Unlinkable Delegation of WebAuthn Credentials

Nick Frymann
n.frymann@surrey.ac.uk
[1]Surrey Centre for Cyber Security
University of Surrey
Guildford, UK

Daniel Gardham[1,2]
daniel.gardham@rhul.ac.uk
[2]Information Security Group
Royal Holloway, University of London
Egham, UK

Mark Manulis
mark@manulis.eu
Universität der Bundeswehr München
Munich, Germany

ABSTRACT

The W3C's WebAuthn standard employs digital signatures to offer phishing protection and unlinkability on the web using authenticators which manage keys on behalf of users. This introduces challenges when the account owner wants to delegate certain rights to a proxy user, such as to access their accounts or perform actions on their behalf, as delegation must not undermine the decentralisation, unlinkability, and attestation properties provided by WebAuthn.

We present two approaches, called *remote* and *direct* delegation of WebAuthn credentials, maintaining the standard's properties. Both approaches are compatible with Yubico's recent Asynchronous Remote Key Generation (ARKG) primitive proposed for backing up credentials. For remote delegation, the account owner stores delegation credentials at the relying party on behalf of proxies, whereas the direct variant uses a delegation-by-warrant approach, through which the proxy receives delegation credentials from the account owner and presents them later to the relying party. To realise direct delegation we introduce Proxy Signature with Unlinkable Warrants (PSUW), a new proxy signature scheme that extends WebAuthn's unlinkability property to proxy users and can be constructed generically from ARKG.

We discuss an implementation of both delegation approaches, designed to be compatible with WebAuthn, including extensions required for CTAP, and provide a software-based prototype demonstrating overall feasibility. On the performance side, we observe only a minor increase of a few milliseconds in the signing and verification times for delegated WebAuthn credentials based on ARKG and PSUW primitives. We also discuss additional functionality, such as revocation and permissions management, and mention usability considerations.

KEYWORDS

WebAuthn/FIDO2, privacy, delegation, proxy signatures, authentication

1 INTRODUCTION

With ever-growing reliance on web-based services, from online shopping and banking to employee intranets and cloud storage, the need to keep accounts secure is imperative. Developments in web-based authentication standards, such as FIDO (Federated Online IDentity) Universal 2nd Factor (U2F) [54] and WebAuthn [35] aim to reduce reliance on passwords and one-time passcodes (OTPs), such as HOTP [40], TOTP [41], and insecure SMS variants [42], by enabling the use of hardware and platform authenticators that manage asymmetric and unlinkable signing keys on behalf of users.

WebAuthn's digital signatures and the use of independent signing keys offer stronger security protection and unlinkability for web accounts, yet introduce challenges when the account owner wishes to delegate certain rights to some proxy user, such as to access their accounts or perform actions on their behalf, possibly for a specific time period. This is particularly useful in the enterprise environment, where employees may be required to give others, such as personal assistants, access to their accounts for day-to-day activities, when unable to work due to sickness or annual leave. Aside from corporate life, users may need to delegate account access to close friends or relatives, e.g., many banks offer third-party access to support vulnerable users, a service that has seen increasing demand since COVID-19 when (vulnerable) people were not able to attend branches.

Interestingly, with traditional authentication methods such as passwords or OTPs (which WebAuthn aims to replace) delegation can be performed relatively easily, albeit not necessarily securely. For example, delegated access to an account protected by a password can be performed by sharing the latter. However, delegation achieved by sharing passwords is susceptible to reused passwords (see e.g. a study by Pearman et al. [45]), which may inadvertently give access to other accounts, and can only be revoked by changing the account password—which must be done manually. This requires greater trust into the proxy who may change the account password without permission. Additionally, accounts protected by multi-factor authentication (MFA), including using OTPs, may be undermined by sharing OTP secrets or registering multiple something-you-own factors (such as phone numbers) for proxies; also these must be separately and often manually revoked by hand when the proxy's access is to be revoked.

Existing methods. Application-specific delegation may be provided to users of the same service provider in a more secure manner, such as allowing access to a mailbox to another user without sharing passwords, e.g., delegated access in Office 365's Outlook. There are also existing standards that aim to achieve this on the web and in local networks. OAuth [29], analysed by Fett et al. [20], is an open standard for authorisation, or access delegation, through which users can grant applications, especially websites, access to account data without sharing passwords. For example, users can create a Facebook account by granting Facebook access to an existing Google account, sharing their display name, email address and birthday. OpenID Connect (OIDC) [53] is an authentication layer built on top of OAuth which uses OpenID [50], analysed by Recordon and Reed [47], which is a decentralised authentication protocol that allows an *identity provider* (IdP) to share identity data (e.g., name) to a *relying party*—who depends on the identity provider.

The OAuth and OIDC standards provide authorisation and authentication mechanisms on the web, much like Kerberos [39] for authentication and client authorisation. Single-Sign On (SSO), a

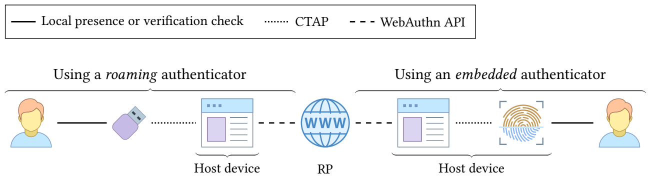

Local presence or verification check
CTAP
WebAuthn API
Using a *roaming* authenticator
Using an *embedded* authenticator
Host device
RP
Host device

Figure 1: Using roaming and embedded authenticators.

federated identity, is often achieved through Security Assertion Markup Language (SAML) [46], analysed by Armando et al. [5] and Groß [26], which uses cookies and provides a standard language for exchanging authentication and authorisation information between IdPs and service providers. Existing identity-providing protocols are discussed and compared by Naik and Jenkins [43].

Additionally, there exist schemes for delegating signing rights in the realm of digital signatures, called proxy signatures, first introduced by Mambo et al. [38]. However, when viewed from the perspective of WebAuthn, existing proxy signatures would not maintain the unlinkability properties of WebAuthn credentials and, in some cases, its decentralised nature. See Section 1.1.1 for more on WebAuthn properties and Section 3.1 for a discussion on existing proxy signature schemes in the context of WebAuthn delegation. For example, access to WebAuthn accounts must be delegatable without disclosing the proxy user's identity to the service provider; otherwise delegated accounts may become linkable.

More recently, Yubico has proposed a protocol for backing up WebAuthn credentials [37] without compromising WebAuthn properties. Their approach is based on a new Asynchronous Remote Key Generation (ARKG) primitive [24], which allows a primary authenticator to create pubic key credentials for one or more trusted backup authenticators, owned by the same user, whilst maintaining the unlinkability and decentralisation properties of WebAuthn—which many of the current approaches do not provide as shown in a recent study by Kunke et al. [36].

We observe that Yubico's approach can be viewed as some form of self-delegation, however, delegating to other users gives rise to further challenges and considerations due to the new and less-trusted security setting, as opposed to the trusted ownership of primary and backup authenticators.

1.1 WebAuthn properties, delegation challenges, and naïve approaches

1.1.1 *Overview of WebAuthn and its properties.*

WebAuthn [35] is a web-based application programming interface (API) that allows web servers, called Relying Parties (RPs), to communicate with conforming authenticators on a user's device—the host device.

As outlined in Figure 1 this communication is facilitated by a WebAuthn client through its implementation of the WebAuthn standard—the RP calls functions provided by the API. RPs may use this to employ asymmetric cryptography to authenticate users on the web, with keying material stored on and managed by the user's authenticator.

Authenticators and clients. WebAuthn authenticators may be software- or platform-based, called *embedded* or *bound* (which are part of the *host* device), such as Windows Hello, or a separate hardware token, called a *roaming* authenticator, for example a YubiKey from Yubico.

These authenticators interact with RPs via a client, such as a web browser, which can communicate with software and hardware on the user's host device, unlike the sandboxed website—the browser bridges this gap.

The Client-to-Authenticator Protocol (CTAP) [21] allows *roaming* hardware tokens to communicate with a client running on a host device, such as a user's phone or computer, over some transport such as Bluetooth, NFC, or USB. The client provides RPs with the ability to communicate with roaming and software authenticators. Note that the older FIDO standard describes CTAP1/U2F and Universal Authentication Framework (UAF), whereas FIDO2 is the newer umbrella specification for W3C's WebAuthn and CTAP (version 2.1) in combination to give passwordless, second- and multi-factor authentication on the web [23].

Registration and authentication. During registration, the authenticator receives a random challenge from the RP. It requires the user to perform a *gesture*, for example, to press an on-screen confirmation button, present a biometric factor, or enter a passcode, in order to unlock the authenticator for use. The authenticator then generates a private-public key pair unique to this registration, submitting this challenge along with the new public key. Later, when authenticating, the authenticator receives another challenge from the RP and signs it with the corresponding private key after the user performs the required gesture.

Gestures provide RPs with a check for either user presence (e.g. press to confirm) or verification (e.g. biometrics), depending on the what is offered by the authenticator. As in Figure 1, the user performs gestures on both hardware tokens and platform authenticators. The ceremonies for registration and authentication are detailed in the standard [35, §§7.1,7.2].

A recent analysis by Barbosa et al. [6] confirms the authentication security of the WebAuthn protocol against active impersonation attacks. In addition, the following properties are commonly associated with the WebAuthn standard, and become important in the context of delegation.

Unlinkability. The unlinkability of registered public keys, required by WebAuthn, prevents users from being correlated across registrations since it cannot be determined whether two keys were produced by the same authenticator.

This is achieved by registering a freshly-generated private-public key pair, giving a unique key pair registered for each account. This property means that users cannot be identified across RPs, unless they willingly reuse login names, e.g., usernames or email addresses.

Attestation. Authenticators may present RPs with certificates from vendors to prove their make and model, and therefore security assurances, so that user verification may be delegated to the authenticator. This means that verification may be performed locally on the authenticator with RPs trusting these local user verification methods where required, since the authenticator attests to this verification with its attestation certificate. This is designed for use in high-security applications [22].

Attestation uses an attestation ID, `AAGUID`, shared between authenticators of similar functionality, makes and model, so as to not break unlinkability. Attestation statements may be verified using the certificate chain and its `AAGUID`.

Since attestation is given as a signature and certificate chain over credentials emitted by an authenticator, credential attestation may be undermined by sharing private keying material between authenticators. Barbosa et al. [6] cover attestation as part of their analysis of WebAuthn and CTAP's authentication security.

Storage. Authenticators may generate fresh key pairs for each registration and record the private key locally to ensure unlinkability. However, some authenticators, such as resource-constrained roaming hardware tokens, may not record their private keys locally.

For example, by using additional credential data that may be sent during registration, private keys may be stored with the RP, encrypted under a symmetric key held by the authenticator. This means that private keys are not recorded by the authenticator and are only learnt during the authentication process, where the authenticator receives these additional credential data as part of its challenge.

1.1.2 Delegation-specific challenges. Any approach for delegating WebAuthn credentials should preserve the unlinkability of WebAuthn credentials, not undermine its decentralised nature, and retain the ability to perform attestation. In addition, it is highly desirable that users are able revoke access for delegated credentials and set permissions which are understood and enforced by RPs. Finally, proxies *need not* hold their own accounts at RPs for delegation. RPs must be unable to link proxies to their delegators, even if the proxy already holds their own account at the same RP.

Delegation must preserve the unlinkability of delegator's original WebAuthn credentials across different accounts. This means that delegated credentials must not contain any information that would allow RPs to link a delegator's accounts and, therefore, the unlinkability guarantees must be extended to cover credentials of proxy users.

In particular, delegated credentials must not reveal any information that would allow RPs to identify the proxy user, who may be in possession of delegated credentials to multiple accounts. This preserves the unlinkability guarantee for delegators and extends it to proxies.

Given the decentralised nature of WebAuthn, where credentials are stored directly with RPs and on user-owned devices, delegation solutions should not require a third party, whose reliability and trustworthiness would need to be relied upon.

Since attestation uses manufacturer certificates to prove the make and model of an authenticator when generating credentials, which may describe security guarantees offered by the authenticator, sharing private keying materials between authenticators may undermine these security guarantees—the trustworthiness of attestation guarantees could be weakened.

However, RPs may offer some flexibility in handling attestation. For example, an RP may accept a credential registered on behalf of another authenticator (i.e., the credential is signed with a private key that does not match the public key being registered), with the provision that the latter authenticator may be rejected when providing its own attestation during authentication, if it does not meet the RP's attestation policy.

Delegators should also be able to revoke account access from proxies at a later time. In addition, the ability to add context- or application-specific permissions that may also include an expiry date, and define the types of actions that can be performed with delegated credentials, should also be considered.

In particular, it must not be possible for a proxy user to take over the ownership of the account by deleting the original delegator's credentials. In order to enforce permissions, RPs must be able to distinguish that account access is being performed by a proxy with a delegated credential and be able to process additional data that identifies the credential as being for a proxy.

These requirements rule out a number of initial approaches to delegation. The first naïve approach that maintains unlinkability would include the proxy user generating multiple independent key pairs in advance and sending these to the delegator, who registers them at RPs to perform delegation. This approach is inefficient and unworkable for resource-constrained authenticators which may not store (many) keys locally.

The second naïve approach is the reversed version—where the delegator generates key pairs for proxy users on-demand (e.g., upon performing the delegation) and then sends them the private keys. This approach would undermine the attestation requirement due to the sharing of private keying material between authenticators.

Finally, a third naïve approach could involve a trusted third party to mediate the delegation and potentially provide a cloud-backed key management solution. However, this would depend on the reliability and trustworthiness of the third party, as well as weakening the decentralised nature of WebAuthn and going against the recommendations of the standard [35, §§13.2,13.4.6].

1.2 Contribution and organisation

We propose two approaches for delegation of WebAuthn credentials aiming to preserve the decentralisation, unlinkability and attestation properties of the standard. Our approaches, presented in Section 2, enable the account owner to either configure delegation credentials and permissions *remotely* at the relying party, or to send them *directly* to the proxy user.

Both approaches are compatible and built on top of the recent Asynchronous Remote Key Generation (ARKG) [24] scheme, which has been proposed by Yubico to W3C for WebAuthn backup and account recovery [37]. Our remote delegation approach uses ARKG directly to create unlinkable delegated credentials for proxy users.Our direct delegation approach is performed using warrants and requires a new class of proxy signatures that we call Proxy Signature with Unlinkable Warrants (PSUW) and present in Section 3.

PSUW is used to create warrants that can extend the required unlinkability property to proxy signers. Our PSUW scheme is very efficient and is constructed generically from ARKG. In Section 4, we provide cryptographic implementations that are compatible with WebAuthn, analyse and compare performance of our delegation approaches, discuss in detail various aspects of their intergration with WebAuthn and CTAP, give a prototype of the delegation schemes which uses standard WebAuthn calls, and provide remarks on their usability. We conclude in Section 5.

2 DELEGATING WEBAUTHN ACCOUNT CREDENTIALS

In this section we specify two approaches allowing users to delegate access to their WebAuthn accounts to proxies. Our delegation approaches are compatible with Yubico's recent proposal for backing up WebAuthn credentials and using them for account recovery [37].

2.1 From account recovery to delegation

Yubico's solution for the credential backup and account recovery problem in WebAuthn is based on a novel ARKG protocol [24] which helps to preserve the decentralisation, unlinkability, and attestation properties of the standard. The ARKG protocol allows a *primary* authenticator to register keys on behalf of additional *backup* authenticators, which may be later used to regain access to accounts. After the authenticators have been setup, the primary authenticator generates and registers unlinkable public keys on behalf of backup authenticators upon registering with RPs.

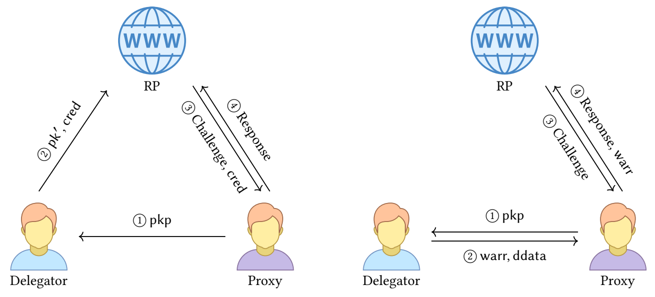
In a nutshell, during the setup phase a backup authenticator with the private key sk sends to the primary authenticator its public key pk. When registering with RPs, *derived* public keys pk′ are generated based on the backup's public key pk, computed and registered by the primary authenticator. After the primary authenticator is lost or damaged, a backup authenticator is able to compute the corresponding private key sk′ for the pk′ registered on its behalf using credential data cred stored at the RP, which gives the link between pk and pk′ and allows the corresponding sk′ to be computed with knowledge of sk.

ARKG maintains the unlinkability property of WebAuthn, as arbitrary derived public keys pk′, and pk, exhibit unlinkability that is compatible with WebAuthn. Attestation is also maintained since sk′ is not shared between authenticators and the backup authenticator may provide attestation statements for RPs to verify when regaining access to accounts, i.e., when completing their authentication challenge.

Yubico's credential backup solution assumes that the primary and backup authenticators are owned and controlled by the same user and, once setup, the primary authenticator can invisibily register or re-register keys for backup authenticators when needed. This approach and the underlying trust assumptions do not readily translate to delegation, as delegators should be able to delegate, as well as grant and revoke permissions, to proxies at their discretion. RPs need to enforce limited access to accounts, requiring that proxy users cannot lock delegators out of their own accounts or perform actions for which they do not have permission. This results in delegation which requires additional data and parameters, so RPs can determine the level of access to grant to proxy users; ARKG without any changes grants full account access, since it is roughly equivalent to a standard account credential.

2.2 Two approaches for delegation

In our first approach, called *remote delegation*, delegation is configured remotely on the relying party, whereas in the second approach, called *direct delegation*, delegated credentials are sent directly to the proxy. Our approaches address the requirements from Section 1.1.2 and cater for different types of authenticators, and their capabilities, such as storage-constrained hardware tokens—the remote variant



(a) Remote delegation. **(b) Direct delegation.**

Figure 2: Overview of WebAuthn credential delegation variants. The remote variant uses RPs to store and manage delegated credentials, whereas the direct variant uses warrants sent to proxies.

is best suited to this kind, whereas direct delegation is suited for platform authenticators in computers and mobile devices. Neither approach requires proxies to hold an account at the RP in order to complete delegation.

We aim to use ARKG, which is currently under consideration for standardisation by W3C's WebAuthn working group, as a building block for both delegation approaches so that they can be realised with minimal changes for RPs and authenticators that may already support ARKG for credential backup.

Both approaches share the same setup phase where the delegator communicates with the proxy. Following the setup phase, delegation can be performed for different accounts at one or more relying parties. In the remote approach, the delegator deposits a public key and some additional data intended for the proxy user with the RP, i.e., delegation is configured and achieved through the RP. The proxy user will be able to prove its authorisation by communicating with the RP at a later stage. In the direct approach, the delegator sends a warrant with some additional data to the proxy user who may later prove to the RP that they are authorised to access the delegator's account. Both variants are depicted in Figure 2 and detailed in Sections 2.3 to 2.5.

2.3 Setup phase (common for remote and direct delegation)

This phase is the same for both delegation approaches and corresponds to step ① where the delegator learns a public key pkp of the proxy user. This public key will enable the generation of multiple delegated credentials, potentially for different accounts owned by the delegator. The delegator already has an independent private-public key pair (skd, pkd), registered via WebAuthn, for each account it holds—with which it can delegate account access.

The proxy user must know the corresponding private key skp in order to sign future WebAuthn challenges when accessing the delegator's account. To access the delegator's account, proxies will need to know that delegation has been granted along with the login name used for the delegator's account. Conceptually, the setup phase is similar to that of ARKG [24] when viewing a proxy user's pkp as a backup authenticator's pk. The main difference is in the channel through which pkp is transmitted given the differences in the ownership of the authenticators, as discussed in more detail in Section 4.

2.4 Remote delegation

In the remote variant (see Figure 2a), to perform delegation in step ②, the account owner logs into their account at the RP using WebAuthn credentials and configures delegation for the proxy user by generating an unlinkable public key pk′ from the proxy user's public key pkp using ARKG. In addition to pk′, the delegator stores at the RP the corresponding ARKG credential information cred. In order to access the account in steps ③ and ④, the proxy user will use ARKG to derive the corresponding signing key sk′ from its private key skp and cred, and sign on the RP's authentication challenge. Remote delegation can be performed for multiple accounts following a single setup phase since many unlinkable pk′ may be generated for the same proxy's pkp.

Note that all derived keys pk′ are unrelated to the delegator's pkd and they are registered directly with the RP for the account, maintaining WebAuthn unlinkability and providing a decentralised design—as credentials are stored with only the RP and authenticator involved, with no third party.

Since the RP records proxy credentials, regardless of whether they have ever been used, credentials registered for proxies may be deleted, i.e., revoked, on an as-needed basis. As the RP is always aware of delegations as soon as they are registered, it may provide a web interface allowing the delegator to set permissions against individual account credentials, including those for proxies. This meets the revocation and policy control requirement, as well as ensuring the RPs are aware that this credential is for a proxy user.

2.5 Direct delegation

Our direct delegation variant (see Figure 2b) uses a new proxy signature scheme that involves a common delegation-by-warrant approach, but outputs warrants that remain unlinkable with respect to the proxy users. We also show how to construct such a proxy signature scheme using ARKG as a building block (see Section 3.3).

To delegate access to the account, for which the delegator's public key pkd is registered with an RP, the delegator uses its private key skd to create a warrant warr for the proxy user for whom pkp was received as part of the setup phase. This warrant contains the delegator's signature on a warrant public key pkw which is different from and unlinkable to pkp. The delegator sends warr together with some delegation data ddata to the proxy user in step ②. The proxy user can compute the warrant signing key skw using received ddata and is able to use skw to sign RP's authentication challenges when accessing the delegator's account in steps ③ and ④. The RP checks validity of the warrant using delegator's registered pkd prior to granting account access.

Observe that multiple warrants containing different pkw can be generated by the delegator for the same proxy user after the initial setup phase. These pkw remain unlinkable. In this way, the delegator can repeatedly grant access to the same proxy user for one or more of its WebAuthn accounts for the same or different RPs. The communication channel required for step ② can be established on-demand, in a similar way as during the setup phase (see Section 4 for more details).

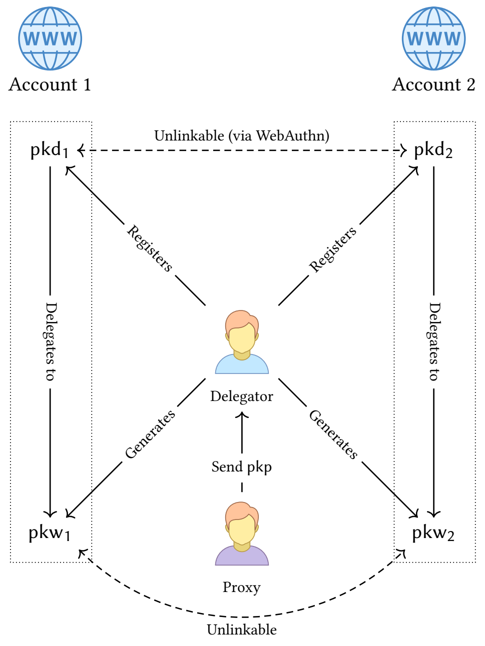


Figure 3: Unlinkability of delegated WebAuthn credentials. pkw **is a warrant public key specific to each delegation.**

Note that since RPs are unaware of direct delegations until they are presented with warrants by proxy signers, warrants must be validated against the delegator's existing WebAuthn account credentials, that is using the delegator's public key pkd for the account, to prove that it was issued by the account owner. This requires that warrants, the public key pkw to which access is granted, and the existing account credential be linked. However, warrants must not undermine the unlinkability of existing account owner's credentials, whilst extending the unlinkability property to their proxies—resulting in registrations and delegations remaining uncorrelatable across accounts and RPs, as visualised in Figure 3. The delegator's account public key pkd is unlinkable to delegator's public keys used on other accounts by virtue of them being registered as normal with WebAuthn. For proxy signers, however, this means that their warrant public keys pkw created by the delegator must also remain unlinkable. This essentially motivates the need for the new type of proxy signatures, which we call PSUW and detail in Section 3, through which delegators can issue warrants that cannot be linked to a proxy signer's pkp. Our direct approach therefore does not require a third party to complete the delegation, maintaining the decentralisation of WebAuthn.

Any issued warrants may be invalidated by replacing the account credential for pkd, which may be seen as equivalent to changing the password given to someone to access accounts. Alternatively, RPs could allow users to blacklist warrants, which would require additional server-side logic and delegators to record generated warrants to later add them to blacklists. The warrants used in the direct variant may contain additional signed data, including expiry timestamps and permissions granted to the proxy. RPs would need to understand the warrant permissions format—this offers the benefit of being signed by the delegator by default, in a well-known and easily-parsed warrant format (see also Section 4 for more details on revocation and permissions).

3 PROXY SIGNATURE WITH UNLINKABLE WARRANTS

In this section we present a new scheme called Proxy Signature with Unlinkable Warrants (PSUW), a new type of proxy signature required for the direct delegation of WebAuthn credentials. PSUW adopts the delegation-by-warrant approach however, in contrast to other proxy signature schemes, delegated warrants and signatures produced by proxies remain unlinkable to the identities of the proxies. This helps to protect the unlinkability of multiple WebAuthn accounts access to which has been delegated to the same or multiple proxies. Following a brief comparison with related work on proxy signatures, we model security of PSUW and show how to construct it generically using ARKG and an ordinary digital signature scheme.

3.1 Related work on proxy signatures

Proxy signatures were introduced by Mambo et al. [38], with security formalised by Boldyreva et al. [8]. Typically, delegation to a proxy signer is performed by issuing a warrant that contains a certificate on the proxy signer's public key that verifies against the delegator's public key. This warrant then becomes part of the proxy signature, and is verified within the proxy signature's verification. Any scheme that follows this approach cannot achieve unlinkable warrants since the proxy signer's public key is required to verify the warrant, which make them unsuitable for application in WebAuthn. Nonetheless, a range of constructions from different hardness assumptions exist, e.g., based on discrete logarithms [8, 31, 38, 55], integer factorisation [57], and lattice-based problems [30].

In addition to standard proxy signatures there have been proposals to provide various degrees of privacy for the warrants, such as anonymous proxy signatures by Shum and Wei [52], with a formal model and a general construction proposed later by Fuchsbauer and Pointcheval [25]—giving the properties of proxy anonymity and traceability. Their model assumes three parties, the issuer, user, and opener. The construction bears similarity with the approach taken in group signatures Chaum and van Heyst [13] through encryption of the warrant under the opener's public key with appropriate zero-knowledge proofs, such that the proxy signer remains anonymous yet traceable by the opener if needed. We observe that although this scheme offers richer functionality than ours, such functionality is not needed in the context of WebAuthn. Removal of the traceability requirement from [25] would still yield a complex construction requiring zero-knowledge proofs to protect privacy and hence would not be compatible with the WebAuthn standard. Using central authorities would also not suit its decentralised nature.

We also note the work of Yu et al. [58], who also call their scheme 'anonymous proxy signature'. However, their construction has different functionality and is a combination of a proxy signature and a ring signature [48], the term 'anonymous' comes from the fact that a proxy is hidden within a larger set of proxies when verifying the ring signature. Moreover, their security analysis is focused only on the unforgeability properties. Similarly, the scheme proposed by Wu et al. [56] combines standard proxy signatures with group signatures, and is similarly not applicable for our setting.

Finally, we note the work by Derler et al. [16], which contructs warrant-hiding proxy signatures (see also [28]) and blank signatures (see also [27]) from anonymous credentials [12]. In the former, warrant-hiding means that the message space delegated to a proxy remains unknown to the verifier, other than the message being presented for verification. In the latter, a format for the message has a structure that can be changed in a prescribed way. These techniques are not what is required to realise unlinkable delegation in WebAuthn.

3.2 Modelling PSUW

We define here the syntax and security properties of PSUW.

3.2.1 Syntax of PSUW. The scheme has six algorithms. *Public parameters* pp *are implicitly given as input to all algorithms.*

Definition 1 (PSUW). *A Proxy Signature with Unlinkable Warrants scheme consists of the following six algorithms:*

- $\mathsf{Setup}(\lambda)$ *generates and outputs public parameters* pp *of the scheme for security parameter* $\lambda \in \mathbb{N}$.
- $\mathsf{DKGen}()$ *samples a private-public key pair* $(\mathsf{skd}, \mathsf{pkd})$ *for a delegator when called.*
- $\mathsf{PKGen}()$ *samples a private-public key pair* $(\mathsf{skp}, \mathsf{pkp})$ *for a proxy signer when called.*
- $\mathsf{Delegate}(\mathsf{skd}, \mathsf{pkp})$ *takes as input* skd *and* pkp. *It probabilistically returns warrant* warr *and delegation data* ddata, *which the proxy signer may use with its own signing key* skp *to generate proxy signatures.*
- $\mathsf{Sign}(\mathsf{skp}, \mathsf{pkd}, \mathsf{warr}, \mathsf{ddata}, m)$ *takes as input* skp, pkd, warr, ddata, *message m, and returns a proxy signature* $\bar{\sigma}$ *under* skp *for m, or* $\perp$ *on error (e.g.* warr *or* ddata *are invalid for* pkd*). Note that* $\mathsf{warr} \in \bar{\sigma}$, *allowing* $\bar{\sigma}$ *to be verifiable as a standalone signature.* ddata *is used to compute the signing key, but is not included in the proxy signature* $\bar{\sigma}$ *output.*
- $\mathsf{Verify}(\mathsf{pkd}, \bar{\sigma}, m)$ *takes as input* pkd, $\bar{\sigma}$, *m, and returns 1 if* $\bar{\sigma}$ *is valid with respect to* pkd *for m, otherwise 0.*

3.2.2 Security definitions. We give the adversarial model and define two properties for PSUW: Warrant-Unlinkability and Unforgeability.

Adversaries and oracles. We model an adversary $\mathcal{A}$ as a probabilistic polynomial time (PPT) algorithm, which may call, using the parameters to which it is given access, any of the public procedures given in Section 3.2.1.

The adversary $\mathcal{A}$ may make a polynomial number of queries to the oracles given in Figure 4:

- $\mathcal{O}_{\mathsf{Reg}}$, when called, samples a new key pair (skp, pkp), storing the key pair in list LReg, and returns pkp.
- $\mathcal{O}_{\mathsf{Delegate}}(\mathsf{pkp})$, is initialised with the delegator's private key skd. On input proxy-signer public key pkp, it records the result of Delegate(skd, pkp) in list LDel, returning warr and ddata without giving $\mathcal{A}$ access to skd. If $(\cdot, \mathsf{pkp}) \notin$ LReg, it aborts. This models an honest delegation to pkp.

```
$\mathcal{O}_{\mathsf{Reg}}$ when called
1 : $(\mathsf{skp}, \mathsf{pkp}) \leftarrow\$ \mathsf{PKGen}()$
2 : $\mathsf{LReg} \leftarrow \mathsf{LReg} \cup (\mathsf{skp}, \mathsf{pkp})$
3 : return $\mathsf{pkp}$

$\mathcal{O}_{\mathsf{Delegate}}$ on input $\mathsf{pkp}$
1 : if $(\cdot, \mathsf{pkp}) \notin \mathsf{LReg}$ then abort
2 : $(\mathsf{warr}, \mathsf{ddata}) \leftarrow \mathsf{Delegate}(\mathsf{skd}, \mathsf{pkp})$
3 : $\mathsf{LDel} \leftarrow \mathsf{LDel} \cup (\mathsf{pkp}, \mathsf{warr})$
4 : return $\mathsf{warr}, \mathsf{ddata}$

$\mathcal{O}_{\mathsf{Corr}}$ on input $\mathsf{pkp}$
1 : retrieve $(\mathsf{skp}, \mathsf{pkp})$ from $\mathsf{LReg}$ else abort
2 : $\mathsf{LCorrupt} \leftarrow \mathsf{LCorrupt} \cup (\mathsf{skp}, \mathsf{pkp})$
3 : return $\mathsf{skp}$

$\mathcal{O}_{\mathsf{Sign}}$ on input $(\mathsf{pkp}, \mathsf{warr}, \mathsf{ddata}, m)$
1 : retrieve $(\mathsf{skp}, \mathsf{pkp})$ from $\mathsf{LReg}$ else abort
2 : $\bar{\sigma} \leftarrow \mathsf{Sign}(\mathsf{skp}, \mathsf{pkd}, \mathsf{warr}, \mathsf{ddata}, m)$
3 : if $\bar{\sigma} \stackrel{?}{=} \perp$ then abort
4 : $\mathsf{LSign} \leftarrow \mathsf{LSign} \cup (\bar{\sigma}, \mathsf{pkp}, \mathsf{warr}, \mathsf{ddata}, m)$
5 : return $\bar{\sigma}$
```

Figure 4: Oracles for PSUW security experiments.

- $\mathcal{O}_{\mathsf{Sign}}(\mathsf{pkp}, \mathsf{warr}, \mathsf{ddata}, m)$, is intialised with the delegator's public key pkd. On input proxy-signer public key pkp, warrant warr, delegation ddata, and a message m, it records in list LSign the result of calling $\mathsf{Sign}(\mathsf{skp}, \mathsf{warr}, \mathsf{ddata}, m)$ for $(\mathsf{skp}, \mathsf{pkp}) \in \mathsf{LReg}$. It aborts if Sign returns $\perp$. This models asking for a signature on a message of $\mathcal{A}$'s choice.
- $\mathcal{O}_{\mathsf{Corr}}(\mathsf{pkp})$, on input proxy-signer public key pkp, returns the corresponding private key skp from list LReg and adds the key to list LCorrupt. If $(\cdot, \mathsf{pkp}) \notin \mathsf{LReg}$, it aborts. This models the leak of private keying material for proxy.

Correctness. Our scheme is correct if, $\forall \lambda \in \mathbb{N}$, the following hold:

$$\begin{aligned} \mathsf{pp} &\leftarrow \mathsf{Setup}(\lambda) \\ (\mathsf{skd}, \mathsf{pkd}) &\leftarrow \mathsf{DKGen}() \\ (\mathsf{skp}, \mathsf{pkp}) &\leftarrow \mathsf{PKGen}() \\ (\mathsf{warr}, \mathsf{ddata}) &\leftarrow \mathsf{Delegate}(\mathsf{skd}, \mathsf{pkp}) \\ \bar{\sigma} &\leftarrow \mathsf{Sign}(\mathsf{skp}, \mathsf{pkd}, \mathsf{warr}, \mathsf{ddata}, m) \\ 1 &\stackrel{?}{=} \mathsf{Verify}(\mathsf{pkd}, \bar{\sigma}, m) \end{aligned}$$

Warrant-Unlinkability. The Warrant-Unlinkability (wu) property of a PSUW scheme ensures that an adversary, when given proxy signature $\bar{\sigma}$, warrant warr, and delegation data ddata, cannot determine the identity pkp for which the delegation was performed.

The unlinkability property of WebAuthn requires that users cannot be correlated across registrations. We capture this property by requiring that an entity, such as an RP in the case of WebAuthn, be able to determine the identity of the user who delegated signing rights, i.e., account access, but not the identity of the proxy signer. This means that unlinkability is maintained for both the delegator—where knowing the identity of proxy signers might provide linkability for account holders too—and the proxy signer.

We model Warrant-Unlinkability in Figure 5a. The experiment $\mathsf{Exp}^{\mathsf{wu}\text{-}b}_{\mathcal{A}}(\lambda)$ is parameterised with bit b. It chooses a delegator's key pair $(\mathsf{skd}, \mathsf{pkd})$ and challenges an adversary $\mathcal{A}$ to determine which of its proxy-signing keys was used to delegate signing rights for pkd and sign message m. $\mathcal{A}$ is given access to oracles $\mathcal{O}_{\mathsf{Reg}}$, $\mathcal{O}_{\mathsf{Delegate}}$, $\mathcal{O}_{\mathsf{Sign}}$ and $\mathcal{O}_{\mathsf{Corr}}$.

Definition 2 (Warrant-Unlinkability). *This is offered by PSUW if the following is negligible in λ:*

$$\mathsf{Adv}^{\mathsf{wu}\text{-}b}_{\mathsf{PSUW},\mathcal{A}}(\lambda) := \left|\Pr\left[\mathsf{Exp}^{\mathsf{wu}\text{-}1}_{\mathcal{A}}(\lambda) = 1\right] - \Pr\left[\mathsf{Exp}^{\mathsf{wu}\text{-}0}_{\mathcal{A}}(\lambda) = 1\right]\right|$$

Unforgeability. For a PSUW scheme to satisfy the Unforgeability property, an adversary $\mathcal{A}$ must not be able to forge proxy signatures with respect to a delegator's pkd without knowledge of the corresponding skd, as modelled in Figure 5b.

The experiment challenges $\mathcal{A}$ to give a valid proxy signature $\bar{\sigma}$, for a message m of its choice, for a delegator's pkd. The adversary is given access to the $\mathcal{O}_{\mathsf{Reg}}$, $\mathcal{O}_{\mathsf{Delegate}}$, $\mathcal{O}_{\mathsf{Sign}}$, and $\mathcal{O}_{\mathsf{Corr}}$ oracles. $\mathcal{A}$ wins if it can break either the delegation (line 4) or signing procedures (lines 5 and 6).

Definition 3 (Unforgeability). *A PSUW scheme provides Unforgeability if the following advantage is negligible in λ:*

$$\mathsf{Adv}^{\mathsf{unforge}}_{\mathsf{PSUW},\mathcal{A}}(\lambda) := \Pr\left[\mathsf{Exp}^{\mathsf{unforge}}_{\mathcal{A}}(\lambda) = 1\right]$$

3.3 Our generic PSUW construction

We proceed with a generic construction of PSUW, based on the ARKG primitive and an ordinary digital signature scheme, and then analyse its security.

3.3.1 *Building blocks.* We recall the two building blocks.

Asynchronous Remote Key Generation (ARKG) [24]. An ARKG scheme has five algorithms, $\mathsf{ARKG} := (\mathsf{Setup}, \mathsf{KGen}, \mathsf{DerivePK}, \mathsf{DeriveSK}, \mathsf{Check})$. $\mathsf{Setup}(\lambda)$ returns public parameters pp for the scheme. KGen samples a key pair $(\mathsf{sk}, \mathsf{pk})$ when called. Derived public key algorithm $\mathsf{DerivePK}(\mathsf{pk}, \mathsf{aux})$ returns a new public key pk' and derivation data cred. Derived private key sk' for pk' is computed and returned by $\mathsf{DeriveSK}(\mathsf{sk}, \mathsf{cred})$. $\mathsf{Check}(\mathsf{sk}', \mathsf{pk}')$ returns 1 if $(\mathsf{sk}', \mathsf{pk}')$ form a valid key pair, otherwise 0. In our construction, we use ARKG which offers the following two properties:

PK-Unlinkability (pku) is provided by ARKG if $\mathsf{Adv}^{\mathsf{pku}}_{\mathsf{ARKG},\mathcal{A}}(\lambda)$ is negligible in λ for a PPT adversary $\mathcal{A}$ to distinguish between derived public keys and uniformly-sampled public keys. The adversary is given access to challenge oracle $\mathcal{O}^{b}_{\mathsf{pkp}}$ which is initialised with a bit b and public key pk. When called, it returns either the

```
$\mathsf{Exp}^{\mathsf{wu}\text{-}b}_{\mathcal{A}}(\lambda)$
1: $\mathsf{pp} \leftarrow \mathsf{Setup}(\lambda)$
2: $(\mathsf{skd}, \mathsf{pkd}) \leftarrow\!\$\ \mathsf{DKGen}()$
3: $(\mathsf{st}, \mathsf{pkp}_0, \mathsf{pkp}_1, m) \leftarrow \mathcal{A}_1^{O_{\mathsf{Reg}}, O_{\mathsf{Delegate}}, O_{\mathsf{Sign}}, O_{\mathsf{Corr}}}(\mathsf{pp}, \mathsf{pkd})$
4: $\textbf{retrieve } (\mathsf{skp}_0, \mathsf{pkp}_0) \text{ and } (\mathsf{skp}_1, \mathsf{pkp}_1) \text{ from } \mathsf{LReg}\ \textbf{else}$
       $\textbf{return } 0$
5: $(\mathsf{warr}, \mathsf{ddata}) \leftarrow \mathsf{Delegate}(\mathsf{skd}, \mathsf{pkp}_b)$
6: $\bar{\sigma} \leftarrow \mathsf{Sign}(\mathsf{skp}_b, \mathsf{pkd}, \mathsf{warr}, \mathsf{ddata}, m)$
7: $b' \leftarrow \mathcal{A}_2^{O_{\mathsf{Reg}}, O_{\mathsf{Delegate}}, O_{\mathsf{Sign}}, O_{\mathsf{Corr}}}(\mathsf{st}, \bar{\sigma}, \mathsf{warr}, \mathsf{ddata})$
8: $\textbf{return } b \overset{?}{=} b' \wedge (\cdot, \mathsf{pkp}_0) \notin \mathsf{LCorrupt} \wedge (\cdot, \mathsf{pkp}_1) \notin \mathsf{LCorrupt}$
```

(a) Warrant-Unlinkability experiment.

```
$\mathsf{Exp}^{\mathsf{unforge}}_{\mathcal{A}}(\lambda)$
1: $\mathsf{pp} \leftarrow \mathsf{Setup}(\lambda)$
2: $(\mathsf{skd}, \mathsf{pkd}) \leftarrow\!\$\ \mathsf{DKGen}()$
3: $(\bar{\sigma}, m) \leftarrow \mathcal{A}^{O_{\mathsf{Reg}}, O_{\mathsf{Delegate}}, O_{\mathsf{Sign}}, O_{\mathsf{Corr}}}(\mathsf{pp}, \mathsf{pkd})$
4: $\textbf{parse } \bar{\sigma} \text{ as } (\mathsf{warr}, \cdot)$
5: $\textbf{return } \mathsf{Verify}(\mathsf{pkd}, \bar{\sigma}, m) = 1$
6:         $\wedge\ ((\cdot, \mathsf{warr}) \notin \mathsf{LDel}\ \vee$
7:             $[\exists \mathsf{pkp}, m \text{ s.t. } (\cdot, \cdot, \mathsf{warr}, \cdot, m) \notin \mathsf{LSign}\ \wedge$
8:              $(\mathsf{pkp}, \mathsf{warr}) \in \mathsf{LDel} \wedge (\cdot, \mathsf{pkp}) \notin \mathsf{LCorrupt}])$
```

(b) Unforgeability experiment.

Figure 5: Security experiments for PSUW.

```
$\mathsf{PSUW.Setup}(1^\lambda)$
1: $\textbf{return } \mathsf{pp} = (1^\lambda, \mathsf{ARKG.Setup}(1^\lambda), \mathsf{DS.Setup}(1^\lambda))$

$\mathsf{PSUW.DKGen}(\mathsf{pp})$
1: $\textbf{return } (\mathsf{skd}, \mathsf{pkd}) = \mathsf{DS.KGen}(\mathsf{pp})$

$\mathsf{PSUW.PKGen}(\mathsf{pp})$
1: $\textbf{return } (\mathsf{skp}, \mathsf{pkp}) = \mathsf{ARKG.KGen}(\mathsf{pp})$

$\mathsf{PSUW.Delegate}(\mathsf{skd}, \mathsf{pkp})$
1: $(\mathsf{pkw}, \mathsf{cred}) \leftarrow\!\$\ \mathsf{ARKG.DerivePK}(\mathsf{pp}, \mathsf{pkp}, \emptyset)$
2: $\sigma \leftarrow \mathsf{DS.Sign}(\mathsf{skd}, \mathsf{pkw})$
3: $\textbf{return } \mathsf{warr} = (\mathsf{pkw}, \sigma), \mathsf{ddata} = \mathsf{cred}$
```

```
$\mathsf{PSUW.Sign}(\mathsf{skp}, \mathsf{pkd}, \mathsf{warr}, \mathsf{ddata}, m)$
1: $\textbf{parse } \mathsf{warr} \text{ as } \mathsf{pkw}, \sigma$
2: $\textbf{parse } \mathsf{ddata} \text{ as } \mathsf{cred}$
3: $\textbf{if } \mathsf{DS.Verify}(\mathsf{pkd}, \mathsf{pkw}, \sigma) \overset{?}{=} 0\ \textbf{ then return } \bot$
4: $\mathsf{skw} \leftarrow \mathsf{ARKG.DeriveSK}(\mathsf{pp}, \mathsf{skp}, \mathsf{cred})$
5: $\textbf{if } \mathsf{skw} \overset{?}{=} \bot \textbf{ then return } \bot$
6: $s \leftarrow \mathsf{DS.Sign}(\mathsf{skw}, m)$
7: $\textbf{return } \bar{\sigma} = (s, \mathsf{warr})$

$\mathsf{PSUW.Verify}(\mathsf{pkd}, \bar{\sigma}, m)$
1: $\textbf{parse } \bar{\sigma} \text{ as } s, \mathsf{warr}$
2: $\textbf{parse } \mathsf{warr} \text{ as } \mathsf{pkw}, \sigma$
3: $\textbf{return } \mathsf{DS.Verify}(\mathsf{pkd}, \sigma, \mathsf{pkw}) \wedge \mathsf{DS.Verify}(\mathsf{pkw}, s, m)$
```

Figure 6: Algorithms of our PSUW construction.

result of DerivePK(pp, pk, aux) when $b = 0$ or samples and returns a new public key from $\mathcal{D}$ when $b = 1$.

Malicious-Strong Key Secrecy (msKS) is provided by ARKG if $\mathsf{Adv}^{\mathsf{msKS}}_{\mathsf{ARKG},\mathcal{A}}(\lambda)$ is negligible in λ for a PPT adversary $\mathcal{A}$ to derive a valid key pair $\mathsf{sk}^\star, \mathsf{pk}^\star$ and corresponding $\mathsf{cred}^\star$ for an initial public key pk. It is given access to derived public key oracle $O_{\mathsf{pk}'}$ and derived private key oracle $O_{\mathsf{sk}'}$. It wins if $\mathsf{sk}^\star, \mathsf{pk}^\star$ and corresponding $\mathsf{cred}^\star$ verify against ARKG.Check and it did not trivially obtain these by querying the oracles.

For formal definitions of these properties, we refer to the work of Frymann et al. [24, §4.1]. Note that their construction of ARKG satisfies the above properties under the well-known snPRF-ODH [10] and Discrete Logarithm hardness assumptions in the random oracle model. This construction will be used to instantiate our generic PSUW scheme and is recalled in Appendix A.

*Digital Signature (*DS*).* A digital signature scheme has three algorithms, DS := (DS.KGen, DS.Sign, DS.Verify). The key generation algorithm DS.KGen takes as input a security parameter λ and outputs a key pair (sk, pk). The signing algorithm takes as input a signing key sk with a message m and outputs a signature σ. DS.Verify takes as input a candidate tuple (pk, σ, m) and outputs 1 if σ verifies with respect to public key pk and message m, otherwise 0.

Two variants of DS unforgeability are required. We first require standard existential unforgeability under chosen-message attack (EUF-CMA), which challenges an adversary to produce a forgery $(\sigma^\star, m^\star)$ that verifies with respect to pk without knowledge of the corresponding sk. In this experiment, the adversary has access to a signing oracle O_{Sign} and wins if $\sigma^\star$ was produced on $m^\star$ that was not queried to O_{Sign}. We also require strong unforgeability under chosen-message attack (SUF-CMA) [4], in which case $\sigma^\star$ was not obtained from O_{Sign} on query $m^\star$.

3.3.2 *PSUW algorithms.*

The algorithms of our generic PSUW construction are specified in Figure 6, which use the algorithms from ARKG and DS as underlying building blocks.

3.3.3 *Security analysis of the generic PSUW scheme.*

Theorem 1. *PSUW satisfies Warrant-Unlinkability if ARKG satisfies PK-Unlinkability.*

Theorem 2. *PSUW satisfies Unforgeability if DS is SUF-CMA secure and ARKG offers both Malicious-Strong Key Secrecy and PK-Unlinkability.*

Table 1: Mean execution time for a single primitive call, in milliseconds and averaged from 1000 timings.

Primitive	Delegate	Sign	Verify
ECDSA (plain WebAuthn)	–	1.9	1.6
ARKG (remote delegation)	5.6	3.8	1.6
PSUW (direct delegation)	7.5	5.4	3.2

Proof. Proofs for Theorems 1 and 2 are given in Appendix B. □

Remark 1. *Looking ahead, we will instantiate DS with ECDSA, which is known to only provide EUF-CMA security. However, it has been shown that the* only *attack on strong unforgeability [19] for an ECDSA signature of the form $(s, t) \in \mathbb{G}$, is the forgery $(-s, t) \in \mathbb{G}$.*

As noted by Fersch in [18, Remark 3.2.3], there are numerous techniques to mitigate such an attack by normalising the s component so that only one of s or $-s$ can be verified. For example, enforcing $s \in [1, (q-1)/2]$, so that either s or $-s = q - s$ verifies. In our instantiation and implementation, we enforce this s-component normalisation to acheive a strongly-unforgeable ECDSA.

4 ACHIEVING DELEGATION IN WEBAUTHN

In this section, we discuss instantiations and perfomance of our ARKG-based delegation approaches and the new PSUW primitive, and provide detailed technical and practical considerations for implementing and integrating delegation functionality into WebAuthn, as well as discussing our prototype.

4.1 Cryptographic implementation

We instantiate the cryptographic building blocks in our delegation approaches using compatible, standard-based, and efficient algorithms that are already being used in the WebAuthn standard. This includes ECDSA standard [44] on the P-256 curve (see ECC [11]), which is used for the DS building block within PSUW and to perform all signing operations used in our delegation approaches. Using ECDSA gives further straightforward compatibility with ARKG and PSUW. Credential public keys [35, §6.5.1] in WebAuthn are described using the COSE [51] format. ECDSA on P-256 curve with SHA-256 is given a standard and registered COSE algorithm name EC256 and type `-7`. The COSE algorithm is specified when producing WebAuthn credentials.

The instantiation and implementation of ARKG uses the original proposal from Yubico [24, 37], which is currently being considered for standardisation of WebAuthn credential backups. It adopts the standards-based HKDF [33] with SHA-256 [17] and HMAC-SHA-256 [32] algorithms, which are widely supported in the WebAuthn ecosystem. We adopt the same algorithms to implement PSUW (see our code [2]). In this way, we ensure that our delegation approaches would work well with authenticators that implement ARKG primitive for backup purposes.

Performance. In Table 1, the performance for each of the primitives required for delegating and authenticating in WebAuthn is presented. The timings were taken using our benchmarking program and PSUW implementation, with Python source code available [2]. The existing ARKG code by Lundberg and Nilsson [37] and Python's `fastecdsa` are used as our ARKG and DS building blocks, respectively. These timings are abstracted from the full delegation, registration, and authentication procedures as these encounter unavoidable, and in some cases unpredictable, overheads, including packing data into message formats for WebAuthn processing and network performance. We capture instead the measurable difference between an example of plain WebAuthn, using ECDSA, and our two delegation approaches: remote using ARKG and direct using PSUW. Each primitive was invoked 1000 times on an Intel i7-8700 (3.20 GHz), using a single-threaded software implementation, with the average for a single call recorded.

Compared to a plain WebAuthn sign-and-verify challenge using ECDSA, the ARKG primitive in the remote delegation approach gives an increase of only 1.9 ms in execution time for the signature generation. From ARKG to PSUW used in the direct delegation, we observe an increase of 1.6 ms for the signing operation. PSUW also incurs an average increase of 1.6 ms in verification over ECDSA and ARKG, as it must verify both the warrant's signature and the signature on the challenge. ARKG's verification requires a single ECDSA signature verification as its response is a standard ECDSA signature, but for a derived private-public key pair. Note that these timings are reported without any bespoke optimisations made to the underlying libraries used.

4.2 Approach for integration with WebAuthn

In order to integrate delegation into WebAuthn, we require either a companion program or a client extension that implements both CTAP with changes listed here and the standard WebAuthn API—no changes are required for WebAuthn as we employ its extensions provision. This program will facilitate communications between the authenticators for the setup phase and direct delegation.

4.2.1 *Using and extending CTAP.*

As authenticators cannot communicate directly, a companion program or client (browser) extension is required to complete the setup and delegation phases between the delegator and proxy. This companion program will use new CTAP [21] calls to communicate with supported authenticators to export and import delegation credentials and data:

`authenticatorExportDelegationKey` Return pkp to be given to a delegator from freshly-sampled (skp, pkp), recording skp.

`authenticatorImportDelegationKey` Record proxy public key pkp and some user-defined identifier `id` for a proxy. This may also include `AAGUID` so the proxy's authenticator could be verified against the RP's attestation policy, if available.

`authenticatorMakeDelegatedCredential` On input `rp`, the RP's domain matched against a stored credential's `rp` field, additional signed data `aux` (e.g., timestamp, permissions) and the user-defined identifier `id` for the proxy, generate delegation data (warr, ddata).

`authenticatorListDelegationKeys` Return list of `id`s and any `AAGUID` for imported delegation keys pkp.

`authenticatorImportWarrant` Record input warr and ddata.

`authenticatorDeleteDelegationKey` Given input `id`, delete pkp corresponding to `id`.

The CTAP `authenticatorMakeCredential` call is used for the remote variant, where extension data instructs the authenticator how to handle the request (i.e., that the credential should be created using a known proxy's key). Since a proxy *may* also have their own account(s) at an RP, a list of `PublicKeyCredentialDescriptors`, given in CTAP's `allowList`, should be supplied by the RP. This allows the proxy's authenticator to locate the *delegated* credential, so it does not inadvertently use the proxy's own credential. The proxy then runs ARKG.DeriveSK for recorded pkp until ARKG.DeriveSK succeeds, so no identifiers are present in ddata.

For remote delegation, the authenticator will supply the RP with $(\mathsf{pk}', \mathsf{cred})$ during RP-initiated delegation, with cred used as the credential's ID. For direct delegation, the companion program will instead send (warr, ddata) to the proxy, which will store this pair.

In order to generate warrants, the private key for a given delegator's account must be stored locally. For the remote variant, encrypted private keys stored remotely may be provided as input to `authenticatorMakeDelegatedCredential`, as the user must first authenticate with the RP before completing the delegation to obtain this encrypted private key. This makes remote delegation better suited for hardware tokens that may not have private keys stored locally nor the space to store warrants for direct delegation.

CTAP and WebAuthn credential extensions. Additional data emitted or used by the authenticator, transferred via CTAP and stored in WebAuthn credentials, is achieved using the credential's 'extensions' field. This is a CBOR-encoded [9] mapping of an extension ID to the extension's data, e.g., the warrant for direct delegation, which is also a map of keys to values.

This field is designed for clients to pass credentials with extension data between authenticators and RPs, allowing for additional functionality between authenticator vendors and RPs without being part of the WebAuthn standard. However, in practice many clients remove extension data for extensions not in their approved list.

4.2.2 Client communication. Both variants require the proxy to send the delegator a public key pkp, and possibly `AAGUID`, over an authenticated channel (step ① in Figures 2a and 2b), also used to send warrants for direct delegation (step ② in Figure 2b). The companion program does this on behalf of authenticators—which cannot communicate directly.

Setting up this direct authenticated channel between the delegator and proxy can be achieved through various means as these channels are *independent* of WebAuthn itself and its credentials. Here we discuss several approaches to discovering users and establishing proxy and delegator communications.

Out of band and messaging platforms. A relatively straightforward approach to send pkp and warrants is to have users employ out-of-band channels, such as email or existing instant messaging applications. Mobile devices could use QR codes to transmit pkp and delegation data, by scanning it in person or remotely when sent through some existing channel. This offers the benefit that users can use existing channels, which may provide mechanisms for at least identifying users, but the security of these channels may not be assured. These platforms may require additional out-of-band authentication channels. In practice, we expect, e.g., email, ideally with DMARC [34], and existing messaging platforms, such as Signal [3]—whose protocol was analysed by Cohn-Gordon et al. [14]—and, for enterprise, Microsoft Teams, to potentially be used to achieve delegation with existing identity services.

Proximity-based physical transports. Physical transports, such as Bluetooth or USB, offer some security by requiring physical proximity between delegator and proxy, and provide some built-in methods to securing the communication channel, e.g., Bluetooth's numeric comparison for LE Secure Connections [7]—examined by Zhang et al. [59]. The offline nature of transports like Bluetooth lends itself to our direct delegation approach. Depending on the transport used, the built-in security functionalities offered *may* be sufficient, but these offer limited flexibility or cross-platform support, e.g., not all desktop computers have Bluetooth or NFC.

Third parties and federated identity providers. Using a third party, or federated identity provider, delegators can establish an end-to-end encrypted channel with proxy users. These identities are used by the delegator and proxy to authenticate one another, but are not used at all in WebAuthn as this would break unlinkability. These identity providers mediate connections between clients and therefore must be trusted to maintain privacy and to help establish authenticated secure channels between clients.

4.2.3 Remote delegation. In the ARKG-based credential backup proposal [37], the credential's ID is used to store data, using which the backup authenticator can compute the private key and sign authentication challenges. This approach alone is not suitable for delegation as the RP must enforce permissions. However, it may still be used to embed credential data for the proxy. The authenticator must add to the credential extensions map an extension ID for 'delegation'—so the RP knows that this is a delegated credential, regardless of how its creation was initiated (e.g., via the RP's interface or the client's handling of the WebAuthn call).

Since RPs store delegated credentials as soon as they are created (step ② in Figure 2a), revocation and other permissions for delegated credentials can be realised easily by configuring delegated credentials directly with the RP. The RP may implement a web interface, and corresponding backend logic, allowing the account owner to set such permissions for submitted delegated credentials or revoke them later by deleting these credentials. This requires no changes to WebAuthn, but does make use of its extension field.

Currently, there is no way to have the client initiate a registration ceremony in the WebAuthn standard. The RP decides when to initiate the ceremony when, e.g., the user clicks a button to create an account or add a credential on their account page. In order to complete the delegation, in the case where RPs provide a button to add additional credentials, the client may ask the user to specify whether they want to register a delegated credential, and, if so, to whom. The companion program must therefore show this option as part of its handling of `navigator.credentials.create()` calls, particularly as it may need to call and display locally the delegator-defined proxy IDs, obtained using `authenticatorListDelegationKeys`, in order to allow the delegator to choose to which proxy it will delegate.

Remote delegation requires the RP to store the derived public key and credential data, much like it would for backup and additional account credentials [35, §13.4.6]. This means that RPs must be able to support having multiple credentials associated with accounts, as recommended by the standard.

As this variant requires the delegator to authenticate with the RP as part of step ② in Figure 2a in order to submit the proxy's credential, it can learn its private key in the case where it does not store its own private keys locally, such as for hardware tokens, and provide this later as input to `authenticatorMakeDelegatedCredential`. However, this does require that RPs create their own user interfaces and additional backend logic to facilitate permissions, revocation and validity periods.

For remote delegation, the delegator may optionally provide `AAGUID` for the proxy user to the RP when submitting delegated credentials. Whilst this cannot be verified, it does allow for early rejection if the proxy's *claimed* `AAGUID` would not be sufficient. The RP may then validate this `AAGUID` when the proxy authenticates by calling `navigator.credentials.create()`, discarding the presented credential.

4.2.4 Direct delegation. For direct delegation, WebAuthn credential extension fields [35, §5.7] must be used to provide RPs with warrants during authentication (step ④ in Figure 2b). This offers the benefit that RPs do not need to support having multiple credentials associated with accounts as the warrant may be verified during the proxy's authentication without recording it. RPs that support our WebAuthn extension will parse the warrant given in the extension field, which may include permissions and validity timestamps, and verify against these as required. We view this additional information about permissions as additional auxiliary (public) input to the Delegate algorithm, signing it together with pkw and storing it in our warr warrant format.

Although the warrant may include additional signed data, such as its validity period and permissions for the proxy user, it may still need to be revoked before it expires. The recommended approach to revoke proxy access is for the delegator to replace its own account credential at the RP, which would result in the replacement of the delegator's public key, thus invalidating any existing warrants signed for that credential. We expect that in practice delegations will not be performed too frequently per account, making this approach viable. Alternatively, the RP could provide blacklist functionality.

As direct delegation occurs offline, its features may be more limited as the delegator cannot lookup, unless cached, what features RPs support, such as permissions and the delegation extension itself. This also means that if the delegator's authenticator does not store keys locally, it cannot sign the warrant and complete the delegation.

We expect direct delegation to be used in devices with better storage capabilities, such as mobile phones, which lends itself to the use of QR codes and proximity-based communication of setup and delegation data.

During authentication (steps ③ and ④ in Figure 2b), the RP may request an attestation statement from the proxy user's authenticator by calling `navigator.credentials.create()`, without storing the presented credential, allowing policies to be enforced.

4.3 Prototype for delegation in WebAuthn

To demonstrate technical feasibility, we have built a prototype with source code [1] using an open-source virtual authenticator by Culnane et al. [15], which we extended to implement the additional CTAP calls from Section 4.2.1. To demonstrate the setup and direct delegation phases, our prototype includes a simple companion application, written in Python, to import and export proxy public keys (pkp) and warrants ($\mathsf{pkw}, \mathsf{ddata}$).

The companion program uses the new CTAP calls and our PSUW primitive, as well as the existing ARKG implementation [37]. We refer to the prototype's source code repository [1] for detailed instructions, screen recordings, and discussion on the practical decisions made and how the prototype works.

4.4 Usability considerations

The usability of the delegation approaches depends greatly on their user-facing implementations, which has not been the focus of our prototype. User interaction is discussed in the standard (see e.g. [35, §§7.1,7.2]). In the case of the remote variant, key setup and permissions configuration is managed through the interface provided by RPs. Although this leads to a differing interfaces across RPs, these interfaces are designed according to the, potentially context-dependent, needs of the RP. We expect remote delegation to have similar usability consideration as with OAuth 2.0 access tokens, which users manage manually over a web interface—the usability and user acceptance of which, and other authentication mechanisms, is investigated by Ruoti et al. [49]. For direct delegation, the use of a client extension or companion application means that the delegation interface and process is consistent across RPs and can be managed in one central interface on user devices. This does mean that, for the occasional delegation, users must find and navigate a foreign interface, whereas for the remote variant, they would instead use their existing account settings and management pages. In contrast, for proxy users we do not see any changes to standard WebAuthn practice when it comes to accessing delegator's web accounts after the delegation credentials have been issued.

We observe that recent usability studies, such as the work by Kunke et al. [36], focus on currently-used approaches for WebAuthn recovery. It is currently too early to conduct a proper usability study for delegation in WebAuthn since current authenticators do not implement ARKG, nor PSUW, and most web browsers do not pass unknown WebAuthn extension data between RPs and authenticators.

5 CONCLUSION

We proposed two approaches for delegation of WebAuthn credentials preserving the security, privacy, and decentralisation aspects of the standard. Both approaches share the same setup procedure, employing ARKG, which makes them compatible with Yubico's recent proposal for account recovery in WebAuthn.

Our remote delegation approach allows account owners to configure and manage delegation with associated permissions at the relying party, whereas for direct delegation, the owner issues a warrant containing delegation credentials directly to the proxy user. To realise this approach, we introduced a novel type of proxy signatures with unlinkable warrants, which might be of independent interest. We discussed integration details using WebAuthn and CTAP extensions and ways to realise communication between delegators and proxies.

We conducted performance experiments of our primitive which show that delegated authentication can be achieved at a low cost of a few extra milliseconds when compared to the standard authentication in WebAuthn. We additionally provided a prototype to demonstrate how delegation could work in practice, with our scheme supporting ECDSA signatures as used in WebAuthn.

REFERENCES

[1] 2021. *Source code and documentation for WebAuthn credential delegation protoype.* https://github.com/UoS-SCCS/WebAuthn-Credential-Delegation

[2] 2021. *Source code for PSUW primitive.* https://github.com/UoS-SCCS/PSUW-Primitive

[3] Joël Alwen, Sandro Coretti, and Yevgeniy Dodis. 2019. The Double Ratchet: Security Notions, Proofs, and Modularization for the Signal Protocol. In *EUROCRYPT 2019*. Springer, LNCS.

[4] Jee Hea An, Yevgeniy Dodis, and Tal Rabin. 2002. On the Security of Joint Signature and Encryption. In *Advances in Cryptology — EUROCRYPT 2002*, Vol. 2332. Springer, LNCS, 83–107.

[5] Alessandro Armando, Roberto Carbone, Luca Compagna, Jorge Cuellar, and Llanos Tobarra. 2008. Formal analysis of SAML 2.0 web browser single sign-on: breaking the SAML-based single sign-on for google apps. In *FMSE '08*. 1–10.

[6] Manuel Barbosa, Alexandra Boldyreva, Shan Chen, and Bogdan Warinschi. 2021. Provable Security Analysis of FIDO2, Vol. 12827. Springer, LNCS.

[7] Bluetooth SIG, Inc. 2019. *Bluetooth Core Specification.* Technical Report.

[8] Alexandra Boldyreva, Adriana Palacio, and Bogdan Warinschi. 2012. Secure Proxy Signature Schemes for Delegation of Signing Rights. *Journal of Cryptology* 25 (2012), 57–115.

[9] Carsten Bormann and Paul E. Hoffman. 2020. *Concise Binary Object Representation (CBOR).* Technical Report.

[10] Jacqueline Brendel, Marc Fischlin, Felix Günther, and Christian Janson. 2017. PRF-ODH: Relations, Instantiations, and Impossibility Results. In *Advances in Cryptology – CRYPTO 2017*. Vol. 10403. LNCS, 651–681.

[11] Certicom Research. 2009. *SEC 1: Elliptic Curve Cryptography.* Technical Report.

[12] David Chaum. 1985. Security Without Identification: Transaction Systems to Make Big Brother Obsolete. *Commun. ACM* 28, 10 (1985), 1030–1044.

[13] David Chaum and Eugène van Heyst. 1991. Group Signatures. In *EUROCRYPT '91*, Vol. 547. Springer, LNCS, 257–265.

[14] Katriel Cohn-Gordon, Cas Cremers, Benjamin Dowling, Luke Garratt, and Douglas Stebila. 2020. A Formal Security Analysis of the Signal Messaging Protoco. *J. Cryptol.* 33, 4 (2020), 1914–1983.

[15] Chris Culnane, Christopher J. P. Newton, and Helen Treharne. 2021. Technical Report on a Virtual CTAP2 WebAuthn Authenticator. arXiv:2108.04131 Source code: https://github.com/UoS-SCCS/VirtualWebAuthn.

[16] David Derler, Christian Hanser, and Daniel Slamanig. 2014. Privacy-Enhancing Proxy Signatures from Non-interactive Anonymous Credentials. In *CODASPY '14*, Vol. 8566. Springer, LNCS, 49–65.

[17] D. Eastlake and T. Hansen. 2006. *US Secure Hash Algorithms (SHA and HMAC-SHA).* Technical Report.

[18] Manuel Fersch. 2018. *The Provable Security of Elgamal-type Signature Schemes.* Ph.D. Dissertation. Ruhr University Bochum, Germany.

[19] Manuel Fersch, Eike Kiltz, and Bertram Poettering. 2016. On the Provable Security of (EC)DSA Signatures. In *ACM CCS 2016*. ACM, 1651–1662.

[20] Daniel Fett, Ralf Küsters, and Guido Schmitz. 2016. A Comprehensive Formal Security Analysis of OAuth 2.0. In *ACM CCS 2016*. ACM, 1204–1215.

[21] FIDO Alliance. 2018. *Client to Authenticator Protocol (CTAP).* Technical Report.

[22] FIDO Alliance. 2018. *FIDO Authenticator Security Requirements.* Technical Report.

[23] FIDO Alliance. 2021. *FIDO Alliance Specifications Overview.* Technical Report.

[24] Nick Frymann, Daniel Gardham, Franziskus Kiefer, Emil Lundberg, Mark Manulis, and Dain Nilsson. 2020. Asynchronous Remote Key Generation: An Analysis of Yubico's Proposal for W3C WebAuthn. In *ACM CCS 2020*. ACM, 939–954.

[25] Georg Fuchsbauer and David Pointcheval. 2008. Anonymous Proxy Signatures. In *SCN '08*. Springer, 201–217.

[26] Thomas Groß. 2003. Security analysis of the SAML single sign-on browser/artifact profile. In *ACSAC '03*. IEEE, 298–307.

[27] Christian Hanser and Daniel Slamanig. 2013. Blank Digital Signatures. In *ASIA CCS '13*. ACM, 95–106.

[28] Christian Hanser and Daniel Slamanig. 2013. Warrant-Hiding Delegation-by-Certificate Proxy Signature Schemes. In *Progress in Cryptology — INDOCRYPT 2013*, Vol. 8250. Springer, LNCS, 60–77.

[29] D. Hardt. 2012. *The OAuth 2.0 Authorization Framework.* Technical Report.

[30] Yali Jiang, Fanyu Kong, and Xiuling Ju. 2010. Lattice-Based Proxy Signature. In *CIS '10*. IEEE, 382–385.

[31] Seungjoo Kim, Sangjoon Park, and Dongho Won. 1997. Proxy signatures, Revisited. In *Information and Communications Security*, Vol. 1334. Springer, 223–232.

[32] H. Krawczyk, M. Bellare, and R. Canetti. 1997. *HMAC: Keyed-Hashing for Message Authentication.* Technical Report.

[33] H. Krawczyk and P. Eronen. 2010. *HMAC-based Extract-and-Expand Key Derivation Function (HKDF).* Technical Report.

[34] Murray Kucherawy and Elizabeth Zwicky. 2015. *Domain-based Message Authentication, Reporting, and Conformance (DMARC).* Technical Report.

[35] Akshay Kumar, Emil Lundberg, J.C. Jones, Michael Jones, and Jeff Hodges. 2021. *Web Authentication: An API for accessing Public Key Credentials - Level 2.* Technical Report. W3C. https://www.w3.org/TR/webauthn-2/.

[36] Johannes Kunke, Stephan Wiefling, Markus Ullmann, and Luigi Lo Iacono. 2021. Evaluation of Account Recovery Strategies with FIDO2-based Passwordless Authentication. arXiv:2105.12477

[37] Emil Lundberg and Dain Nilsson. 2019. *Webauthn recovery extension.* https://github.com/Yubico/webauthn-recovery-extension/

[38] Masahiro Mambo, Keisuke Usuda, and Eiji Okamoto. 1996. Proxy signatures: Delegation of the power to sign messages. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Science* 79, 9 (1996), 1338–1354.

[39] Steven P Miller, B Clifford Neuman, Jeffrey I Schiller, and Jermoe H Saltzer. 1988. Kerberos authentication and authorization system. In *Project Athena Technical Plan.*

[40] D. M'Raihi, M. Bellare, F. Hoornaert, D. Naccache, and O. Ranen. 2005. *HOTP: An HMAC-Based One-Time Password Algorithm.* Technical Report.

[41] D. M'Raihi, S. Machani, M. Pei, and J. Rydell. 2011. *TOTP: Time-Based One-Time Password Algorithm.* Technical Report.

[42] Collin Mulliner, Ravishankar Borgaonkar, Patrick Stewin, and Jean-Pierre Seifert. 2013. SMS-Based One-Time Passwords: Attacks and Defense. In *DIMVA '13*, Vol. 7967. Springer, LNCS, 150–159.

[43] Nitin Naik and Paul Jenkins. 2016. An Analysis of Open Standard Identity Protocols in Cloud Computing Security Paradigm. In *ASC/PiCom/DataCom/CyberSciTech '16*. IEEE, 428–431.

[44] National Institute of Standards and Technology. 2013. *Digital Signature Standard (DSS).* Technical Report. https://doi.org/10.6028/nist.fips.186-4

[45] Sarah Pearman, Jeremy Thomas, Pardis Emami Naeini, Hana Habib, Lujo Bauer, Nicolas Christin, Lorrie Faith Cranor, Serge Egelman, and Alain Forget. 2017. Let's Go in for a Closer Look: Observing Passwords in Their Natural Habitat. In *ACM CCS 2017*. ACM, 295–310.

[46] Nick Ragouzis, John Hughes, Rob Philpott, Eve Maler, Paul Madsen, and Tom Scavo. 2008. *Security Assertion Markup Language (SAML) V2.0 Technical Overview.* Technical Report. OASIS Open.

[47] David Recordon and Drummond Reed. 2006. OpenID 2.0: a platform for user-centric identity management. In *DIM '06*. ACM, 11–16.

[48] Ronald L. Rivest, Adi Shamir, and Yael Tauman. 2001. How to Leak a Secret. In *Advances in Cryptology — ASIACRYPT 2001*, Vol. 2248. Springer, LNCS, 552–565.

[49] Scott Ruoti, Brent Roberts, and Kent Seamons. 2015. Authentication Melee: A Usability Analysis of Seven Web Authentication Systems. In *WWW '15*. ACM.

[50] N. Sakimura, J. Bradley, M. Jones, B. de Medeiros, and C. Mortimore. 2014. *OpenID Connect Core 1.0 incorporating errata set 1.* Technical Report. OpenID Foundation.

[51] Jim Schaad. 2017. CBOR Object Signing and Encryption (COSE). RFC 8152.

[52] Kwan Shum and Victor K Wei. 2002. A strong proxy signature scheme with proxy signer privacy protection. In *11th IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises*. IEEE, 55–56.

[53] Prabath Siriwardena. 2020. OpenID Connect (OIDC). In *Advanced API Security*. Apress, Berkeley, CA, 129–155.

[54] Sampath Srinivas, Dirk Balfanz, and Eric Tiffany. 2014. *Universal 2nd Factor (U2F) Overview.* Technical Report.

[55] Zuowen Tan and Zhuojun Liu. 2004. Provably secure delegation-by-certification proxy signature schemes. In *InfoSecu '04*. ACM, 38–43.

[56] Ke-li Wu, Jing Zou, Xiang-He Wei, and Feng-Yu Liu. 2008. Proxy group signature: a new anonymous proxy signature scheme. In *ICML '08*, Vol. 3. IEEE, 1369–1373.

[57] Yong Yu, Yi Mu, Willy Susilo, Ying Sun, and Yafu Ji. 2012. Provably secure proxy signature scheme from factorization. *Mathematical and Computer Modelling* 55, 3 (2012), 1160–1168.

[58] Yong Yu, Chunxiang Xu, Xinyi Huang, and Yi Mu. 2009. An efficient anonymous proxy signature scheme with provable security. *Computer Standards & Interfaces* 31, 2 (2009), 348–353.

[59] Yue Zhang, Jian Weng, Rajib Dey, Yier Jin, Zhiqiang Lin, and Xinwen Fu. 2019. On the (In)security of Bluetooth Low Energy One-Way Secure Connections Only Mode. arXiv:1908.10497

A RECALLING ARKG

In Figure 7, we recall the ARKG construction by Frymann et al. [24] for DL-based keys.

Their construction is in a cyclic group $\mathbb{G}$ of order q based on DL keys $(\mathsf{sk}, \mathsf{pk}) = (x, g^x)$ for a generator $g \in \mathbb{G}$ and $x \in \mathbb{Z}_q$. Note that multiplicative notation is used for group operations. The following three building blocks are required: a pseudorandom function $\mathsf{PRF}(k, m)$, message authentication code $\mathsf{MAC} = (\mathsf{KGen}, \mathsf{Tag}, \mathsf{Verify})$ and key derivation function $\mathsf{KDF}(k, l)$, used as $\mathsf{KDF}_1(k) = \mathsf{KDF}(k, l_1)$ and $\mathsf{KDF}_2(k) = \mathsf{KDF}(k, l_2)$, where $\mathsf{KDF}_1 : \mathbb{G} \to \mathbb{Z}_q$, $\mathsf{KDF}_2 : \mathbb{G} \to \{0,1\}^*$ with fixed labels l_1, l_2.

```
Setup(λ)
return $\mathsf{pp} = ((\mathbb{G}, g, q), \mathsf{MAC}, \mathsf{KDF}_1, \mathsf{KDF}_2)$

DerivePK($\mathsf{pk} = S$, aux)
1 : $(e, E) \leftarrow \mathsf{KGen}()$
2 : $ck \leftarrow \mathsf{KDF}_1(S^e)$
3 : $mk \leftarrow \mathsf{KDF}_2(S^e)$
4 : $P \leftarrow g^{ck} \cdot S$
5 : $\mu \leftarrow \mathsf{MAC}(mk, (E, \mathsf{aux}))$
6 : return $\mathsf{pk}' = P, \mathsf{cred} = (E, \mathsf{aux}, \mu)$

DeriveSK($\mathsf{sk} = s, (E, \mathsf{aux}, \mu)$)
1 : $ck \leftarrow \mathsf{KDF}_1(E^s)$
2 : $mk \leftarrow \mathsf{KDF}_2(E^s)$
3 : if $\mu = \mathsf{MAC}(mk, (E, \mathsf{aux}))$ then
4 :     return $\mathsf{sk}' = ck + s \mod q$
5 : else return $\perp$

KGen()
$x \leftarrow\$ \mathbb{Z}_q$
return $(\mathsf{sk}, \mathsf{pk}) = (x, g^x)$

Check($\mathsf{sk}' = x, \mathsf{pk}' = X$)
return $g^x \overset{?}{=} X$
```

Figure 7: ARKG construction for DL-based keys.

B SECURITY PROOFS

Here we present the security proofs for Theorems 1 and 2.

B.1 Proof of Theorem 1

PROOF. $\mathcal{G}_0$ is defined exactly by $\mathsf{Exp}^{\text{wu-}b}_{\mathcal{A}}(\lambda)$, giving

$$\Pr[\mathcal{G}_0 = 1] = \Pr\left[\mathsf{Exp}^{\text{wu-}b}_{\mathcal{A}}(\lambda) = 1\right]$$

We define $\mathcal{G}_1$ to be $\mathcal{G}_0$ where we replace line 4 with '$(\mathsf{warr}, \mathsf{ddata}) \leftarrow \mathsf{PSUW.Delegate}(\mathsf{skd}, \mathsf{pkp}_0)$' and line 5 with '$\bar{\sigma} \leftarrow \mathsf{PSUW.Sign}(\mathsf{skp}_0, \mathsf{warr}, \mathsf{ddata}, m)$'. We argue that the difference in success probabilities for $\mathcal{G}_0$ and $\mathcal{G}_1$ is negligible. In particular, [24, Remark 2] ensures any two keys output by (ARKG.DerivePK, ARKG.DeriveSK) are indistinguishable if ARKG has the PK-Unlinkability property. Thus, the difference in success probabilities for $\mathcal{G}_0$ and $\mathcal{G}_1$ is also bound by an adversary against the pku property, for a function $\varepsilon(\lambda)$ that is negligible in λ.

$$|\Pr[\mathcal{G}_0 = 1] - \Pr[\mathcal{G}_1 = 1]| \leqslant \varepsilon(\lambda)$$

We now observe that $\mathcal{G}_1$ is independent of the challenge bit b and thus the winning probability of $\mathcal{A}$ is exactly $\frac{1}{2}$. The sequence of games $\mathcal{G}_0$ to $\mathcal{G}_1$ gives

$$\Pr\left[\mathsf{Exp}^{\text{wu-}b}_{\mathcal{A}}(\lambda)\right] = \frac{1}{2} + \varepsilon(\lambda)$$

and hence

$$\mathsf{Adv}^{\text{wu-}b}_{\mathsf{PSUW},\mathcal{A}}(\lambda) = \left|\Pr\left[\mathsf{Exp}^{\text{wu-}b}_{\mathcal{A}}(\lambda)\right] - \frac{1}{2}\right| \leqslant \varepsilon(\lambda).$$

□

B.2 Proof of Theorem 2

PROOF. We begin by observing that an adversary $\mathcal{A}$ wins the experiment $\mathsf{Exp}^{\text{unforge}}_{\mathcal{A}}(\lambda)$ if, intuitively, it is either able to break the delegation of warrants *or* it is able to produce a forgery on the signature. This is reflected in line 4 of the experiment, in the body of PSUW.Verify, where two checks are performed. That is '$\mathsf{DS.Verify}(\mathsf{pkd}, \sigma, \mathsf{pkw})$' and '$\mathsf{DS.Verify}(\mathsf{pkw}, s, m)$'. We split the advantage of $\mathcal{A}$ over these two winning conditions $\mathcal{E}_1$ and $\mathcal{E}_2$, which we formally define in Figures 8 and 9, giving

$$\Pr\left[\mathsf{Exp}^{\text{unforge}}_{\mathcal{A}}(\lambda) = 1\right] = \Pr[\mathcal{E}_1 = 1] + \Pr[\mathcal{E}_2 = 1]$$

We argue that an adversary against unforgeability is bound by the union of probabilities over each winning condition.

$\mathcal{E}_1$: Forgery on delegation σ w.r.t pkd. We argue directly that the probability an adversary wins this game is bound by the strong unforgeability property of the digital signature DS.

To see this, we build an adversary $\mathcal{B}$ that utilises $\mathcal{A}$ against $\mathsf{Exp}^{\text{unforge}}_{\mathsf{PSUW},\mathcal{A}}(\lambda)$ to win $\mathsf{Exp}^{\text{suf-cma}}_{\mathsf{DS},\mathcal{B}}(\lambda)$. $\mathcal{B}$ invokes its experiment and receives its challenge key pair $(\mathsf{sk}^\star, \mathsf{pk}^\star)$, for which it must create a forgery. It challenges $\mathcal{A}$ against $\mathcal{E}_1$, given in Figure 8, with the exception that it sets $(\mathsf{skd}, \mathsf{pkd}) \leftarrow (\mathsf{sk}^\star, \mathsf{pk}^\star)$. Furthermore, in the delegate oracle $\mathcal{O}_{\mathsf{Delegate}}$, line 2 is replaced with a call to $\mathcal{B}$'s signing oracle, provided by the $\mathsf{Exp}^{\text{suf-cma}}_{\mathsf{DS},\mathcal{B}}(\lambda)$ experiment. All other oracle queries can be answered by $\mathcal{B}$, including $\mathcal{O}_{\mathsf{Sign}}$ which executes DS.Sign, since this is done with respect to keys generated by $\mathcal{E}_1$. Finally, $\mathcal{A}$ outputs $\bar{\sigma}$. $\mathcal{B}$ extracts σ and pkw from $\bar{\sigma}$. It sets

```
Exp^unforge_A(λ)
1 :  $(\mathsf{skd}, \mathsf{pkd}) \leftarrow\$ \mathsf{PSUW.DKGen}(\mathsf{pp})$
2 :  $(\bar{\sigma}, m) \leftarrow \mathcal{A}^{\mathcal{O}_{\mathsf{Reg}}, \mathcal{O}_{\mathsf{Delegate}}, \mathcal{O}_{\mathsf{Sign}}, \mathcal{O}_{\mathsf{Corr}}}(\mathsf{pp}, \mathsf{pkd})$
3 :  retrieve warr from $\bar{\sigma}$
4 :  return  PSUW.Verify$_{\mathcal{E}_1}(\mathsf{pkd}, \bar{\sigma}, m)$
             1 :  parse $\bar{\sigma}$ as $s$, warr
             2 :  parse warr as pkw, $\sigma$
             3 :  return $\mathsf{DS.Verify}(\mathsf{pkd}, \sigma, \mathsf{pkw})$
5 :          $\wedge\ ((\cdot, \mathsf{warr}) \notin \mathsf{LDel}\ \vee$
6 :              $[\exists \mathsf{pkp}, m \text{ s.t. } (\cdot, \cdot, \mathsf{warr}, \cdot, m) \notin \mathsf{LSign}\ \wedge$
7 :              $(\mathsf{pkp}, \mathsf{warr}) \in \mathsf{LDel} \wedge (\cdot, \mathsf{pkp}) \notin \mathsf{LCorrupt}])$
```

Figure 8: Formal experiment for $\mathcal{E}_1$.

```
Exp^unforge_A(λ)
1 :  $(\mathsf{skd}, \mathsf{pkd}) \leftarrow\$ \mathsf{PSUW.DKGen}(\mathsf{pp})$
2 :  $(\bar{\sigma}, m) \leftarrow \mathcal{A}^{\mathcal{O}_{\mathsf{Reg}}, \mathcal{O}_{\mathsf{Delegate}}, \mathcal{O}_{\mathsf{Sign}}, \mathcal{O}_{\mathsf{Corr}}}(\mathsf{pp}, \mathsf{pkd})$
3 :  retrieve warr from $\bar{\sigma}$
4 :  return  PSUW.Verify$_{\mathcal{E}_2}(\mathsf{pkd}, \bar{\sigma}, m)$
             1 :  parse $\bar{\sigma}$ as $s$, warr
             2 :  parse warr as pkw, $\sigma$
             3 :  return $\mathsf{DS.Verify}(\mathsf{pkw}, s, m)$
5 :          $\wedge\ ((\cdot, \mathsf{warr}) \notin \mathsf{LDel}\ \vee$
6 :              $[\exists \mathsf{pkp}, m \text{ s.t. } (\cdot, \cdot, \mathsf{warr}, \cdot, m) \notin \mathsf{LSign}\ \wedge$
7 :              $(\mathsf{pkp}, \mathsf{warr}) \in \mathsf{LDel} \wedge (\cdot, \mathsf{pkp}) \notin \mathsf{LCorrupt}])$
```

Figure 9: Formal experiment for $\mathcal{E}_2$.

$\sigma^\star \leftarrow \sigma$ and $m^\star \leftarrow \mathsf{pkw}$, and forwards $(\sigma^\star, m^\star)$ as a response to its unforgeability game. The check '$(\cdot, \mathsf{warr}) \notin \mathsf{LDel}$' in $\mathcal{E}_1$ ensures that if $\mathcal{A}$ wins, then $\sigma^\star, m^\star$ was not obtained from the signing oracle. In particular, since $\mathcal{A}$ is playing against strong unforgeability, any randomisation of a signature σ comprising $\mathsf{warr} \in \mathsf{LDel}$ (to trivially achieve $\mathsf{warr}^\star \notin \mathsf{LDel}$) would allow $\mathcal{B}$ to win its SUF-CMA experiment. Thus if $\mathcal{A}$ wins against $\mathcal{E}_1$, then $\mathcal{B}$ also wins. Hence

$$\Pr[\mathcal{E}_1 = 1] \leqslant \mathsf{Adv}^{\text{suf-cma}}_{\mathsf{DS},\mathcal{B}}(\lambda)$$

$\mathcal{E}_2$: Forgery on message signature s w.r.t pkw. We begin by defining a new game $\mathcal{G}_1$ as $\mathcal{G}_0 := \mathcal{E}_2$, given in Figure 9, except we require the adversary to output '$(\bar{\sigma}, m, \mathsf{skw})$' on line 2. We further modify the game with an additional check on line 6, giving '$(\cdot, \cdot, \mathsf{warr}, \cdot, m) \notin \mathsf{LSign}\ \wedge\ \mathsf{ARKG.Check}(\mathsf{skw}, \mathsf{pkw})$'. The difference in success probability between $\mathcal{G}_1$ and $\mathcal{G}_0$ is bound by the adversary's ability to produce s such that $\mathsf{DS.Verify}(\mathsf{pkw}, s, m) = 1$ without knowledge of an skw. We require that $\mathcal{A}$ does not rerandomise a signature obtained from an oracle query, but this is already enforced by the check that $(\cdot, \cdot, \mathsf{warr}, \cdot, m) \notin \mathsf{LSign}$. To embed the challenge keys for EUF-CMA, it must guess not only the proxy signer the adversary will target for a forgery for, but also the delegation. It does so with probability $1/q_{\mathsf{Del}}$ where q_{Del} is the length of

LDel. We note that the PK-Unlinkability property of ARKG ensures that keys derived from pkp are indistinguishable from keys created from DS.KGen, and in particular, from the challenge keys provided by the unforgeability experiment of DS. Finally, to answer oracle queries to $\mathcal{O}_{\mathsf{Sign}}$, the challenger instead uses its access to the DS signing oracle to compute $\sigma \leftarrow \mathsf{DS.Sign}(\mathsf{skw}^*, m)$. Hence, this is precisely the EUF-CMA property of DS and we have

$$|\Pr[\mathcal{G}_1 = 1] - \Pr[\mathcal{G}_0 = 1]| \leqslant q_{\mathsf{Del}} \cdot \mathsf{Adv}^{\text{euf-cma}}_{\mathsf{DS},\mathcal{B}}(\lambda)$$

$\mathcal{G}_1$. We now claim that $\mathcal{A}$'s advantage against $\mathcal{G}_1$ is bound by an adversary $\mathcal{B}$ against $\mathsf{Exp}^{\mathsf{msKS}}_{\mathsf{ARKG},\mathcal{B}}(\lambda)$ of ARKG. $\mathcal{B}$ invokes its experiment to obtain a challenge public key $\mathsf{pk}^\star$. It executes $\mathcal{G}_1$ against $\mathcal{A}$ and guesses for which key pair generated by $\mathcal{O}_{\mathsf{Reg}}$ the adversary will produce a forgery. On this registration query, $\mathcal{B}$ instead sets $(\mathsf{skp}, \mathsf{pkp}) \leftarrow (\perp, \mathsf{pk}^\star)$, otherwise it executes $\mathcal{O}_{\mathsf{Reg}}$ as normal. To answer $\mathcal{O}_{\mathsf{Sign}}$ queries, in the case $\mathsf{pkp} \neq \mathsf{pk}^\star$, it runs the oracle as described. Otherwise, instead of calling ARKG.DeriveSK on line 4 of the PSUW.Sign algorithm, it uses its private key oracle $\mathcal{O}_{\mathsf{sk}'}$ to obtain $\mathsf{skw}^\star$. If $\mathcal{A}$ queries $\mathcal{O}_{\mathsf{Corr}}(\mathsf{pkp})$ for $\mathsf{pkp} = \mathsf{pk}^\star$ then the game aborts. For $\mathcal{O}_{\mathsf{Delegate}}$ queries on $\mathsf{pkp}^\star$, it calls the $\mathcal{O}_{\mathsf{pk}'}$ oracle from $\mathsf{Exp}^{\mathsf{msKS}}_{\mathsf{ARKG},\mathcal{B}}(\lambda)$. Thus, $\mathcal{B}$ is able to simulate the experiment against $\mathcal{A}$ perfectly.

After some time, $\mathcal{A}$ outputs forgery $(\bar{\sigma}, m, \mathsf{skw})$. $\mathcal{B}$ retrieves $\mathsf{pkw}^\star$ from $\bar{\sigma}$ and searches LDel for the entry in the form of $(\mathsf{pkw}^\star, \mathsf{cred}^\star)$. It then builds the response $(\mathsf{skw}^\star, \mathsf{pkw}^\star, \mathsf{cred}^\star)$ and forwards it to its $\mathsf{Exp}^{\mathsf{msKS}}_{\mathsf{ARKG},\mathcal{B}}(\lambda)$ experiment. Since $(\mathsf{pkw}^\star, \mathsf{cred}^\star)$ were obtained from an execution of ARKG.DerivePK, the correctness of ARKG ensures a corresponding $\mathsf{sk}' \leftarrow \mathsf{ARKG.DeriveSK}(\mathsf{sk}^\star, \mathsf{cred}^\star)$ such that $\mathsf{ARKG.Check}(\mathsf{sk}', \mathsf{pkw}^\star)$ passes. Furthermore, since the experiment has verified that $\mathsf{ARKG.Check}(\mathsf{skw}^\star, \mathsf{pkw}^\star) = 1$, we have that $(\mathsf{skw}^\star, \mathsf{pkw}^\star, \mathsf{cred}^\star)$ wins the msKS game provided $\mathsf{cred}^\star$ was not queried. However, this only occurs if $\mathcal{A}$ called $\mathcal{O}_{\mathsf{Sign}}$ for $\mathsf{warr}^\star \ni \mathsf{pkw}^\star$, $\mathsf{ddata} = \mathsf{cred}^\star$, in which case it has lost its Unforgeability game. We note that $\mathcal{A}$ only wins its msKS experiment if it correctly embedded the challenge into the pkp for which $\mathcal{B}$ created a forgery. This occurs with probability $1/q_{\mathsf{Reg}}$, where q_{Reg} is the number of registration queries made to $\mathcal{O}_{\mathsf{Reg}}$, which is polynomial in λ. Thus the advantage of $\mathcal{A}$ is bound by an adversary $\mathcal{B}$ against msKS of ARKG, giving

$$\Pr[\mathcal{G}_1 = 1] \leqslant q_{\mathsf{Reg}} \cdot \mathsf{Adv}^{\mathsf{msKS}}_{\mathsf{ARKG},\mathcal{B}}(\lambda)$$

Therefore,

$$\Pr[\mathcal{E}_2 = 1] \leqslant q_{\mathsf{Del}} \cdot \mathsf{Adv}^{\text{euf-cma}}_{\mathsf{DS},\mathcal{B}}(\lambda) + q_{\mathsf{Reg}} \cdot \mathsf{Adv}^{\mathsf{msKS}}_{\mathsf{ARKG},\mathcal{B}}(\lambda)$$

Since the variables q_{Del} and q_{Reg} are polynomial in the security parameter, we conclude that the advantage of an adversary is bounded by a function $\varepsilon(\lambda)$ that is negligible in λ.

$$\Pr\left[\mathsf{Exp}^{\mathsf{unforge}}_{\mathcal{A}}(\lambda) = 1\right] = \Pr[\mathcal{E}_1 = 1] + \Pr[\mathcal{E}_2 = 1] \leqslant \varepsilon(\lambda)$$

□