of services on the Web today. Implementations of the U2F concept provide important security features. Nevertheless, we reasoned that the client side of U2F does not meet the acclaimed universality of the approach. Several limitations, e.g. only one usable browser client and the requirement of specific USB tokens remain.

We presented an approach for an extensible and flexible method for 2FA. We integrate the promising U2F standard into the flexible CrySIL architecture. This combination of two existing and established systems provides an advantageous approach. We replace the actual U2F token in the authentication process with a CrySIL instance. We implemented several modules to be used as the back-end for such an instance. Our implementation allows the user to use existing smart cards, eID cards, the Android key store on smartphones and even genuine U2F tokens over NFC for secure second-factor authentication.

We show the universality of our approach by implementing two different client applications. As the first application, we extend the existing U2F client implementation in the Chromium and Google Chrome browsers. As the second application, we provide a client to enhance the security of the Microsoft Windows login. The flexibility of our approach allows the user to utilize an implementation of our approach fitted to her needs. As one example, the user can use her personal smartphone as the second factor while performing a login on a website in her desktop browser. As another example, the user can augment her login to a local

Microsoft Windows system with a second factor and use a smart card or eID card for the cryptographic operations.

Our approach offers several advantages when compared to traditional methods for 2FA. We build on the strong security features of U2F, including resilience to phishing and man-in-the-middle attacks. Compared to the conventional U2F approach, we enable the uncomplicated use of existing cryptographic devices, including smart cards. Additionally, our approach makes it possible to use an Android smartphone as the authenticator device. Our approach can be extended to support even more devices, given the modular concept of the CrySIL system. Since our implementation is compatible to existing U2F relying parties, we can use our approach for authentication to various real-world services on the Web. In the security analysis, we compared our model to a traditional U2F setting. We showed that, despite the added components, we keep the security properties of the U2F standard intact. We introduce only a few residual risks, and those risks mostly arise from limitations of the used tokens, e.g. smart cards and eID cards. The user is free to select secure devices to diminish the remaining risks and strengthen the security properties of our approach. We argued that this is the case when selecting an Android device running a MoCrySIL instance as the authenticator device.

Only a few limitations arise when using our approach: When using a smart card (or an eID card), the same key is used for cryptographic operations in all authentication challenges. This key reuse contradicts the concept of one key per relying party of U2F but does not influence the strong cryptographic properties. When using the Android key store or an eID card, self-signed certificates are used as the attestation certificates. These certificates may prevent the relying party from identifying the token, which contradicts a convention of U2F. Nevertheless, tests have shown that various websites supporting U2F unconditionally accept our approach. Furthermore, the client system performing the login must be able to reach the CrySIL instance over HTTPS to perform the authentication.

Besides the strong results of our work, we identify certain areas of possible future work. Yubico, for example, is working on a token-less U2F solution for Android and iOS devices[1]. Details remain unclear at this point, but the solution may be

[1] `https://www.yubico.com/?p=92445`

implemented as follows: The Chrome browser for Android could forward U2F authentication requests to the Google Authenticator app. The app, in turn, could use a YubiKey over NFC to answer the challenge[2]. Future work may explore how to integrate our approach into this mobile scenario. This integration would enable the user to easily use existing smart cards for 2FA even when performing the login on a mobile phone.

Other browsers could also provide a target for integration of our approach. According to the bug tracker[3] at Mozilla, the implementation of a U2F client in the Firefox browser is underway. Subsequent work may cover the integration of our approach into this browser extension.

Another opportunity for future work is the development of alternative means of communication between the browser and an Android device used for authentication. If the user performs the login on his desktop device and has the smartphone in physical proximity, a connection between the two devices over the Internet using the WebVPN relay service appears cumbersome. A direct connection between the devices, e.g. over Bluetooth or WiFi, could be more convenient.

With the latest release of the Android operating system, version 6.0, developers can store ECC key pairs in the key store directly. In future work, our approach using the MoCrySIL instance could be adapted to use this feature. Currently, our approach derives the key pairs for authentication on-the-fly in software when handling an authentication request. However, when using this new feature of Android, key pairs created for a registration request could be stored in the key store. On subsequent authentication requests, the key pair stored in the key store can be used for the calculation of the signature. This change would most certainly further strengthen the security properties of our approach.

All in all, the challenge of secure authentication becomes more important as the average user today uses a vast number of accounts on various services. We take a step towards a flexible and extensible approach for 2FA with compelling security properties. We enable the uncomplicated use of existing cryptographic devices, even mobile phones, as a secure second factor. We showcase another application of cryptographically strong eID cards by integrating them into our approach.

[2] https://security.googleblog.com/2015/12/a-new-version-of-authenticator-for.html
[3] https://bugzilla.mozilla.org/show_bug.cgi?id=1065729

Overall, we considerably increase security in authentication processes, both local and on the Web. In times of increased sensibility for security and privacy, more users are eager to protect their data. We help everybody to protect precious private data by offering a secure and easy-to-use method for strong authentication.

Appendix

Bibliography

[1] ANSI. *X9.62: 2005: Public Key Cryptography for the Financial Services Industry: The Elliptic Curve Digital Signature algorithm (ECDSA)*. ANSI. 2005 (cit. on p. 10).

[2] Dirk Balfanz, Arnar Birgisson, and Juan Lang. *FIDO U2F Javascript API*. FIDO Alliance Proposed Standard. May 14, 2015. URL: https://fidoalliance.org/specs/fido-u2f-javascript-api-ps-20150514.pdf (cit. on p. 11).

[3] Dirk Balfanz and Jakob Ehrensvard. *FIDO U2F Raw Message Formats*. FIDO Alliance Proposed Standard. May 14, 2015. URL: https://fidoalliance.org/specs/fido-u2f-raw-message-formats-ps-20150514.pdf (cit. on p. 11).

[4] Dirk Balfanz and Ryan Hamilton. *Transport Layer Security (TLS) Channel IDs*. Internet Draft draft-balfanz-tls-channelid-01. IETF Secretariat, June 2013. URL: https://tools.ietf.org/html/draft-balfanz-tls-channelid-01 (cit. on p. 18).

[5] Elaine B. Barker. *Digital Signature Standard (DSS)*. Tech. rep. July 2013. DOI: 10.6028/nist.fips.186-4. URL: https://dx.doi.org/10.6028/NIST.FIPS.186-4 (cit. on p. 10).

[6] Elaine B. Barker and Allen L. Roginsky. *Transitions: Recommendation for Transitioning the Use of Cryptographic Algorithms and Key Lengths*. Tech. rep. Nov. 2015. DOI: 10.6028/nist.sp.800-131ar1. URL: https://dx.doi.org/10.6028/nist.sp.800-131ar1 (cit. on p. 61).

[7] Daniel J Bernstein and Tanja Lange. *SafeCurves: choosing safe curves for elliptic-curve cryptography*. 2013. URL: https://safecurves.cr.yp.to (visited on 03/01/2016) (cit. on p. 61).

[8] Joseph Bonneau et al. "The Quest to Replace Passwords: A Framework for Comparative Evaluation of Web Authentication Schemes." In: *2012 IEEE Symposium on Security and Privacy*. Institute of Electrical & Electronics Engineers (IEEE), May 2012. DOI: 10.1109/sp.2012.44. URL: https://dx.doi.org/10.1109/SP.2012.44 (cit. on pp. 63, 64).

[9] John Brainard et al. "Fourth-factor authentication." In: *Proceedings of the 13th ACM conference on Computer and communications security - CCS '06*. Association for Computing Machinery (ACM), 2006. DOI: 10.1145/1180405.1180427. URL: https://dx.doi.org/10.1145/1180405.1180427 (cit. on p. 27).

[10] William E. Burr et al. *Electronic Authentication Guideline*. Tech. rep. National Institute of Standards and Technology (NIST), Nov. 2013. DOI: 10.6028/nist.sp.800-63-2. URL: https://dx.doi.org/10.6028/NIST.SP.800-63-2 (cit. on p. 1).

[11] Alexei Czeskis et al. "Strengthening user authentication through opportunistic cryptographic identity assertions." In: *Proceedings of the 2012 ACM conference on Computer and communications security - CCS '12*. Association for Computing Machinery (ACM), 2012. DOI: 10.1145/2382196.2382240. URL: https://dx.doi.org/10.1145/2382196.2382240 (cit. on p. 27).

[12] Quynh H. Dang. *Secure Hash Standard*. Tech. rep. National Institute of Standards and Technology (NIST), July 2015. DOI: 10.6028/nist.fips.180-4. URL: https://dx.doi.org/10.6028/NIST.FIPS.180-4 (cit. on p. 11).

[13] Alexandra Dmitrienko et al. "On the (In)Security of Mobile Two-Factor Authentication." In: *Financial Cryptography and Data Security*. Springer Science + Business Media, 2014, pp. 365–383. DOI: 10.1007/978-3-662-45472-5_24. URL: https://dx.doi.org/10.1007/978-3-662-45472-5_24 (cit. on p. 26).

[14] Donald Eastlake and Tony Hansen. *US Secure Hash Algorithms (SHA and SHA-based HMAC and HKDF)*. RFC 6234. RFC Editor, May 2011. URL: https://www.rfc-editor.org/rfc/rfc6234.txt (cit. on p. 21).

[15] Maarten Everts, Jaap-Henk Hoepman, and Johanneke Siljee. "UbiKiMa: Ubiquitous authentication using a smartphone, migrating from passwords to strong cryptography." In: *Proceedings of the 2013 ACM workshop on Digital identity management - DIM*. Association for Computing Machinery (ACM),

2013. DOI: 10.1145/2517881.2517885. URL: https://dx.doi.org/10.1145/2517881.2517885 (cit. on p. 26).

[16] Cormac Herley and Paul van Oorschot. "A Research Agenda Acknowledging the Persistence of Passwords." In: *IEEE Security & Privacy Magazine* 10.1 (Jan. 2012), pp. 28–36. DOI: 10.1109/msp.2011.150. URL: https://dx.doi.org/10.1109/msp.2011.150 (cit. on p. 2).

[17] Russell Housley. *Cryptographic Message Syntax (CMS)*. RFC 5652. RFC Editor, Sept. 2009. URL: https://www.rfc-editor.org/rfc/rfc5652.txt (cit. on p. 42).

[18] ITU-T. *Information Technology – Abstract Syntax Notation One (ASN.1): Specification of basic notation*. Recommendation X.680. International Telecommunication Union (ITU-T), Aug. 2015. URL: http://handle.itu.int/11.1002/1000/12479 (cit. on p. 11).

[19] ITU-T. *Information Technology – ASN.1 encoding rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER)*. Recommendation X.690. International Telecommunication Union (ITU-T), Aug. 2015. URL: http://handle.itu.int/11.1002/1000/12483 (cit. on p. 11).

[20] Andrew Teoh Beng Jin, David Ngo Chek Ling, and Alwyn Goh. "Biohashing: two factor authentication featuring fingerprint data and tokenised random number." In: *Pattern Recognition* 37.11 (Nov. 2004), pp. 2245–2255. DOI: 10.1016/j.patcog.2004.04.011. URL: https://dx.doi.org/10.1016/j.patcog.2004.04.011 (cit. on p. 28).

[21] Don Johnson, Alfred Menezes, and Scott Vanstone. "The Elliptic Curve Digital Signature Algorithm (ECDSA)." In: *International Journal of Information Security* 1.1 (Aug. 2001), pp. 36–63. DOI: 10.1007/s102070100002. URL: https://dx.doi.org/10.1007/s102070100002 (cit. on p. 11).

[22] Simon Josefsson. *The Base16, Base32, and Base64 Data Encodings*. RFC 4648. RFC Editor, Oct. 2006. URL: https://www.rfc-editor.org/rfc/rfc4648.txt (cit. on p. 11).

[23] Neal Koblitz and Alfred Menezes. *A riddle wrapped in an enigma*. Tech. rep. IACR Cryptology ePrint Archive, Report 2015/1018, 2015 (cit. on p. 61).

[24] Juan Lang et al. "Security Keys: Practical Cryptographic Second Factors for the Modern Web." In: *Proceedings of the 20th International Conference on Financial Cryptography and Data Security (FC)*. Feb. 2016 (cit. on pp. 9, 29, 63, 64).

[25] Herbert Leitold, Arno Hollosi, and Reinhard Posch. "Security architecture of the Austrian citizen card concept." In: *18th Annual Computer Security Applications Conference, 2002. Proceedings.* IEEE. Institute of Electrical & Electronics Engineers (IEEE), 2002, pp. 391–400. DOI: 10.1109/csac.2002.1176311. URL: https://dx.doi.org/10.1109/csac.2002.1176311 (cit. on p. 2).

[26] Rolf Lindemann, Davit Baghdasaryan, and Brad Hill. *FIDO Security Reference*. FIDO Alliance Proposed Standard. May 14, 2015. URL: https://fidoalliance.org/specs/fido-security-ref-ps-20150514.pdf (cit. on pp. 65, 69).

[27] Salah Machani et al. *FIDO UAF Architectural Overview*. FIDO Alliance Proposed Standard. Dec. 8, 2014. URL: https://fidoalliance.org/specs/fido-uaf-v1.0-ps-20141208/fido-uaf-overview-v1.0-ps-20141208.pdf (cit. on p. 28).

[28] Microsoft. *Credential Provider driven Windows Logon Experience*. Microsoft Corporation. 2016. URL: https://go.microsoft.com/fwlink/?LinkId=717287 (cit. on p. 23).

[29] Dennis Mirante and Justin Cappos. *Understanding Password Database Compromises*. Tech. rep. TR-CSE-2013-02. Dept. of Computer Science and Engineering Polytechnic Inst. of NYU, Sept. 13, 2013. URL: https://isis.poly.edu/~jcappos/papers/tr-cse-2013-02.pdf (cit. on p. 1).

[30] David M'Raihi et al. *HOTP: An HMAC-Based One-Time Password Algorithm*. RFC 4226. RFC Editor, Dec. 2005. URL: https://www.rfc-editor.org/rfc/rfc4226.txt (cit. on p. 25).

[31] David M'Raihi et al. *TOTP: Time-Based One-Time Password Algorithm*. RFC 6238. RFC Editor, May 2011. URL: https://www.rfc-editor.org/rfc/rfc6238.txt (cit. on p. 25).

[32] Collin Mulliner et al. "SMS-based One-Time Passwords: Attacks and Defense." In: *Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2013, pp. 150–159. DOI: 10.1007/978-3-642-39235-1_9. URL: https://dx.doi.org/10.1007/978-3-642-39235-1_9 (cit. on p. 26).

[33] Dain Nilsson. *Key generation*. Yubico. URL: https://www.yubico.com/2014/11/yubicos-u2f-key-wrapping/ (visited on 02/21/2016) (cit. on p. 20).

[34] Lawrence O'Gorman. "Comparing passwords, tokens, and biometrics for user authentication." In: *Proceedings of the IEEE* 91.12 (2003), pp. 2021–2040. DOI: 10.1109/jproc.2003.819611. URL: https://dx.doi.org/10.1109/jproc.2003.819611 (cit. on p. 1).

[35] Thanasis Petsas et al. "Two-factor Authentication: Is the World Ready?: Quantifying 2FA Adoption." In: *Proceedings of the Eighth European Workshop on System Security*. Association for Computing Machinery (ACM), 2015, p. 4. DOI: 10.1145/2751323.2751327. URL: https://dx.doi.org/10.1145/2751323.2751327 (cit. on p. 2).

[36] Adrei Popov et al. *The Token Binding Protocol Version 1.0*. Internet Draft draft-ietf-tokbind-protocol-04. IETF Secretariat, Jan. 2016. URL: https://tools.ietf.org/html/draft-ietf-tokbind-protocol-04 (cit. on p. 19).

[37] Florian Reimair, Peter Teufl, and Thomas Zefferer. "WebCrySIL - Web Cryptographic Service Interoperability Layer." In: *WEBIST 2015 - Proceedings of the 11th International Conference on Web Information Systems and Technologies, Lisbon, Portugal, 20-22 May, 2015*. 2015, pp. 35–44. DOI: 10.5220/0005488400350044. URL: https://dx.doi.org/10.5220/0005488400350044 (cit. on pp. 5, 22).

[38] Florian Reimair et al. "MoCrySIL - Carry Your Cryptographic Keys in Your Pocket." In: *SECRYPT 2015 - Proceedings of the 12th International Conference on Security and Cryptography, Colmar, Alsace, France, 20-22 July, 2015.* 2015, pp. 285–292. DOI: 10.5220/0005547902850292. URL: https://dx.doi.org/10.5220/0005547902850292 (cit. on pp. 6, 22, 50, 79).

[39] Bruce Schneier. "Two-factor authentication: too little, too late." In: *Communications of the ACM* 48.4 (Apr. 2005), p. 136. DOI: 10.1145/1053291.1053327. URL: https://dx.doi.org/10.1145/1053291.1053327 (cit. on pp. 2, 18).

[40] Sampath Srinivas et al. *Universal 2nd Factor (U2F) Overview*. FIDO Alliance Proposed Standard. May 14, 2015. URL: https://fidoalliance.org/specs/fido-u2f-overview-ps-20150514.pdf (cit. on pp. 3, 8).

[41] Jingchao Sun et al. "TouchIn: Sightless two-factor authentication on multi-touch mobile devices." In: *2014 IEEE Conference on Communications and Network Security*. Institute of Electrical & Electronics Engineers (IEEE), Oct. 2014. DOI: 10.1109/cns.2014.6997513. URL: https://dx.doi.org/10.1109/cns.2014.6997513 (cit. on p. 28).

[42] Sanna Suoranta, André Andrade, and Tuomas Aura. "Strong Authentication with Mobile Phone." In: *Lecture Notes in Computer Science*. Springer Science + Business Media, 2012, pp. 70–85. DOI: 10.1007/978-3-642-33383-5_5. URL: https://dx.doi.org/10.1007/978-3-642-33383-5_5 (cit. on p. 27).

[43] Viktor Taneski, Marjan Hericko, and Bostjan Brumen. "Password security—No change in 35 years?" In: *Information and Communication Technology, Electronics and Microelectronics (MIPRO), 2014 37th International Convention on*. IEEE. 2014, pp. 1360–1365 (cit. on p. 1).

[44] Roland M. van Rijswijk and Joost van Dijk. "Tiqr: A Novel Take on Two-factor Authentication." In: *Proceedings of the 25th International Conference on Large Installation System Administration*. LISA'11. Boston, MA: USENIX Association, 2011, pp. 7–7. URL: https://dl.acm.org/citation.cfm?id=2208488.2208495 (cit. on pp. 26, 62).

[45] Catherine S. Weir et al. "Usable Security: User Preferences for Authentication Methods in eBanking and the Effects of Experience." In: *Interact. Comput.* 22.3 (May 2010), pp. 153–164. ISSN: 0953-5438. DOI: 10.1016/j.intcom.2009.10.001. URL: https://dx.doi.org/10.1016/j.intcom.2009.10.001 (cit. on p. 2).

[46] Yubico. *Key Generation*. Yubico Developer Documentation. URL: https://developers.yubico.com/U2F/Protocol_details/Key_generation.html (visited on 02/21/2016) (cit. on p. 20).

[47] Bernd Zwattendorfer and Arne Tauber. "Secure cloud authentication using eIDs." In: *2012 IEEE 2nd International Conference on Cloud Computing and Intelligence Systems*. Institute of Electrical & Electronics Engineers (IEEE), Oct. 2012. DOI: 10.1109/ccis.2012.6664435. URL: https://dx.doi.org/10.1109/ccis.2012.6664435 (cit. on p. 28).

A Message Format

In this chapter, we describe the format of all messages exchanged in a typical use case enabled by our approach: Registration of a new authenticator device on a website with U2F support while using the Google Chrome browser and our extension for the browser. Therefore, we will outline all messages in the JSON format defined by either the U2F standard or the CrySIL system.

Listing A.1 describes the general form of a CrySIL command. A single command comprises a header and a payload. The format of the payload part is specific to the commands, whereas the header is usually in the same format for all commands. The listings following omit the header if it is not specific to that command.

```
{ header: {
    type: "standardSkyTrustHeader",
    commandId: "",
    sessionId: "",
    path: [ ],
    protocolVersion: "2.0"
  },
  payload: {
    type: <specificType>
    /* other content depends on specific command */
  }
}
```

Listing A.1: General layout of CrySIL commands

A.1 Registration

The message flow for registration of a new authenticator device (or token) occurs between several software components: The website, the crypto-token extension in the browser, our extension, and the CrySIL instance. These messages match the description of the message flow we give in Section 5.1.2 and Section 5.2.2. The following steps are executed:

1. The website sends a request to the crypto-token extension:

```
{ type: "u2f_register_request",
  requestId: 1,
  registerRequests: [ {
    version: "U2F_V2",
    challenge: <challenge>,
    appId: <appId>
  } ],
  signRequests: [],
  timeoutSeconds: 29
}
```

2. The crypto-token extension generates a client data structure, and forwards the message to the bottom-half helper, i.e. our extension:

```
{ type: "enroll_helper_request",
  enrollChallenges: [ {
    version: "U2F_V2",
    challengeHash: Base64(SHA256(<clientData>)),
    /* clientData was generated by the crypto-token extension */
    appIdHash: Base64(SHA256(<appId>))
  } ],
  signData: [],
  timeout: 29,
  timeoutSeconds: 29
}
```

3. Our extension sends a request to generate a new wrapped key to the CrySIL instance. The field `encodedRandom` represents the key handle:

```
{ payload: {
    type: "generateU2FKeyRequest",
    appParam: <appIdHash>,
    clientParam: <challengeHash>,
    encodedRandom: null,
    certificateSubject: "CN=CrySIL"
  }
  /* header omitted */
}
```

4. The CrySIL instance responds with a new wrapped key:

```
{ payload: {
    type: "generateU2FKeyResponse",
    encodedRandom: <encodedRandom>,
    encodedWrappedKey: <encodedWrappedKey>,
    encodedX509Certificate: <encodedX509Certificate>
  }
  /* header omitted */
}
```

5. Our extension sends a request to calculate a signature to the CrySIL instance:

```
{ payload: {
    type: "signRequest",
    signatureKey: {
      type: "wrappedKey",
      encodedWrappedKey: <encodedWrappedKey>
    },
    algorithm: "SHA256withECDSA",
    hashesToBeSigned: [ (0x00, <appIdHash>, <challengeHash>,
      <keyHandle>, <publicKey>) ]
  }
  /* header omitted */
}
```

6. The CrySIL instance responds with the signature:

```
{ payload: {
    type: "signResponse",
    signedHashes: [ <signature> ]
    /* signature over (0x00, <appIdHash>, <challengeHash>,
       <keyHandle>, <publicKey>) */
  }
  /* header omitted */
}
```

7. Our extension responds to the crypto-token extension:

```
{ type: "enroll_helper_reply",
  code: 0,
  version: "U2F_V2",
  enrollData: Base64(0x05, <publicKey>, <keyHandleLength>,
    <keyHandle>, <attestationCertificate>, <signature>)
}
```

8. The crypto-token extension responds to the website:

```
{ type: "u2f_register_response",
  requestId: 1,
  responseData: {
    registrationData: <enrollData>,
    challenge: <challenge>,
    version: "U2F_V2",
    appId: <appId>,
    clientData: <clientData>
  }
}
```

A.2 Authentication

The message flow for authentication of an already registered authenticator token occurs between the same software components as in the case of registration: The website, the crypto-token extension in the browser, our extension, and the CrySIL instance. These messages match the description of the message flow we give in Section 5.1.2 and Section 5.2.2. The following steps are executed:

1. The website sends a request to the crypto-token extension:

```
{ type: "u2f_sign_request",
  requestId: 1,
  registerRequests: [],
  signRequests: [ {
    version: "U2F_V2",
    challenge: <challenge>,
    keyHandle: <keyHandle>,
    appId: <appId>
  } ],
  timeoutSeconds: 29
}
```

2. The crypto-token extension generates a client data structure, and forwards the message to the bottom-half helper, i.e. our extension:

```
{ type: "sign_helper_request",
  enrollChallenges: [],
  signData: [ {
    version: "U2F_V2",
    challengeHash: Base64(SHA256(<clientData>)),
    /* clientData was generated by the crypto-token extension */
    appIdHash: Base64(SHA256(<appId>)),
    keyHandle: <keyHandle>
  } ],
  timeout: 29,
  timeoutSeconds: 29
}
```

3. Our extension sends a request to generate a new wrapped key to the CrySIL instance:

```
{ payload: {
    type: "generateU2FKeyRequest",
    appParam: <appIdHash>,
    clientParam: null,
    encodedRandom: <keyHandle>,
    certificateSubject: "CN=CrySIL"
  }
  /* header omitted */
}
```

4. The CrySIL instance responds with a new wrapped key:

```
{ payload: {
    type: "generateU2FKeyResponse",
    encodedRandom: <encodedRandom>,
    encodedWrappedKey: <encodedWrappedKey>,
    encodedX509Certificate: <encodedX509Certificate>
  }
  /* header omitted */
}
```

5. Our extension sends a request to create a signature to the CrySIL instance:

```
{ header: {
    type: "u2fHeader",
    counter: 0
    /* remaining values unchanged */
  },
  payload: {
    type: "signRequest",
    signatureKey: {
      type: "wrappedKey",
      encodedWrappedKey: <encodedWrappedKey>
    },
    algorithm: "SHA256withECDSA",
    hashesToBeSigned: [ (<appIdHash>, 0x01,
      0x00, 0x00, 0x00, 0x00, <challengeHash>) ]
  }
}
```

6. The CrySIL instance responds with the signature, plus the counter value:

```
{ header: {
    type: "u2fHeader",
    counter: <counter>
    /* remaining values unchanged */
  },
  payload: {
    type: "signResponse",
    signedHashes: [ <signature> ]
    /* signature over (<appIdHash>, 0x01, <counterArray>,
                       <challengeHash>) */
    /* <counterArray> represents the <counter> in 4 bytes
  }
}
```

7. Our extension responds to the crypto-token extension:

```
{ type: "sign_helper_reply",
  code: 0,
  responseData: {
    version: "U2F_V2",
    challengeHash: <challengeHash>,
    appIdHash: <appIdHash>,
    signatureData: Base64(0x01, <counterArray>, <signature>)
  }
}
```

8. The crypto-token extension responds to the website:

```
{ type: "u2f_sign_response",
  requestId: 1,
  responseData: {
    signatureData: <signatureData>,
    challenge: <challenge>,
    version: "U2F_V2",
    appId: <appId>,
    keyHandle: <keyHandle>,
    clientData: <clientData>
  }
}
```

B Screenshots

Figure B.1 shows the Chromium browser with our extension enabled. The extension offers a button in the toolbar of the browser. Clicking this button, the user can enter the URL of the CrySIL instance that will handle all U2F authentication requests.

Figure B.2 shows the login screen for a user on a local Microsoft Windows system with our credential provider selected. There, the user can enter the password as usual. Below the password field, we provide detailed information about the authenticator device registered with this account. The user has still the option to select the common login method of a password only by selecting the other tile presented at the bottom of the screen.

Figure B.3 shows the second application of our approach for Microsoft Windows. This application is used to configure the connection to the CrySIL instance and to register a new token. This information will be stored in the Windows registry and used by the credential provider on login.

Figure B.4 shows the MoCrySIL instance running on an Android device. On the screenshot, an incoming request is pending, and the user can choose which actor shall perform the operation.

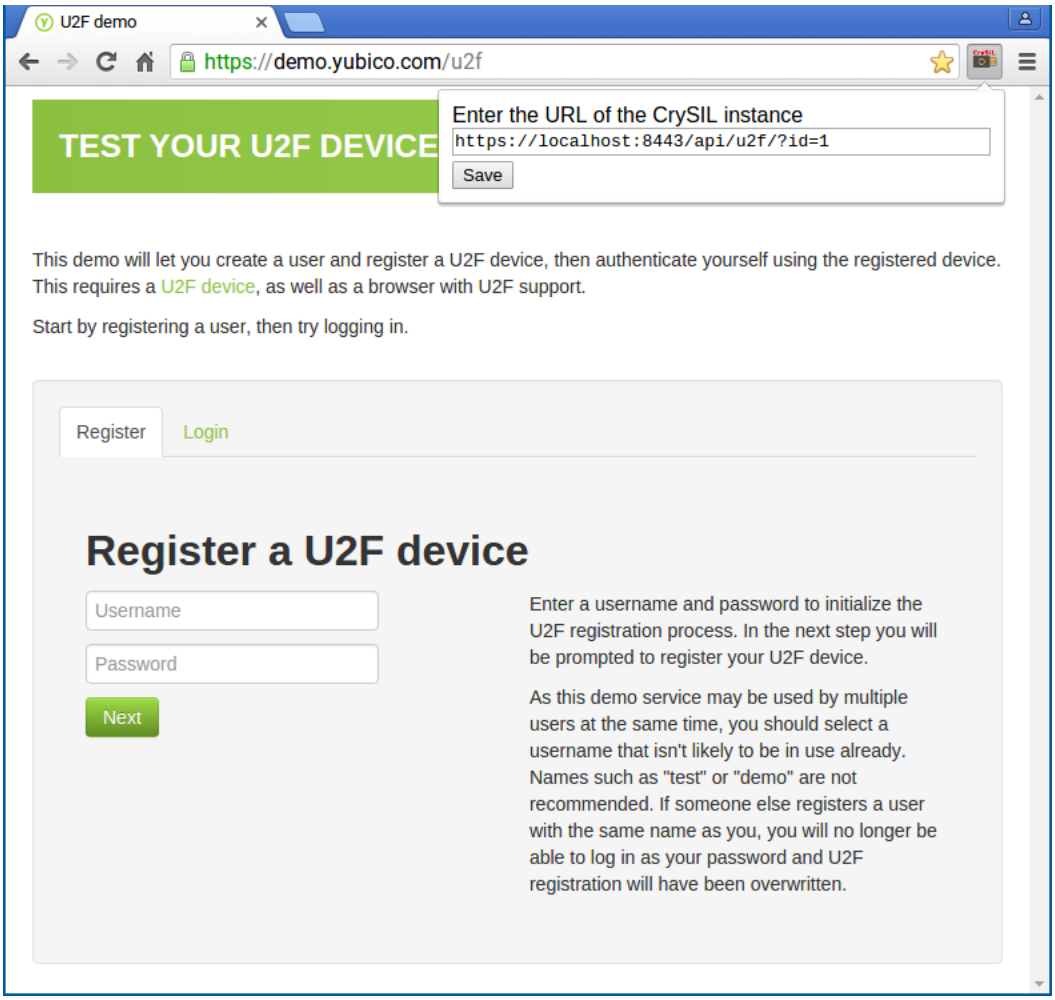
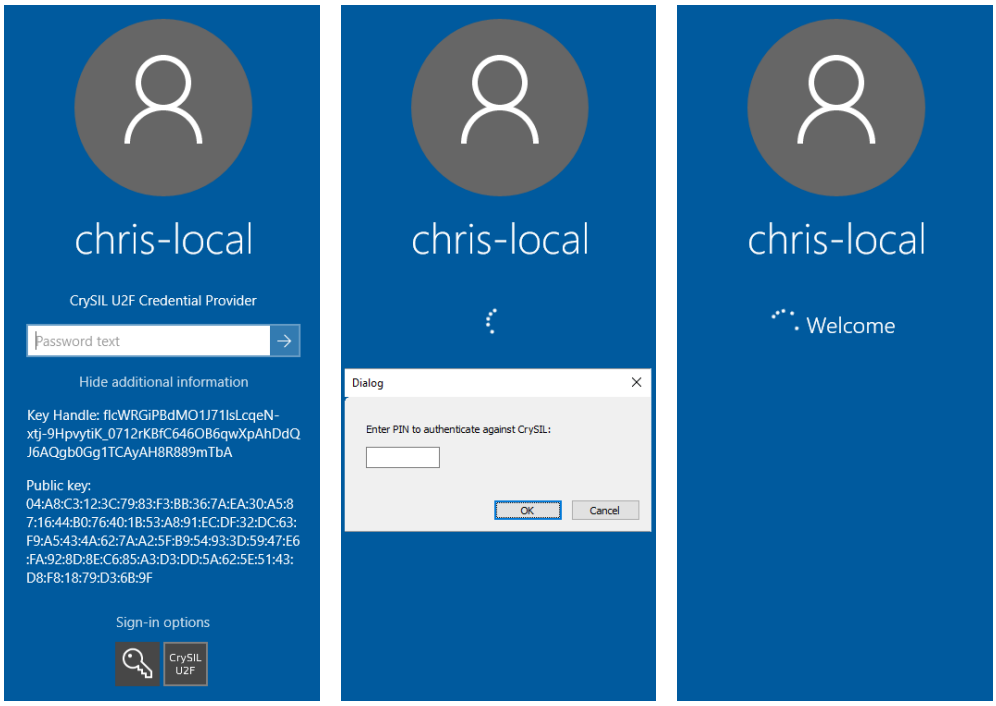


Figure B.1: Screenshot of the Chromium browser with our extension installed and the configuration dialog opened (upper right corner). The user can enter the URL of the CrySIL instance that will handle all U2F authentication requests.



(a) The credential provider shows details of the token registered for this account. At the bottom of the screen, the user can select other credential providers installed on the system.

(b) The user can respond to authentication challenges, e.g. a PIN for an eID card, by entering the value in a dialog box.

(c) After successful verification of the authenticator device, the credential provider logs in the user.

Figure B.2: Screenshots of the login flow with our credential provider for Microsoft Windows.

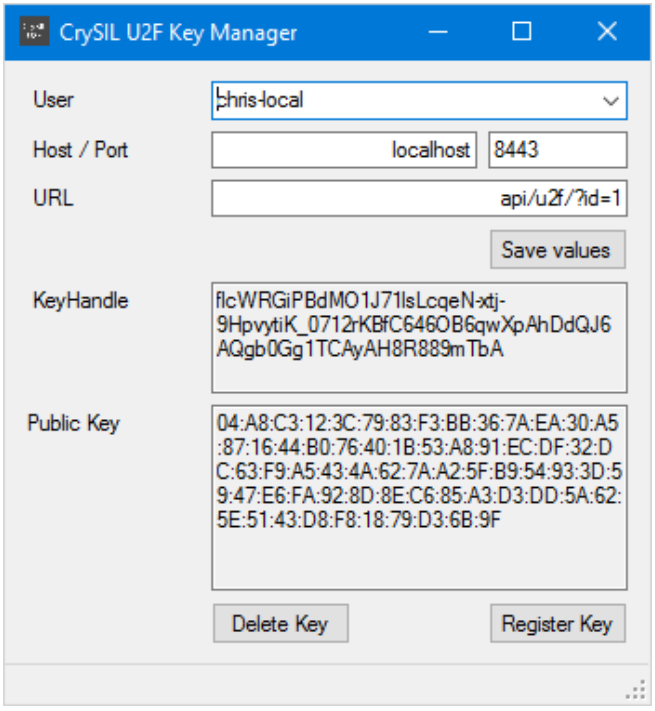


Figure B.3: Screenshot of the second application for Microsoft Windows, used for configuration of the credential provider. The user can enter the connection information to the CrySIL instance and register a new token.

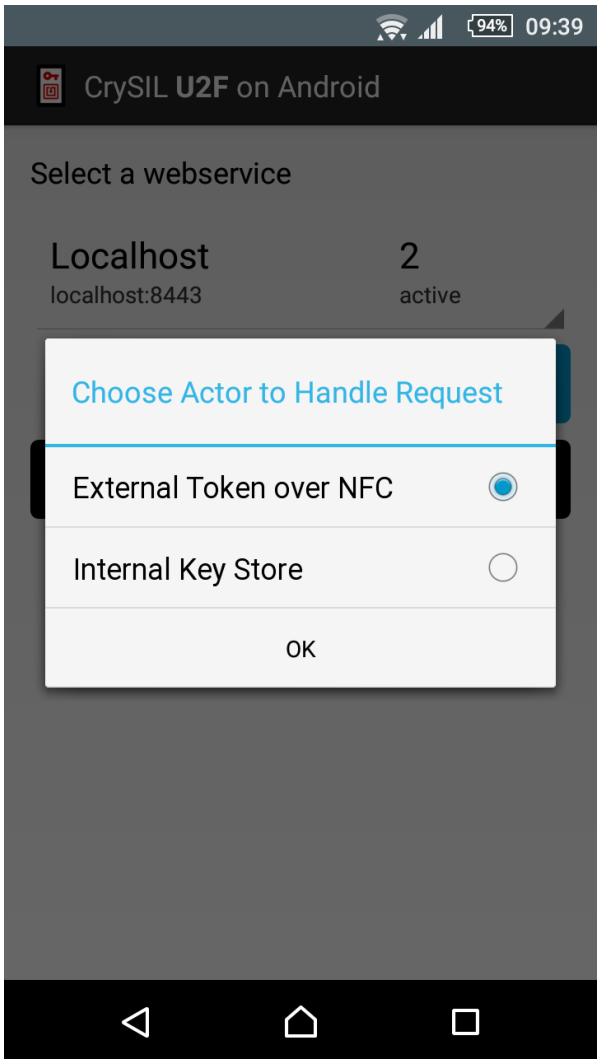


Figure B.4: Screenshot of the Android application when receiving a registration or authentication request. The user can select which actor of the MoCrySIL instance shall perform the operation.