FIDO2, CTAP 2.1, and WebAuthn 2: Provable Security and Post-Quantum Instantiation

Nina Bindel
SandboxAQ
Palo Alto, USA
nina.bindel@sandboxaq.com

Cas Cremers
CISPA Helmholtz Center for Information Security
Saarbrücken, Germany
cremers@cispa.de

Mang Zhao
CISPA Helmholtz Center for Information Security
Saarbrücken, Germany
mang.zhao@cispa.de

2023 IEEE Symposium on Security and Privacy (SP) | 978-1-6654-9336-9/23/$31.00 ©2023 IEEE | DOI: 10.1109/SP46215.2023.10179454

***Abstract*—The FIDO2 protocol is a globally used standard for passwordless authentication, building on an alliance between major players in the online authentication space. While already widely deployed, the standard is still under active development. Since version 2.1 of its CTAP sub-protocol, FIDO2 can potentially be instantiated with post-quantum secure primitives.**

We provide the first formal security analysis of FIDO2 with the CTAP 2.1 and WebAuthn 2 sub-protocols. Our security models build on work by Barbosa et al. for their analysis of FIDO2 with CTAP 2.0 and WebAuthn 1, which we extend in several ways. First, we provide a more fine-grained security model that allows us to prove more relevant protocol properties, such as guarantees about token binding agreement, the `None` attestation mode, and user verification. Second, we can prove post-quantum security for FIDO2 under certain conditions and minor protocol extensions. Finally, we show that for some threat models, the downgrade resilience of FIDO2 can be improved, and show how to achieve this with a simple modification.

I. Introduction

One of the largest projects globally to mitigate the problems of weak passwords is the FIDO protocol by the Fast Identity Online (FIDO) Alliance. The alliance has brought together over forty key companies in the online authentication space, including Amazon, Apple, Google, Intel, Microsoft, RSA, VISA, and Yubico, and has brought security devices to the wider public to improve the security of important logins.

The FIDO2 standard – the latest of the protocols – is built around two sub-protocols that are critical for enabling security-device supported logins. The first one is *WebAuthn*, which is a protocol between web applications, web browsers, and authenticator hardware tokens. At its core, WebAuthn allows a website (a Relying Party) to perform a *passwordless* challenge-response protocol with a token (an Authenticator) – where the browser acts as an intermediary – and challenges are signed by credential keys generated and stored in the token. The protocol supports multiple optional modes and features, such as attestation and user involvement.

The second relevant protocol is *CTAP* (Client To Authenticator Protocol), which is a protocol between an authenticator (e.g., a hardware security token) and a client (e.g., a browser). The goal of the protocol is to bind (and thus to restrict) which clients can use the authenticator's API (Application Programming Interface). To enable API access, the client asks the user to enter the authenticator's PIN; this PIN is checked by the token, and a shared secret is established that represents the binding and is used to authenticate all subsequent client accesses to the authenticator.

The FIDO2 standard, while already widely deployed, is subject to ongoing development. Previous versions of these standards have been studied. However, as we will see later, the main study has made strong assumptions that do no hold for the majority of deployed systems, such as relying on the attestation[1] mode to prove core properties. Moreover, the recently proposed CTAP 2.1 [6] includes a completely new base protocol that has not yet been analyzed in any framework.

Notably, the most recent version of the FIDO2 standard with CTAP 2.1 and WebAuthn 2 [11] appears to be "post-quantum ready", because it enables a mode of operation that only uses on symmetric cryptographic primitives, digital signatures, and KEMs (Key Encapsulation Mechanisms). However, no post-quantum instantiations have been proposed, nor has the CTAP 2.1 protocol received any analysis. In this work we set out to fill this gap: analyse the newest version and assess its post-quantum security.

Contributions

1) We prove that FIDO2 with WebAuthn 2 and CTAP 2.1 is provably secure against classical adversaries in a fine-grained security and protocol model. Our security models are more fine-grained or cover other aspects than previous versions such as [2, 10]. For example, we add important aspects such as algorithm negotiation, required user actions, and token binding. For CTAP 2.1, our security proofs confirm the stronger containment properties (reduced "blast radius") offered by the protocol compared to CTAP 2.0. Our analysis of WebAuthn 2 also has new implications for WebAuthn 1: we provide the first guarantees of the most widely used `None` attestation mode, user verification, user presence, and token binding. Notably, our analysis shows the registration

This research was supported by the Microsoft Identity Project Research Grant "Developing Post-Quantum Secure Identity Services" from Microsoft Security Response Center (MSRC).

[1] In the context of WebAuthn, "attestation" means identification of device type/manufacturer, and notably does not imply any check of the software that is being executed.

phase must be trusted, as acknowledged by the standard, but seemingly contradicting a result in [2].

2) We prove that if FIDO2 with WebAuthn 2 and CTAP 2.1 is instantiated with post-quantum (PQ) secure KEMs and signatures, then it is secure against quantum adversaries in the same model. We give concrete suggestions for PQ secure algorithm and negotiation design choices, including classical-PQ hybrids as suggested by standardization agencies, such as NIST (National Institute for Standards and Technology) [8].
3) We propose a simple improvement to WebAuthn 2 that improves its resilience to certain types of downgrade attack. While these can only occur for strong threat models, these improvements yield stronger classical security against broken cryptographic primitives, and are even more relevant for their PQ instantiations.

Overview

We provide a high-level background on FIDO2's CTAP and WebAuthn protocols, and previous analysis models, in Section II. Next, we define notational preliminaries in Section III. Afterwards, we first present the analysis of WebAuthn 2 in Section IV, and then that of CTAP 2.1 in Section V. We prove the security of their composition in FIDO2 in Section VI. We then return to related work in Section VII, and describe limitations and future work in Section VIII.

In the appendix, we give more detailed descriptions of the algorithms modeled. We provide a long version with supplementary material at [4] with the full proofs and further details.

II. Background

A. High-level overview of FIDO2

The FIDO2 protocol incorporates the two sub-protocols WebAuthn and CTAP, and involves four main types of parties: relying parties (e.g., a server, online service, or an operating system feature), authenticators (e.g., token or security key), clients (e.g., web browsers or other applications), and users. WebAuthn typically leaves the users implicit in the description of the authenticator.

Initially, the client and authenticator run the setup and binding phase of the CTAP 2.1 protocol. Once this is completed, relying parties can register and authenticate authenticators by running WebAuthn 2 through CTAP 2.1, which we depict in Figure 1. WebAuthn 2 is used in two phases: in the registration phase, an authenticator produces a fresh credential key pair whose public key is sent to the relying party and stored. Afterwards, each time the relying party wants to authenticate a user, it performs a challenge-response protocol with the authenticator who signs the challenge using the credential private key, which is then verified by the relying party. We next expand the two protocols in more detail.

1) WebAuthn: The goal of WebAuthn is to enable relying parties to authenticate users through authenticator tokens using a challenge-response protocol. WebAuthn is specified as an API rather than as a protocol; in practice, a common scenario is that the relying party is an online service with server

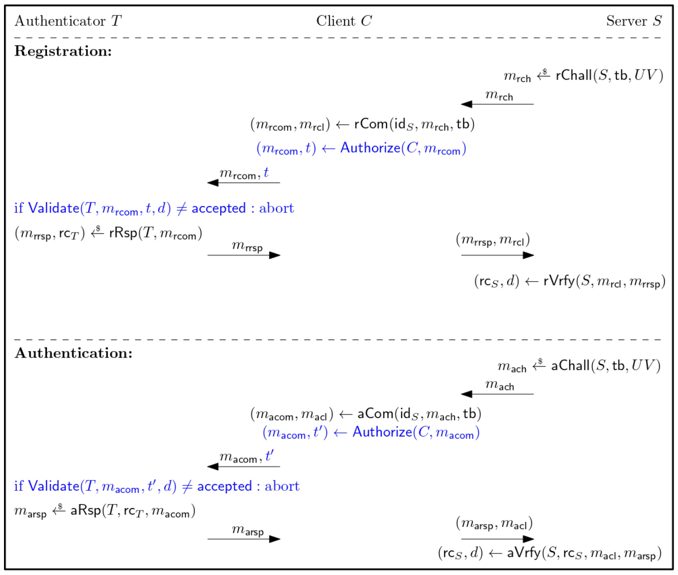

Authenticator T
Client C
Server S
Registration:
$m_{\mathsf{rch}} \overset{\$}{\leftarrow} \mathsf{rChall}(S, \mathsf{tb}, UV)$
m_{rch}
$(m_{\mathsf{rcom}}, m_{\mathsf{rcl}}) \leftarrow \mathsf{rCom}(\mathsf{id}_S, m_{\mathsf{rch}}, \mathsf{tb})$
$(m_{\mathsf{rcom}}, t) \leftarrow \mathsf{Authorize}(C, m_{\mathsf{rcom}})$
m_{rcom}, t
if $\mathsf{Validate}(T, m_{\mathsf{rcom}}, t, d) \neq \mathsf{accepted}$: abort
$(m_{\mathsf{rrsp}}, \mathsf{rc}_T) \overset{\$}{\leftarrow} \mathsf{rRsp}(T, m_{\mathsf{rcom}})$
m_{rrsp}
$(m_{\mathsf{rrsp}}, m_{\mathsf{rcl}})$
$(\mathsf{rc}_S, d) \leftarrow \mathsf{rVrfy}(S, m_{\mathsf{rcl}}, m_{\mathsf{rrsp}})$
Authentication:
$m_{\mathsf{ach}} \overset{\$}{\leftarrow} \mathsf{aChall}(S, \mathsf{tb}, UV)$
m_{ach}
$(m_{\mathsf{acom}}, m_{\mathsf{acl}}) \leftarrow \mathsf{aCom}(\mathsf{id}_S, m_{\mathsf{ach}}, \mathsf{tb})$
$(m_{\mathsf{acom}}, t') \leftarrow \mathsf{Authorize}(C, m_{\mathsf{acom}})$
m_{acom}, t'
if $\mathsf{Validate}(T, m_{\mathsf{acom}}, t', d) \neq \mathsf{accepted}$: abort
$m_{\mathsf{arsp}} \overset{\$}{\leftarrow} \mathsf{aRsp}(T, \mathsf{rc}_T, m_{\mathsf{acom}})$
m_{arsp}
$(m_{\mathsf{arsp}}, m_{\mathsf{acl}})$
$(\mathsf{rc}_S, d) \leftarrow \mathsf{aVrfy}(S, \mathsf{rc}_S, m_{\mathsf{acl}}, m_{\mathsf{arsp}})$

Fig. 1. The main message flow of FIDO2 with WebAuthn 2 with attestation type `None` is shown in black. The blue flows indicate interaction with the third CTAP 2.1 phase (i.e. after CTAP 2.1's setup and binding phases.) The user is left implicit in the flow of the authenticator token. For registration, the server generates a challenge. This is forwarded through the client to the token (possibly authorized through CTAP 2.1), which returns a public credential key and additional data, which is stored by the server. Afterwards, for each authentication, a similar process occurs, but the token now signs challenge and data with the with the signing key corresponding to the public credential key that was registered previously.

backend code and Javascript running in the browser, and the server's Javascript then uses the WebAuthn API supported by the browser to communicate with the token. The first interaction, when the server communicates with the token, is called registration phase. In this phase, the server S sends a challenge message (m_{rch}) to the token through the client C. This challenge contains a random nonce, parameters such as whether user verification (UV) is required, and optionally a value $\mathtt{tb}$ that uniquely identifies the underlying channel (in practice typically identifying a unique Transport Layer Security (TLS) connection, which can provide channel binding to prevent some types of man-in-the-middle attacks).

The client C parses the challenge message and turns it into a command message (m_{rcom}) and a client message (m_{rcl}), and forwards the command message to the token T. The token T produces a credential public-private key pair, which is bound to the server S and enables S to perform verification during the following authentication phase, and outputs a response message (m_{rrsp}). The client then returns this together with the client message to the server S. The response message specifies the type of "attestation statement" selected by the token, which enables the server S to perform verification during the registration phase, and includes the credential public key. WebAuthn 2 supports five attestation types; these include

`Basic` and `None` [2]. Tokens that support type `Basic` are equipped with an attestation key pair, which is specific to the token model, but not unique: by design, the attestation key pair is shared by a batch of tokens[3]. The `None` mode provides no token-specific information and is supported by all tokens.

The authentication phase is executed after the completion of the registration in a slightly different way. When the client parses the challenge message (m_{ach}) from the server S and turns it into a command message (m_{acom}) and a client message (m_{acl}), followed by sending the command message to the token T. The token T produces a response message (m_{arsp}) signed using the credential private key, and bound to the server S. The server S finally accepts a response message and a client message only when they pass verification using the corresponding credential public key.

2) CTAP: (Client To Authenticator Protocol) The CTAP protocol allows the client (e.g., a browser) to communicate with the authenticator. Using only WebAuthn, any application might try to access a token to request credential keys or responses to challenges. In practice, we would like to limit the client applications that are allowed to use the token's API. One of the goals of the CTAP protocol is to limit this access.

CTAP proceeds in three phases. In the setup phase, a client C' initializes a PIN, which is collected from the user, into the token T. In the binding phase, the client C (not necessarily same as C') and the token T exchange a shared binding state, if the client C is able to provide information about the PIN stored on the token T. The binding state is expected to uniquely bind the client C to the token T. If the client C fails 3 times consecutively, the token T is rebooted and all previously established binding states are reset. If the client C fails 8 times in total, the token T is blocked. When the above preparation is done, the client C authorizes any command message by outputting a tag t, which is forward to the token T along with the command message itself. The token T only proceeds upon the positive decision d from the user, e.g., by pressing a button, and then validates the command message and the tag. In particular, a token only produces a response message in WebAuthn when its validation process in CTAP succeeds. Note that the binding state is repeatedly used during a period, the length of which depends on the concrete CTAP version and the type of token devices, and will be blocked afterwards.

B. Previous analysis by Barbosa et al. [2]

Barbosa et al. [2] gave the first formal analysis of FIDO2, and in particular the version with CTAP 2.0 and WebAuthn 1. We recall some important conclusions.

1) **WebAutnn**: Barbosa et al. formalize WebAuthn 1 as a *passwordless authentication* (PlA) protocol. Assuming the uniqueness of each attestation key pair, they then prove that WebAuthn 1 with attestation type `Basic` provides *secure passwordless authentication*. However, since each attestation key pair is in fact necessarily shared by a large batch of tokens, their main theorem establishes uniqueness properties of partnering that actually do not hold in practice, and has no clear implications for the `None` mode.
2) **CTAP**: Barbosa et al. formalize CTAP 2.0 as a *PIN-based access control for authenticators* (PACA) protocol. Then, they prove the *Unforgeability with trusted binding* ($\mathsf{UF\text{-}t}$) of CTAP 2.0. In Section VII-A we show that the difference between CTAP 2.0 and CTAP 2.1 is substantial, which means the previous results cannot simply be translated.

Thus, Barbosa et al. [2] provided the first formal analysis of FIDO2 with CTAP 2.0 and WebAuthn 1, which was groundbreaking in many ways, but as a first attempt also left open many questions and subtle proof issues. We provide a detailed comparison between [2] and our work in Section VII-B.

III. Preliminaries

Notation. In this paper, we write PQ in place of "post-quantum". We write λ for the security parameter of each protocol. We assume that λ is the implicit input of each algorithm if it is unambiguous. Let PPT and QPT respectively denote the probabilistic and quantum turning machines (e.g., adversaries) that are executed in polynomial time. For a finite set S, we use $x \overset{\$}{\leftarrow} S$ to represent sampling x uniformly at random from set S. For a value y, we write $x \leftarrow y$ for assigning y to x. For a probabilistic algorithm Y (resp. a deterministic algorithm Y'), we use $x \overset{\$}{\leftarrow} Y(z)$ (resp. $x \leftarrow Y'(z)$) to denote assigning the output of the execution Y (resp. Y') on input z. For an integer n, we denote by $[n] := \{1, ...n\}$ the set of integers from 1 to n. By $\{0,1\}^\star$ we denote the set of all strings with finite length. For each string x, let $|x|$ denote the bit-length of x. All undefined variables are initialized with a specific symbol $\perp$. In this paper, we use $\epsilon_{\Pi}^{\mathsf{sec}}$ to denote the advantage of any Compl adversary that breaks sec security of Π protocol, if the complexity $\mathsf{Compl} \in \{\mathsf{PPT}, \mathsf{QPT}\}$ is unambiguous from the context. We introduce two novel security notions in Section A and the cryptographic building blocks and all other security notions in [4]. We omit the analysis of protocol correctness for page limitations.

IV. WebAuthn 2 and Extended Passwordless Authentication Protocols

For our analysis of WebAuthn 2 and its PQ instantiation, we follow the high-level approach from [2], which proposed the class of PlA protocols that generalizes WebAuthn 1, and proposed a corresponding security notion. We provide a more fine-grained model of WebAuthn 2, notably including the default mode `None` in which no attestation is performed, as well as the user presence and user verification checks, and a stronger threat model. We compare the details in our work and [2] in Section VII-B These aspects and their security cannot be captured in the PlA class without modification. In this section, we therefore first extend [2]'s formalisation and propose the *extended* PlA (ePlA) protocol class, and instantiate WebAuthn 2 as an ePlA protocol. We then introduce our new

[2]The remaining three modes are: `Self`, `AttCA`, and `AnonCA`, which are less common and out of scope of this work.

[3]The number of tokens in each batch is at least 100,000, cf. [11, Section 14.4.1].

model to define *secure passwordless authentication* (auth) for ePlA protocols and prove that WebAuthn 2 satisfies it. We then show how to instantiate PQ-WebAuthn 2. Our proof of auth implies PQ security against a PPT if the schemes used in a session are PQ secure. We return to downgrade attacks in Section IV-E.

A. Extended Passwordless Authentication Protocols (ePlA)

Similar to the PlA model from [2], we define our *extended passwordless authentication protocol* ePlA by two phases, Register and Authenticate:

Register**:** a two-pass challenge-response protocol run between a token T, a client C, and a server S, which is run at most once per tuple (T, S) (i.e., not for additional clients). At the end, both T and S hold registration contexts, which are relevant for subsequent authentications. Register can be decomposed into the following algorithms:

- rChall**:** inputs a server S, a token binding state tb, and a user verification condition $UV \in \{\mathsf{true}, \mathsf{false}\}$, and outputs a challenge message m_{rch}, i.e., $m_{\mathsf{rch}} \xleftarrow{\$} \mathsf{rChall}(S, \mathsf{tb}, UV)$.
- rCom**:** inputs the intended server identity id_S, a challenge message m_{rch}, and a token binding state tb, and outputs a client message m_{rcl} and a command message m_{rcom}, i.e., $(m_{\mathsf{rcom}}, m_{\mathsf{rcl}}) \leftarrow \mathsf{rCom}(\mathsf{id}_S, m_{\mathsf{rch}}, \mathsf{tb})$.
- rRsp**:** inputs a token T and a command message m_{rcom} and outputs a response message m_{rrsp} and an token-associated registration context rc_T, i.e., $(m_{\mathsf{rrsp}}, \mathsf{rc}_T) \xleftarrow{\$} \mathsf{rRsp}(T, m_{\mathsf{rcom}})$.
- rVrfy**:** inputs a server S, a client message m_{rcl}, and a response message m_{rrsp}, and outputs a server-associated registration context rc_S and a decision bit $d \in \{0, 1\}$ to indicate whether the registration request was accepted $(d = 1)$ or not $(d = 0)$, i.e., $(\mathsf{rc}_S, d) \leftarrow \mathsf{rVrfy}(S, m_{\mathsf{rcl}}, m_{\mathsf{rrsp}})$.

Authenticate**:** a two-pass challenge-response protocol run between a token T, a client C, and a server S after a successful run of Register, in which both T and S generated their registration contexts. At the end, S either accepts or rejects the authentication attempt. Similarly to Register, Authenticate can be decomposed into four algorithms:

- aChall**:** inputs a server S, a token binding state tb, and a user verification condition $UV \in \{\mathsf{true}, \mathsf{false}\}$, and outputs a challenge message m_{ach}, i.e., $m_{\mathsf{ach}} \xleftarrow{\$} \mathsf{aChall}(S, \mathsf{tb}, UV)$.
- aCom**:** inputs the intended server identity id_S, a challenge message m_{ach}, and a token binding state tb, and outputs a client message m_{acl} and a command message m_{acom}, i.e., $(m_{\mathsf{acl}}, m_{\mathsf{acom}}) \leftarrow \mathsf{aCom}(\mathsf{id}_S, m_{\mathsf{ach}}, \mathsf{tb})$.
- aRsp**:** inputs a token T along with its associated registration context rc_T, and a command message m_{acom}, and outputs a response message m_{arsp} and the updated registration context rc_T, i.e., $(m_{\mathsf{arsp}}, \mathsf{rc}_T) \xleftarrow{\$} \mathsf{aRsp}(T, \mathsf{rc}_T, m_{\mathsf{acom}})$.
- aVrfy**:** inputs a server S along with its associated registration context rc_S, a client message m_{acl}, and a response message m_{arsp}, and outputs the updated registration context rc_S and a decision bit $d \in \{0, 1\}$ indicating whether the authentication request was accepted by the user (output 1) or not (output 0), i.e., $(\mathsf{rc}_S, d) \leftarrow \mathsf{aVrfy}(S, \mathsf{rc}_S, m_{\mathsf{acl}}, m_{\mathsf{arsp}})$.

To model concurrent or sequential sessions of a server S (associated with ID id_S) and sequential sessions of a token T, we use π_S^i and π_T^j to denote their i-th and j-th instances respectively, i.e., $S = \{\pi_S^i\}_i$ and $T = \{\pi_T^j\}_j$. Our new abstraction retains the black message flow from Figure 1.

B. WebAuthn 2 is an ePlA Protocol

We use the following session variables for WebAuthn 2.

$\pi_S^i.\mathsf{ch}$: challenge nonce sampled in this session
$\pi_S^i.\mathsf{uid}$: user identifier sampled in this session
$\pi_S^i.\mathsf{tb}$: token binding state used in this session
$\pi_S^i.UV$: user verification condition, indicating whether the user should be verified, e.g., via PIN or Biometrics
$\pi_S^i.UP$: user presence condition, indicating whether the presence of the user is sufficient; constant true value
$\pi_S^i.\mathsf{pkCP}$: list of digital signature schemes accepted by S
$\pi_T^j.\mathsf{suppUV}$: indicates whether T supports user verification
$\pi_S^i.\mathsf{st}_{\mathsf{exe}}, \pi_T^j.\mathsf{st}_{\mathsf{exe}} \in \{\perp, \mathsf{running}, \mathsf{accepted}\}$: execution state of each session
$\pi_S^i.\mathsf{agCon}, \pi_T^j.\mathsf{agCon}$: the content that is expected to be agreed with other parties. These variables are protocol-specific. In WebAuthn 2, both variables include the server identifier, the hash of the client messages, the UP and UV conditions, and other session-specific data.
$\pi_S^i.\mathsf{sid}, \pi_T^j.\mathsf{sid}$: session identifiers. Two distinct sessions that have communicated with each other are expected to own the identical session identifiers. These variables are protocol-specific. In WebAuthn 2, both variables include the hash of the server identifier and other session-specific data.

Intuitively, the registration phase starts with the execution of rChall algorithm, where the server S samples random challenge nonce $\pi_S^i.\mathsf{ch}$ and user identifier $\pi_S^i.\mathsf{uid}$, initializes the user verification condition $\pi_S^i.UV$, and outputs the challenge message m_{rch}, which includes the above data as well as the server domain id_S and accepted list $\pi_S^i.\mathsf{pkCP}$. Additionally, the server also stores the token binding states $\pi_S^i.\mathsf{tb}$, which is shared with a client. Receiving m_{rch}, the client is supposed to verify the server domain followed by computing the hash value h of the client message $m_{\mathsf{rcl}} := (\mathsf{ch}, \mathsf{tb})$. Compared with m_{rch}, the output command message m_{rcom} replaces ch with h and add a constant user presence condition $UP := \mathsf{true}$. Receiving m_{rcom}, the token T picks a suitable signature scheme Σ in the list pkCP (if available) and checks whether the user verification mechanism is supported (if required). After that, T samples a public-private key pair (pk, sk) of Σ and a credential identifier cid, followed by initializing the associated registration context $\mathsf{rc}_T[\mathsf{id}_S]$ and the agreed content $\pi_T^j.\mathsf{agCon}$. The session identifier $\pi_T^j.\mathsf{sid}$ is set to the hash of the server domain, the credential identifier, and the initial counter $n := 0$. The output response message m_{rrsp} includes the session identifier $\pi_T^j.\mathsf{sid}$ as well as pk, Σ, UP and UV. The server S finally inputs both m_{rcl} and m_{rrsp} and executes a number of checks. If all checks pass, S also initializes its associated registration context $\mathsf{rc}_S[\mathsf{cid}]$ and the agreed content $\pi_S^i.\mathsf{agCon}$. The session identifier $\pi_S^i.\mathsf{sid}$ is identical to $\pi_T^j.\mathsf{sid}$.

The authentication phase is very similar. The crucial difference is that the token outputs a signature, which signs the $\pi_S^i.\mathsf{agCon}$-relevant data using the private key sk of Σ. Moreover, the session identifiers of token and servers additionally include the hash of the client message m_{acl}.

We give the concrete definition of algorithms of WebAuthn 2 with the default attestation type `None` in Section C.

C. Security Experiment for ePIA

The desired security property is that a server accepts an authentication response if and only if it was generated by a unique honest partnered token session. We capture it by our auth security experiment in Figure 2.

a) Threat Model: To closely capture the official security statement[4], we assume that all communication channels in the registration phase are authenticated. In contrast, there are no security assumptions on the communication channels between token, client, and server in the authentication phase. We assume that the users always provide the user presence or user verification confirmation when it is required and leave the users implicit in the security model. We assume the identifier id_S of each server S is unique. Unlike [2], we do *not* assume tokens to be "tamper-proof", i.e., the adversary is allowed to corrupt locally stored registration contexts.

b) Oracles: During the game execution the adversary $\mathcal{A}$ can create new servers and tokens through the oracles NEWS and NEWTPLA. In particular, the adversary can customize the concrete setting of the created parties, i.e., the supported signature list of the server and whether the token supports user verification. By invoking the REGISTER oracle, $\mathcal{A}$ is able to eavesdrop on honest registrations between servers and tokens of its choice. Moreover, via the oracles CHALLENGE, RESPONSE and COMPLETE, $\mathcal{A}$ can actively interfere during authentication. Note that sessions which have accepted or rejected can no longer be queried. Furthermore, the adversary $\mathcal{A}$ can also query the CORRUPT oracle to reveal a token's registration context related to a server.

c) Session Partnering: Partnering identifies token and server sessions that are successfully communicating with each other as expected, and is encoded through matching session identifiers. More precisely, we say a server session π_S^i *partners with a token session* π_T^j if and only if $\pi_S^i.\mathsf{sid} = \pi_T^j.\mathsf{sid} \neq \perp$. We say a server session π_S^i *partners with a token* T if it partners with one of T's sessions. We say a token T is the *registration partner* of a server S, if the registration context of T at S has been set, i.e., $\mathsf{rc}_T[\mathsf{id}_S] \neq \perp$.

d) Winning Conditions: We call a server session a *test session* if it accepts a response message. We say that the secure passwordless authentication for an ePIA holds if there exists a test session π_S^i such that none of the following winning conditions holds:

1) the non-$\perp$ session identifiers of two token sessions collide.
2) the non-$\perp$ session identifiers of two server sessions collide.
3) π_S^i does not partner with T and CORRUPT(S, T) was not queried (i.e., the registration context of T at S has not been revealed), where T is any registration partner of S.
4) the agreed contents of a pair of partnered server session $\pi_{S'}^{i'}$ and token sessions $\pi_{T'}^{j'}$ are distinct and CORRUPT(S', T') has not been queried.

[4]"Under the assumption that a registration ceremony is completed securely, and that the authenticator maintains confidentiality of the credential private key, subsequent authentication ceremonies using that public key credential are resistant to man-in-the-middle attacks" [11, Section 13.4.4]

```
$\mathsf{Expt}^{\mathsf{auth}}_{\mathsf{ePIA},\mathsf{Compl}}(\mathcal{A})$:
  $\mathcal{L}_{\mathsf{frsh}} \leftarrow \emptyset$
  $\mathsf{win\text{-}auth} \leftarrow 0$
  $() \xleftarrow{\$} \mathcal{A}^{\mathcal{O}}(1^\lambda)$
  return $\mathsf{win\text{-}auth}$

$\mathsf{regPartner}(S)$:
  if $\exists T$ such that $\mathsf{rc}_T[\mathsf{id}_S] \neq \perp$
    return $T$
  return $\perp$

$\mathsf{Win\text{-}auth}(S, i)$:
  if $\exists (T_1, j_1), (T_2, j_2)$ such that $(T_1, j_1) \neq (T_2, j_2)$ and $\pi_{T_1}^{j_1}.\mathsf{sid} = \pi_{T_2}^{j_2}.\mathsf{sid} \neq \perp$ : return 1
  if $\exists (S_1, i_1), (S_2, i_2)$ such that $(S_1, i_1) \neq (S_2, i_2)$ and $\pi_{S_1}^{i_1}.\mathsf{sid} = \pi_{S_2}^{i_2}.\mathsf{sid} \neq \perp$ : return 1
  $T \leftarrow \mathsf{regPartner}(S)$
  if $(S, T) \in \mathcal{L}_{\mathsf{frsh}}$ and $\neg\exists j$ such that $\pi_S^i.\mathsf{sid} = \pi_T^j.\mathsf{sid}$: return 1
  if $\exists (S', i'), (T', j')$ such that $\pi_{S'}^{i'}.\mathsf{sid} = \pi_{T'}^{j'}.\mathsf{sid} \neq \perp$ and $(S', T') \in \mathcal{L}_{\mathsf{frsh}}$ and $\pi_{S'}^{i'}.\mathsf{agCon} \neq \pi_{T'}^{j'}.\mathsf{agCon}$: return 1
  return 0

REGISTER$((S, i), (T, j), \mathsf{tb}, UV)$:
  if $\mathsf{pkCP}_S = \perp$ or $\mathsf{suppUV}_T = \perp$ or $\pi_S^i \neq \perp$ or $\pi_T^j \neq \perp$ or $\mathsf{rc}_T[S] \neq \perp$
    return $\perp$
  $\pi_S^i.\mathsf{pkCP} \leftarrow \mathsf{pkCP}_S$
  $\pi_T^j.\mathsf{suppUV} \leftarrow \mathsf{suppUV}_T$
  $m_{\mathsf{rch}} \xleftarrow{\$} \mathsf{rChall}(\pi_S^i, \mathsf{tb}, UV)$
  $(m_{\mathsf{rcom}}, m_{\mathsf{rcl}}) \leftarrow \mathsf{rCom}(\mathsf{id}_S, m_{\mathsf{rch}}, \mathsf{tb})$
  $(m_{\mathsf{rrsp}}, \mathsf{rc}_T) \xleftarrow{\$} \mathsf{rRsp}(\pi_T^j, m_{\mathsf{rcom}})$
  $(\mathsf{rc}_S, d) \xleftarrow{\$} \mathsf{rVrfy}(\pi_S^i, m_{\mathsf{rcl}}, m_{\mathsf{rrsp}})$
  $\mathcal{L}_{\mathsf{frsh}} \leftarrow \mathcal{L}_{\mathsf{frsh}} \cup \{(S, T)\}$
  return $(m_{\mathsf{rch}}, m_{\mathsf{rcl}}, m_{\mathsf{rcom}}, m_{\mathsf{rrsp}}, d)$

NEWS$(S, \mathsf{pkCP})$:
  if $\mathsf{pkCP}_S \neq \perp$
    return
  $\mathsf{pkCP}_S \leftarrow \mathsf{pkCP}$
  return

NEWTPLA$(T, \mathsf{suppUV})$:
  if $\mathsf{suppUV}_T \neq \perp$
    return
  $\mathsf{suppUV}_T \leftarrow \mathsf{suppUV}$
  return

CHALLENGE$((S, i), \mathsf{tb}, UV)$:
  if $\mathsf{pkCP}_S = \perp$ or $\pi_S^i \neq \perp$
    return $\perp$
  $\pi_S^i.\mathsf{pkCP} \leftarrow \mathsf{pkCP}_S$
  $m_{\mathsf{ach}} \leftarrow \mathsf{aChall}(\pi_S^i, \mathsf{tb}, UV)$
  return $m_{\mathsf{ach}}$

RESPONSE$((T, j), m_{\mathsf{acom}})$:
  if $\mathsf{suppUV}_T = \perp$ or $\pi_T^j \neq \perp$
    return $\perp$
  $\pi_T^j.\mathsf{suppUV} \leftarrow \mathsf{suppUV}_T$
  $(m_{\mathsf{arsp}}, \mathsf{rc}_T) \xleftarrow{\$} \mathsf{aRsp}(\pi_T^j, \mathsf{rc}_T, m_{\mathsf{acom}})$
  return $m_{\mathsf{arsp}}$

COMPLETE$((S, i), m_{\mathsf{acl}}, m_{\mathsf{arsp}})$:
  if $\pi_S^i = \perp$ or $\pi_S^i.\mathsf{st}_{\mathsf{exe}} \neq \mathsf{running}$
    return $\perp$
  $(\mathsf{rc}_S, d) \xleftarrow{\$} \mathsf{aVrfy}(\pi_S^i, \mathsf{rc}_S, m_{\mathsf{acl}}, m_{\mathsf{arsp}})$
  if $d = 1$
    $\mathsf{win\text{-}auth} \leftarrow \mathsf{Win\text{-}auth}(S, i)$
  return $d$

CORRUPT$(S, T)$:
  if $\mathsf{rc}_T[S] = \perp$
    return $\perp$
  $\mathcal{L}_{\mathsf{frsh}} \leftarrow \mathcal{L}_{\mathsf{frsh}} \setminus \{(S, T)\}$
  return $\mathsf{rc}_T[S]$
```

Fig. 2. Security experiment for extended Passwordless Authentication Protocols ePIA = (Register, Authenticate), where $\mathcal{O}$ = {NEWS, NEWTPLA, CORRUPT, REGISTER, CHALLENGE, RESPONSE, COMPLETE} and Compl ∈ {PPT, QPT}. We highlight the difference to PIA from [2] in blue. The variables agCon and sid are instance-specific, see Section IV-B.

Definition 1 (Secure passwordless authentication (auth) for ePIA)**.** *Let* Compl ∈ {PPT, QPT}. *Let* ePIA = (Register, Authenticate) *be an extended passwordless authentication protocol. We say that* ePIA *provides* secure passwordless authentication, *or* auth *for short, if for all* Compl *adversaries*

$\mathcal{A}$ the advantage

$$\mathsf{Adv}^{\mathsf{auth}}_{\mathsf{ePIA},\mathsf{Compl}}(\mathcal{A}) := \Pr\left[\mathsf{Expt}^{\mathsf{auth}}_{\mathsf{ePIA},\mathsf{Compl}}(\mathcal{A}) = 1\right]$$

in winning the game $\mathsf{Expt}^{\mathsf{auth}}_{\mathsf{ePIA},\mathsf{Compl}}$ defined in Figure 2 is negligible in the security parameter λ.

Conversely, we say a Compl adversary $\mathcal{A}$ breaks the secure passwordless authentication of ePIA for some test session π, if $\mathcal{A}$ wins $\mathsf{Expt}^{\mathsf{auth}}_{\mathsf{ePIA},\mathsf{Compl}}$ game via π.

In the following theorem, we show that WebAuthn 2 satisfies the defined security property auth. We sketch the proof here and give the full proof in [4].

Theorem 1 (PPT/QPT security of WebAuthn 2)**.** *Let $\mathsf{Compl} \in \{\mathsf{PPT}, \mathsf{QPT}\}$. Let $\mathsf{ePIA} = (\mathsf{Register}, \mathsf{Authenticate})$ denote the WebAuthn 2 protocol depicted in Figure 11. Assume that the underlying function H is $\epsilon^{\mathsf{coll\text{-}res}}_{\mathsf{H}}$-collision resistant. If there exists a Compl adversary $\mathcal{A}$ that breaks the secure passwordless authentication of ePIA for a test session π and the digital signature scheme Σ used in π is $\epsilon^{\mathsf{euf\text{-}cma}}_{\Sigma}$-$\mathsf{euf\text{-}cma}$ secure against Compl adversaries, then it holds that*

$$\mathsf{Adv}^{\mathsf{auth}}_{\mathsf{ePIA},\mathsf{Compl}}(\mathcal{A}) \leq \binom{q_{\text{Register}}}{2} 2^{-\lambda} + \binom{q_{\text{Challenge}}}{2} 2^{-\lambda} + \epsilon^{\mathsf{coll\text{-}res}}_{\mathsf{H}} + 2q_{\text{Register}}\epsilon^{\mathsf{euf\text{-}cma}}_{\Sigma}$$

where $q_{\mathcal{O}}$ denotes the number of $\mathcal{A}$'s queries to $\mathcal{O} \in \{\text{Register}, \text{Challenge}\}$.

Proof Sketch. Notice that the token session identifiers include credential identifiers, which are sampled of length $\geq \lambda$ for different tokens only in the Register queries, and a counter n, which is incremented in each sessions of the same token. The adversary $\mathcal{A}$ cannot win via winning condition in Line 8 except probability $\binom{q_{\text{Register}}}{2}2^{-\lambda}$. Note that the server session identifiers include the hash of server id, which is assumed to be unique for each server. Note also that the server session identifiers in the authentication phases additionally includes the hash of the token binding state tb and challenge nonces ch, which are of length $\geq \lambda$ and sampled only in the Challenge queries. The adversary $\mathcal{A}$ cannot win via winning condition in Line 9 except with probability $\binom{q_{\text{Challenge}}}{2}2^{-\lambda} + \epsilon^{\mathsf{coll\text{-}res}}_{\mathsf{H}}$. Finally, observe that the registration phases are authenticated and that the identifier of each server session in the authentication phases is set only when the corresponding server session accepts a signature, which signs the hash of the unique server id, the counter n, the hash of the client message m_{acl}, UP, and UV. Moreover, there are at most q_{Register} private signing keys in the experiment. The winning conditions in Line 11 and Line 12 indicate that the adversary $\mathcal{A}$ can forge any signature of Σ without corrupting the private signing key of any token, which happens with probability at most $2\epsilon^{\mathsf{euf\text{-}cma}}_{\Sigma}$ for each token and thus in total $2q_{\text{Register}}\epsilon^{\mathsf{euf\text{-}cma}}_{\Sigma}$. □

Theorem 1 shows that no polynomial-time attackers against WebAuthn 2 in the auth experiment can trigger any winning condition, through which the following aspects are captured. Conditions 1 and 2 capture the uniqueness of each session identifiers. i.e., if two sessions are partnered with each other, they are each other's unique partners. Condition 3 encodes the official security statement (see footnote 4). Condition 4 ensures that under the same assumption, the token and server sessions in the subsequent authentication ceremonies using that public key credential must agree on the server identifier id_S, the hash value $\mathsf{H}(\mathsf{ch}, \mathsf{tb})$, the local counter n, and the user presence UP and verification UV conditions. As a corollary, if the underlying hash function H is collision resistant, then the token and server sessions also implicitly agree on the token binding state tb.

D. Post-Quantum Instantiation of WebAuthn 2

To add the ability to authenticate using PQ or hybrid signature schemes with minimal changes to the WebAuthn 2 protocol, we propose to only extend the supported digital signature list pkCP (encoding an "or" choice) and explicitly allowing hybrid schemes (to encode "and", e.g., for classical and PQ schemes).

Following the WebAuthn 2 specification, the server has the option to include RSASSA–PKCS1–v1_5, RSASSA–PSS [12], or/and ECDSA–P256 [13] in pkCP, see Section II for an explanation of pkCP. Recall that the auth security of the WebAuthn 2 is proven in the standard model in Theorem 1. Therefore, the auth security for WebAuthn 2 also holds against quantum adversaries, assuming that $\epsilon^{\mathsf{coll\text{-}res}}_{\mathsf{H}}$ and $\epsilon^{\mathsf{euf\text{-}cma}}_{\Sigma}$ are sufficiently small against quantum adversaries, i.e., are instantiated with PQ secure algorithms. Instead of accepting only plain PQ signatures schemes, the server could also select hybrid signature schemes for pkCP as below.

Let Σ_1 and Σ_2 be signature schemes. We write $\mathcal{C}[\Sigma_1, \Sigma_2] = (\mathsf{KG}_{\mathcal{C}}, \mathsf{Sign}_{\mathcal{C}}, \mathsf{Vfy}_{\mathcal{C}})$ for the hybrid signature schemes constructed from Σ_1 and Σ_2[5]. $\mathsf{KG}_{\mathcal{C}}$ simply returns the concatenation of the two ingredient public and secret keys. Similarly, the signature returned by $\mathsf{Sign}_{\mathcal{C}}$ is the concatenation of the ingredient signatures over the same message. $\mathsf{Vfy}_{\mathcal{C}}$ returns 1 if and only if both ingredient signatures are valid. Otherwise it returns 0. The ingredient schemes could either be instantiated with different PQ (PQ-PQ hybrid), or with one classical and one PQ signature scheme (classical-PQ hybrid). Note that many other combiners exists, such as nested approaches that have been formalized in [5], which are particularly well suited to achieve backwards compatibility in, e.g., X.509 certificates.

In case of WebAuthn 2, backwards compatibility is important as not all authenticators, e.g., USB tokens, can be updated to support new algorithms via software updates. To offer backwards compatibility, the server includes classical algorithms in pkCP as less preferred algorithms and PQ/hybrid schemes with higher preference, e.g., $\mathsf{pkCP} = \{\Sigma_1 = \mathcal{C}[\Sigma_2, \Sigma_3], \Sigma_2, \Sigma_3\}$ with $\Sigma_3 \in$ {RSASSA–PKCS1–v1_5, RSASSA–PSS, ECDSA–P256}. Then, the (honest) token would always choose the more preferred hybrid or PQ algorithms for the PQ security, unless they are not supported.

[5]This description can easily be extended to more than two ingredient schemes.

E. Stronger Downgrade Protection

Our WebAuthn 2 results in the previous sections assume that the registration phase is authenticated (as in the standard), which means that the supported schemes list cannot be modified, and thus basic scheme downgrade attacks are impossible. On the other end of the spectrum, if an active attacker interferes continuously with *all* phases, we cannot detect or prevent downgrades.

However, there is an intermediate threat model, for which WebAuthn 2 could, but does not, provide downgrade protection. Note that the (ordered) list of the relying party's accepted signature algorithms $\pi_S^i.\mathsf{pkCP}$ is sent in plain from the relying party to the authenticator via the client (see Figure 11). The credential keys are then generated using the first algorithm in the *received* pkCP that is supported by the authenticator, see [11, Section 6.3.2.7.1]. During rVrfy, the relying party checks that the used signature scheme Σ is in $\pi_S^i.\mathsf{pkCP}$. Hence, if the communication in the registration phase is not authenticated, an adversary can easily change the list pkCP during transmission to the authenticator. For example, during the PQ transition, ideally security is based on classical and PQ algorithms in a backwards compatible way. While we explain how to achieve backwards compatibility with authenticators that only support classical algorithms in Section IV-D, a quantum adversary is able to break RSA or ECDSA might change pkCP such that the authenticator only has the choice between classical algorithms.

Consider an adversary that can forge signatures of one of the accepted and supported algorithms. Moreover, assume this adversary is able to compromise the browser or control the network used during registration but not the ones used for authentication, e.g., in an internet cafe a compromised machine is used for registration but others for authentication. Then tricking the authenticator to choose the vulnerable algorithm (and create a corresponding credential key pair) is beneficial because it allows the adversary to forge authentications later on even if they do not control the network anymore.

If the adversary has permanent control of the machine used for registration and authentication, and can forge signatures of an algorithm that is accepted and supported by the relying party and the authenticator, respectively, this attack cannot be prevented. Moreover, it is impossible to prevent the authenticator being tricked into using a less preferred algorithm without substantial changes to the WebAuthn 2 protocol and the public-key infrastructure within. However, we suggest changes that enable *detecting* such an event with high probability, calling the resulting protocol WebAuthn 2^+, if at least one message without interference of the adversary is sent. We depict the changes as boxed operations in Figure 11. Essentially, the idea is to include the hash $\mathsf{h_{CP}}$ of the *received* list of accepted algorithms pkCP' during registration, in the authentication response. The relying party compares $\mathsf{H}(\mathsf{pkCP})$ with $\mathsf{h_{CP}}$ to detect whether authenticator and relying party agree on the list of algorithms. To enable the above changes, both the relying party and the authenticator must store respective lists; we suggest to include them in the registration context.

```
Expt^{AlgAgree}_{WebAuthn 2^+,Compl}(A):
1   $(S, i, T, j, \mathsf{tb}, UV) \xleftarrow{\$} \mathcal{A}^{\mathcal{O}}(1^\lambda)$
2   $m^\star_{\mathsf{ach}} \xleftarrow{\$} \textsc{Challenge}((S,i), \mathsf{tb}, UV)$
3   $(m^\star_{\mathsf{acom}}, m^\star_{\mathsf{acl}}) \leftarrow \mathsf{aCom}(\mathsf{id}_S, m^\star_{\mathsf{ach}}, \mathsf{tb})$
4   $m^\star_{\mathsf{arsp}} \xleftarrow{\$} \textsc{Response}((T,j), m^\star_{\mathsf{acom}})$
5   $d^\star_a \xleftarrow{\$} \textsc{Complete}((S,i), m^\star_{\mathsf{acl}}, m^\star_{\mathsf{arsp}})$
6   $\mathsf{Supp} \leftarrow$ list of supported algorithms by $T$
7   $d^\star_{T,S} \leftarrow [\![(\mathsf{pkCP}_S \cap \mathsf{Supp})[1] \neq \mathsf{rc}_T[S].\Sigma]\!]$
8   return $[d^\star_{(T,S)} = 1 \wedge d^\star_a = 1]$

rChallenge((S, i), tb, UV):
9   if $\mathsf{pkCP}_S = \bot$ or $\pi_S^i \neq \bot$
10      return $\bot$
11  $\pi_S^i.\mathsf{pkCP} \leftarrow \mathsf{pkCP}_S$ //ordered list of accepted algorithms
12  $m_{\mathsf{rrsp}} \leftarrow \mathsf{rChall}(\pi_S^i, \mathsf{tb}, UV)$
13  return $m_{\mathsf{rrsp}}$

rResponse((T, j), m_rcom):
14  if $\mathsf{suppUV}_T = \bot$ or $\pi_T^j \neq \bot$
15      return $\bot$
16  $\pi_T^j.\mathsf{suppUV} \leftarrow \mathsf{suppUV}_T$
17  $(m_{\mathsf{rrsp}}, \mathsf{rc}_T) \xleftarrow{\$} \mathsf{rRsp}(\pi_T^j, m_{\mathsf{rcom}})$
18  return $m_{\mathsf{rrsp}}$

rComplete((S, i), m_rcl, m_rrsp):
19  if $\mathsf{pkCP}_S = \bot$ or $\pi_S^i = \bot$
    or $\pi_S^i.\mathsf{st_{exe}} \neq \mathsf{running}$ : return $\bot$
20  $(\mathsf{rc}_S, d) \xleftarrow{\$} \mathsf{rVrfy}(\pi_S^i, m_{\mathsf{rcl}}, m_{\mathsf{rrsp}})$
21  return $d$
```

Fig. 3. Game $\mathsf{Expt}^{\mathsf{AlgAgree}}_{\text{WebAuthn } 2^+,\mathsf{Compl}}$ and oracles rCHALLENGE, rRESPONSE, rCOMPLETE; note that NEWS, NEWTPLA, CHALLENGE, RESPONSE, and COMPLETE are given in Figure 2.

If an adversary changed the list pkCP during registration in WebAuthn 2^+, the adversary would need to change the value $\mathsf{h_{CP}}$ during every authentication response to avoid detection of the attack. We stress that it would not be sufficient to only reject authentications when such an attack is detected, since the honest authenticator would then be unable to communicate with the relying party due to the disagreement on the list pkCP. Even worse, only those authentication responses in which the adversary successfully switched the value $\mathsf{h_{CP}}$ would be accepted. Thus, the detection of this downgrade attack should trigger deregistering the authenticator by the relying party and notifying the user (ideally out-of-band).

More formally, we say that WebAuthn 2^+ satisfies our property *Algorithm Agreement* (AlgAgree) against $\mathsf{Compl} \in \{\mathsf{PPT}, \mathsf{QPT}\}$ adversaries if the advantage

$$\mathsf{Adv}^{\mathsf{AlgAgree}}_{\text{WebAuthn } 2^+,\mathsf{Compl}}(\mathcal{A}) := \Pr\Big[\mathsf{Expt}^{\mathsf{AlgAgree}}_{\text{WebAuthn } 2^+,\mathsf{Compl}}(\mathcal{A}) = 1\Big]$$

in winning the game $\mathsf{Expt}^{\mathsf{AlgAgree}}_{\text{WebAuthn } 2^+,\mathsf{Compl}}$ (defined in Figure 3) is negligible in the security parameter λ. We view WebAuthn 2^+ as an instantiation of an ePIA and give the adversary access to the following oracles: rCHALLENGE, rRESPONSE, and rCOMPLETE given in Figure 3, and NEWS, NEWTPLA, CHALLENGE, RESPONSE, and COMPLETE given in Figure 2.

The adversary wins the game $\mathsf{Expt}^{\mathsf{AlgAgree}}_{\text{WebAuthn } 2^+,\mathsf{Compl}}$ if the generated key pair is not of the most preferred server's algorithm that is supported by the token (i.e., it is not the first element in the intersection of the supported and the preferred algorithms, see line 7 in Figure 3), and honestly generated authentications are always accepted by the server (see line 5 in

Figure 3). It is important to emphasize that our threat model here is different than the one for Section IV-C. Namely, we assume that the communication channels in the registration and authentication phase are unauthenticated with one exception. We assume that there is at least one honest authentication, i.e., during this one authentication the adversary does not actively interfere with the communication between the three parties.

We can show that WebAuthn 2^+ satisfies the above property if H is a collision resistant hash function. The proof sketch is as follows. Assume the adversary $\mathcal{A}$ wins $\mathsf{Expt}^{\mathsf{AlgAgree}}_{\text{WebAuthn } 2^+,\mathsf{Compl}}$ (i.e., $d^\star_{(T,S)} = 1$ and $d^\star_a = 1$). This implies that the adversary is able to successfully register the token T at server S such that the chosen signature algorithm is supported by the token, accepted by the server, and not the most preferred algorithm in the intersection of supported and accepted algorithms. Furthermore, it means that line 57 in Figure 11 holds, i.e., that the hash value h_{CP} over the received list pkCP' (computed and sent by the token) is the same as the hash value $\mathsf{rc}_S[\mathsf{cid}].\mathsf{h}_{\mathsf{CP}}$ over the original pkCP. This contradicts the collision-resistance of H, as $\mathsf{pkCP} \neq \mathsf{pkCP}'$.

V. CTAP 2.1 and Extended Pin-based Access Control for Authenticator Protocols

In this section, we first define the *extended PIN-based Access Control for Authenticators* (ePACA) protocol following [2] and describe CTAP 2.1 as an ePACA instance. Next, we present a variant of the strong unforgeability with trust-binding (SUF-t′) experiment. Finally, we extend CTAP 2.1 for PQ compatibility and formally prove the SUF-t′ security of the extension.

A. Extended Pin-based Access Control for Authenticator Protocols

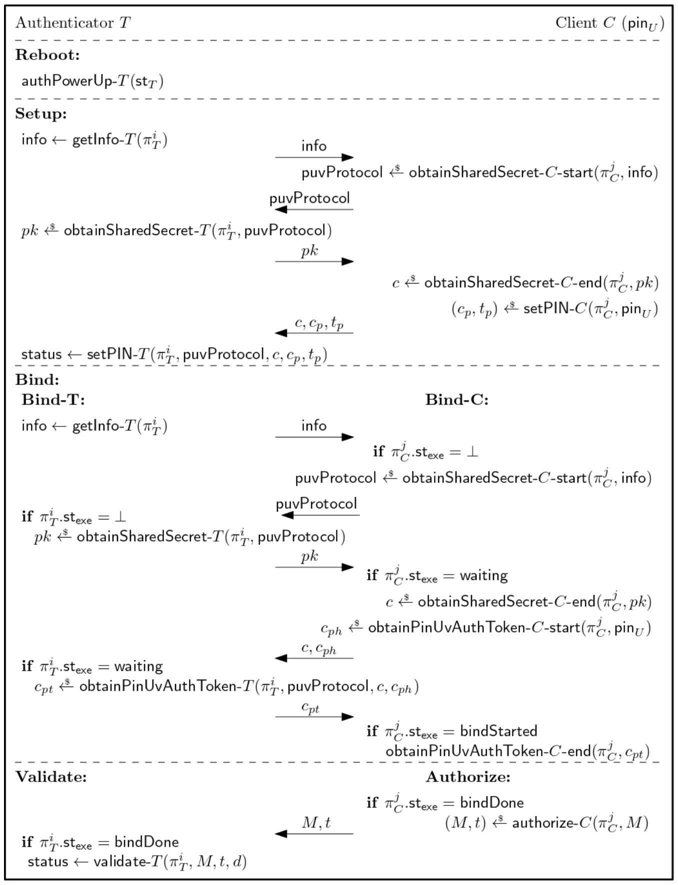
An *extended PIN-based Access Control for Authenticators* protocol $\mathsf{ePACA} = (\mathsf{Reboot}, \mathsf{Setup}, \mathsf{Bind}, \mathsf{Auth}, \mathsf{Validate})$ is an interactive protocol between a client C, an authenticator token T, and a user U, specified by the following algorithms:

$\mathsf{Reboot}(T)$**:** runs at each power-up of the token T and initializes the inherent state with a mandatory user interaction. This algorithm is expected to be invoked to power up T and initialize the local state before the execution of any other algorithms on T.

$\mathsf{Setup}(T, C, U)$**:** inputs a token T, a client C, and a user U and outputs the transcript trans. During this interactive sub-protocol, U securely transfers the PIN to T via C. Note that this algorithm is invoked on each token T at most once. We write $\mathsf{trans} \xleftarrow{\$} \mathsf{Setup}(T, C, U)$.

$\mathsf{Bind}(T, C, U)$**:** During this interactive sub-protocol, the client C is bound to the token T under the confirmation of the user U. This sub-protocol is further divided into two algorithms:

- $\mathsf{Bind\text{-}C}(C, U, m)$**:** inputs a client C, a user U, and an incoming message m and outputs an outgoing message m'. During this algorithm, C processes m under the confirmation from U. We write $m' \xleftarrow{\$} \mathsf{Bind\text{-}C}(C, U, m)$.
- $\mathsf{Bind\text{-}T}(T, m)$**:** inputs a token T and an incoming message m, and outputs an outgoing message m'. We write $m' \xleftarrow{\$} \mathsf{Bind\text{-}T}(T, m)$.



Authenticator T
Client C (pin_U)
Reboot:
$\mathsf{authPowerUp}\text{-}T(\mathsf{st}_T)$
Setup:
$\mathsf{info} \leftarrow \mathsf{getInfo}\text{-}T(\pi^i_T)$
info
$\mathsf{puvProtocol} \xleftarrow{\$} \mathsf{obtainSharedSecret}\text{-}C\text{-}\mathsf{start}(\pi^j_C, \mathsf{info})$
puvProtocol
$pk \xleftarrow{\$} \mathsf{obtainSharedSecret}\text{-}T(\pi^i_T, \mathsf{puvProtocol})$
pk
$c \xleftarrow{\$} \mathsf{obtainSharedSecret}\text{-}C\text{-}\mathsf{end}(\pi^j_C, pk)$
$(c_p, t_p) \xleftarrow{\$} \mathsf{setPIN}\text{-}C(\pi^j_C, \mathsf{pin}_U)$
c, c_p, t_p
$\mathsf{status} \leftarrow \mathsf{setPIN}\text{-}T(\pi^i_T, \mathsf{puvProtocol}, c, c_p, t_p)$
Bind:
Bind-T:
Bind-C:
$\mathsf{info} \leftarrow \mathsf{getInfo}\text{-}T(\pi^i_T)$
info
if $\pi^j_C.\mathsf{st}_{\mathsf{exe}} = \bot$
$\mathsf{puvProtocol} \xleftarrow{\$} \mathsf{obtainSharedSecret}\text{-}C\text{-}\mathsf{start}(\pi^j_C, \mathsf{info})$
puvProtocol
if $\pi^i_T.\mathsf{st}_{\mathsf{exe}} = \bot$
$pk \xleftarrow{\$} \mathsf{obtainSharedSecret}\text{-}T(\pi^i_T, \mathsf{puvProtocol})$
pk
if $\pi^j_C.\mathsf{st}_{\mathsf{exe}} = \mathsf{waiting}$
$c \xleftarrow{\$} \mathsf{obtainSharedSecret}\text{-}C\text{-}\mathsf{end}(\pi^j_C, pk)$
$c_{ph} \xleftarrow{\$} \mathsf{obtainPinUvAuthToken}\text{-}C\text{-}\mathsf{start}(\pi^j_C, \mathsf{pin}_U)$
c, c_{ph}
if $\pi^i_T.\mathsf{st}_{\mathsf{exe}} = \mathsf{waiting}$
$c_{pt} \xleftarrow{\$} \mathsf{obtainPinUvAuthToken}\text{-}T(\pi^i_T, \mathsf{puvProtocol}, c, c_{ph})$
c_{pt}
if $\pi^j_C.\mathsf{st}_{\mathsf{exe}} = \mathsf{bindStarted}$
$\mathsf{obtainPinUvAuthToken}\text{-}C\text{-}\mathsf{end}(\pi^j_C, c_{pt})$
Validate:
Authorize:
if $\pi^j_C.\mathsf{st}_{\mathsf{exe}} = \mathsf{bindDone}$
$(M, t) \xleftarrow{\$} \mathsf{authorize}\text{-}C(\pi^j_C, M)$
M, t
if $\pi^i_T.\mathsf{st}_{\mathsf{exe}} = \mathsf{bindDone}$
$\mathsf{status} \leftarrow \mathsf{validate}\text{-}T(\pi^i_T, M, t, d)$

Fig. 4. CTAP 2.1 is an $\mathsf{ePACA} = (\mathsf{Reboot}, \mathsf{Setup}, \mathsf{Bind}, \mathsf{Auth}, \mathsf{Validate})$ protocol. All algorithms are defined in Section D.

$\mathsf{Auth}(C, M)$**:** inputs a client C and a command M, and outputs both the command M and its authorization tag t. We write $(M, t) \xleftarrow{\$} \mathsf{Auth}(C, M)$.

$\mathsf{Validate}(T, M, t, d)$**:** inputs a token T, a command M, an authorization tag t, and a user decision $d \in \{\mathsf{accepted}, \mathsf{rejected}\}$, and outputs $\mathsf{status} \in \{\mathsf{accepted}, \mathsf{rejected}\}$ indicating whether the authorization can be verified or not. We write $\mathsf{status} \xleftarrow{\$} \mathsf{Validate}(T, M, t, d)$.

B. CTAP 2.1 is an ePACA protocol

CTAP 2.1 [6] is a substantial change from CTAP 2.0 [7] in terms of generalization and modularity. More concretely, CTAP 2.1 makes use of a generic stateful so-called *Pin/Uv Auth Protocol* $\mathsf{puvProtocol} = (\mathsf{initialize}, \mathsf{regenerate}, \mathsf{resetpuvToken}, \mathsf{getPublicKey}, \mathsf{encapsulate}, \mathsf{decapsulate}, \mathsf{encrypt}, \mathsf{decrypt}, \mathsf{authenticate}, \mathsf{verify})$, which can be instantiated using $\mathsf{puvProtocol}_1$ and $\mathsf{puvProtocol}_2$ from the standard that we depict in Section D. Additionally, we here propose a third instantiation $\mathsf{puvProtocol}_3$ that allows for PQ security in Section V-C. Each puvProtocol has its internal state including a public-private key pair (pk, sk) and a string pt.

Similar to the treatment in Section IV, we use π_T^i and π_C^j to denote token T's i-th and client C's j-th instance respectively. In addition, each T has a token-associated state st_T that is shared by all of T's instances. Namely, we have $T = \{\mathsf{st}_T\} \cup \{\pi_T^i\}_i$ and $C = \{\pi_C^j\}_j$. We use pin_U to denote U's unique PIN. In addition, we define the following variables for tokens T or clients C:

$\mathsf{st}_T.\mathsf{version} \in \{2.0, 2.1\}$**:** denotes the CTAP version.

$\mathsf{st}_T.\mathsf{puvProtocol}$**:** denotes a stateful Pin/Uv Auth Protocol.

$\mathsf{st}_T.\mathsf{puvProtocolList}$**:** denotes the list of Pin/Uv Auth Protocol instantiations that T supports.

$\mathsf{st}_T.\mathsf{pinHash} \in \{0,1\}^\star \cup \{\bot\}$**:** denotes the hash of a user PIN. This variable is expected to be set during Setup.

$\mathsf{st}_T.\mathsf{pinRetries} \in \{0, ..., \mathsf{pinRetriesMax}\}$**:** denotes the number of remaining tries for clients to deliver a pinHash, where pinRetriesMax denotes the maximal number of tries.

$\mathsf{st}_T.m \in \{0, ..., 3\}$**:** denotes the remaining consecutive tries for clients to deliver pinHash.

$\pi_T^i.\mathsf{st}_{\mathsf{exe}}, \pi_C^j.\mathsf{st}_{\mathsf{exe}} \in \{\mathsf{waiting}, \mathsf{bindStart}, \mathsf{bindDone}, \bot\}$**:** denotes the execution state of a token/client session.

$\pi_T^i.\mathsf{bs}, \pi_C^j.\mathsf{bs} \in \{0,1\}^\star \cup \{\bot\}$**:** denotes the binding state. This variable is expected to be set during Bind.

$\pi_T^i.\mathsf{sid}, \pi_C^j.\mathsf{sid} \in \{0,1\}^\star \cup \{\bot\}$**:** denotes the session identifiers; defined as the full transcript of the Bind execution.

$\pi_C^j.\mathsf{selectedpuvProtocol}$**:** denotes the puvProtocol instantiation chosen by the client.

$\pi_C^j.\mathsf{K} \in \{0,1\}^\star \cup \{\bot\}$**:** denotes the shared key with a token.

Next, we formalize CTAP 2.1 as an ePACA protocol. Overall CTAP 2.1 includes 12 algorithms[6]. We depict the communication flow of CTAP 2.1 in Figure 4. Intuitively, the Reboot algorithm initializes the underlying puvProtocols and resets the remaining consecutive tries $\mathsf{st}_T.m$ to 3.

In the Setup interaction, the token T first outputs its information info, which includes the supported list $\mathsf{st}_T.\mathsf{puvProtocolList}$. Next, the client selects and initializes one $\pi_C^j.\mathsf{selectedpuvProtocol}$ from the received list followed by sending its choice back to T. Then, the token T returns the public key pk of the chosen $\mathsf{st}_T.\mathsf{puvProtocol}$. Afterwards, the client runs the encapsulation of its $\pi_C^j.\mathsf{selectedpuvProtocol}$ upon pk for a key $\pi_C^j.\mathsf{K}$, which is then used to encrypt and authenticate the PIN pin_U collected from user U, and forwards all derived ciphertexts and tags to T. The token T finally decapsulates the key, followed by verifying the ciphertext, and recovers pin_U. The local $\mathsf{st}_T.\mathsf{pinHash}$ stores the hash of pin_U and the remaining retries $\mathsf{st}_T.\mathsf{pinRetries}$ is set to pinRetriesMax.

The Bind interaction is identical to Setup until the client derives the key $\pi_C^j.\mathsf{K}$. Then, the client uses $\pi_C^j.\mathsf{K}$ to encrypt the hash of the user pin pin_U and sends the ciphertext to T. If $\mathsf{st}_T.\mathsf{pinRetries}$ does not reach 0, the token decapsulates the key and recovers a pinHash. If the pinHash does not match the hash of the local $\mathsf{st}_T.\mathsf{pinHash}$, the underlying $\mathsf{st}_T.\mathsf{puvProtocol}$ re-generates the public key pk. If the remaining consecutive retries meanwhile arrive at 0, the token is forced to reboot. If the pinHash matches the hash of the local $\mathsf{st}_T.\mathsf{pinHash}$, the remaining retries $\mathsf{st}_T.m$ and $\mathsf{st}_T.\mathsf{pinRetries}$ are reset to their maximal values. The pts of all underlying $\mathsf{st}_T.\mathsf{puvProtocol}$ are re-sampled. The token finally sets the binding state $\pi_T^i.\mathsf{bs}$ to the pt of the current $\mathsf{st}_T.\mathsf{puvProtocol}$, which is then encrypted using the decapsulated key. The client eventually recovers the pt and sets it to $\pi_C^j.\mathsf{bs}$.

After the negotiation for the binding states, the client can invoke Auth algorithm to authorize command M using its binding state $\pi_T^i.\mathsf{bs}$. Similarly, the token can invoke the Validate algorithm to verify the authorized command using $\pi_C^j.\mathsf{bs}$.

We delay the description of the 12 algorithms to Section D.

C. Post-Quantum Instantiation of CTAP 2.1

We propose a third instantiation of the Pin/Uv Auth Protocol in Figure 5 that provides PQ compatibility in a hybrid manner. Compared to $\mathsf{puvProtocol}_2$, the most important changes made to achieve PQ security are as follows. First, in addition to an ECDH (over curve NIST P-256) key pair, a key pair of a PQ secure KEM is sampled during $\mathsf{regenerate}_3$. Second, the algorithm $\mathsf{encapsulate}_3$ executes both the ECDH key exchange and the encapsulation of the PQ KEM to derive a hybrid ciphertext c and key $K = (K_1, K_2)$. Finally, the algorithm $\mathsf{decapsulate}_3$ correspondingly recovers the hybrid key $K = (K_1, K_2)$ from the ciphertext c.

Instantiation: We suggest to instantiate the underlying KEM with any Round 3 Finalist nominated by NIST and the SKE_3 with AES-512-CBC with randomized initial vector. The underlying functions $\mathsf{H}_i : \{0,1\}^\star \to \{0,1\}^{l_i}$ for $i \in \{5,6,7\}$ can be instantiated with HMAC-SHA-512. Moreover, we suggest $\mu' \geq 2$ to against Grover's attack and to achieve 256-bits security.

D. Security Model of ePACA Protocols

Moving forward, we model the security of ePACA protocols as security experiment $\mathsf{Expt}^{\mathsf{SUF\text{-}t'}}_{\mathsf{ePACA},\mathsf{Compl}}$. The security goal is to ensure that a token can only accept a command that has been authorized by a trusted client under user permission.

a) Trust Model: Similarly to [2], we assume "trust-on-first-use", which means that the interactive execution of Setup is authenticated without any active interference of an eavesdropping adversary. Moreover, we assume no active attacks against clients during the interactive execution of Bind, while active attacks against tokens are allowed. More concretely, active attacks against clients are allowed only when the execution state of the clients turns from waiting to bindStart. However, active attacks against tokens are allowed even if the execution state of tokens is still waiting. We further assume that each user holds a unique PIN pin_U that is independently sampled from the domain $\mathcal{PIN}$[7] following some

[6]Similar to the treatment in [2], we omit the algorithms for PIN reset and leave it for future work. The suffix -T and -C in the names of algorithms indicates the algorithm executor to be either a token or a client. The suffix -start and -end indicates that this algorithm is the first or the final step in an interactive execution.

[7]In practice, each PIN must have a maximal length of 63 bytes and a minimal length of four code points (on tokens) or four unicode characters (on client).

```
$\mathsf{initialize}_3()$:
    $\mathsf{regenerate}_3()$
    $\mathsf{resetpuvToken}_3()$
$\mathsf{regenerate}_3()$:
    $(pk_1, sk_1) \xleftarrow{\$} \mathsf{ECDH.KG}()$
    $(pk_2, sk_2) \xleftarrow{\$} \mathsf{KEM.KG}()$
    $pk \leftarrow (pk_1, pk_2)$
    $sk \leftarrow (sk_1, sk_2)$
$\mathsf{encrypt}_3(K, m)$:
    $(K_1, K_2) \leftarrow K$
      s.t. $|K_1| = \mu'\lambda$
    $c \leftarrow \mathsf{SKE}_3.\mathsf{Enc}(K_2, m)$
    return $c$
$\mathsf{decrypt}_3(K, c)$:
    $(K_1, K_2) \leftarrow K$
      s.t. $|K_1| = \mu'\lambda$
    $m \leftarrow \mathsf{SKE}_3.\mathsf{Dec}(K_2, c)$
    return $m$
$\mathsf{authenticate}_3(K', m)$:
    $(K'_1, K'_2) \leftarrow K'$
      s.t. $|K'_1| = \mu'\lambda$
    $t \leftarrow \mathsf{H}_7(K'_1, m)$
    return $t$
$\mathsf{verify}_3(K', m, t)$:
    $(K'_1, K'_2) \leftarrow K'$
      s.t. $|K'_1| = \mu'\lambda$
    $t' \leftarrow \mathsf{H}_7(K', m)$
    return $[\![t = t']\!]$
$\mathsf{getPublicKey}_3()$:
    return $pk$
$\mathsf{resetpuvToken}_3()$:
    $pt \xleftarrow{\$} \{0,1\}^{\mu'\lambda}$
$\mathsf{encapsulate}_3(pk')$:
    $(pk'_1, pk'_2) \leftarrow pk'$
    $(sk_1, sk_2) \leftarrow sk$
    $Z_1 \leftarrow \mathsf{XCoordinateOf}(sk_1 \cdot pk'_1)$
    $(c_2, Z_2) \leftarrow \mathsf{KEM.Encaps}(pk'_2)$
    $Z \leftarrow \mathsf{H}_5(Z_1, Z_2)$
    $K_1 \leftarrow \mathsf{H}_6(Z, \text{“CTAP2 HMAC key”})$
    $K_2 \leftarrow \mathsf{H}_6(Z, \text{“CTAP2 AES key”})$
    $K \leftarrow (K_1, K_2)$
    $c \leftarrow (pk, c_2)$
    return $(c, K)$
$\mathsf{decapsulate}_3(c)$:
    Parse $(c_1, c_2) \leftarrow c$
    Parse $(sk_1, sk_2) \leftarrow sk$
    $Z_1 \leftarrow \mathsf{XCoordinateOf}(sk_1 \cdot c_1)$
    $Z_2 \leftarrow \mathsf{KEM.Decaps}(sk_2, c_2)$
    $Z \leftarrow \mathsf{H}_5(Z_1, Z_2)$
    $K_1 \leftarrow \mathsf{H}_6(Z, \text{“CTAP2 HMAC key”})$
    $K_2 \leftarrow \mathsf{H}_6(Z, \text{“CTAP2 AES key”})$
    $K \leftarrow (K_1, K_2)$
    return $K$
```

Fig. 5. The third instantiation of PIN/UV Auth Protocol $\mathsf{puvProtocol}_3$. The operation $\cdot$ denotes the scalar-multiplication.

distribution $\mathfrak{D}$ with min-entropy $\alpha_{\mathfrak{D}}$. All tokens are assumed to share a common pinRetriesMax. We assume that each ECDH point is bijective to its x-coordinate.

b) Experiment-specific Variables: Each session π is associated with a variable isValid $\in \{\mathsf{true}, \mathsf{false}, \perp\}$ that denotes whether the session is still accessible (by users or attackers) or not. Each token session π_T^i is associated with a variable pinCorr $\in \{\mathsf{true}, \mathsf{false}\}$ that indicates whether the setup user PIN of T has been corrupted.

c) Oracles: The oracles in our security experiment (see Figure 6) are defined similarly to the ones in [2]. More concretely, the oracles NEWT and NEWU create new tokens and users, respectively. In particular, the adversary can customize the token with specific initial data when querying NEWT. The REBOOT(T) oracle invokes Reboot and marks all previously established sessions of T as invalid. The oracle SETUP runs the authenticated interaction of Setup. The oracle EXECUTE captures that the Bind interaction is partially authenticated until the client's execution state is set to bindStart and the remaining interaction of Bind is not authenticated, as the adversary can deliver messages to token and client by SEND-BIND-T and SEND-BIND-C oracles respectively. The AUTH and VALIDATE oracles simulate the Auth and Validate execution of clients and tokens, respectively. Furthermore, querying CORRUPTUSER and COMPROMISE reveals a user's PIN and a client's binding state, respectively. Notably, whenever Reboot or Bind are completed on a token T, we mark all of T's previously established sessions as invalid.

d) Session Partnering: Partnering identifies the sessions of a token T and a client C that successfully completed Bind(T, C, U) for some user U. We call a token session π_T^i *partnered* with a client session π_C^j if and only if $\pi_T^i.\mathsf{sid} = \pi_C^j.\mathsf{sid} \neq \perp$.

e) Winning Conditions: We call a token session *test session* if it accepts an authorized command-tag pair under some user decision. An adversary $\mathcal{A}$ wins $\mathsf{Expt}^{\mathsf{SUF\text{-}t'}}_{\mathsf{ePACA},\mathsf{Compl}}$ (with Compl $\in \{\mathsf{PPT}, \mathsf{QPT}\}$) if there exists a test session π_T^i that accepts an authorized command (M, t) with user decision d and any of the following conditions holds:

1) the user decision $d \neq$ accepted.
2) two distinct client sessions that completed Bind have the same session identifiers.
3) two distinct token sessions that completed Bind have the same session identifiers.
4) (M, t) was not output by any of π_T^i's uncompromised valid partners π_C^j before the corruption of the user PIN that was setup on the token T.

Definition 2 (SUF-t$'$ security of ePACA)**.** *Let* Compl $\in \{\mathsf{PPT}, \mathsf{QPT}\}$. *Let* ePACA $=$ (Reboot, Setup, Bind, Auth, Validate) *be an extended PIN-based Access Control for Authenticators protocol. We say that* ePACA *is* strongly unforgeable with trusted binding, *or is* SUF-t$'$*-secure for short, if for all* Compl *adversaries* $\mathcal{A}$

$$\mathsf{Adv}^{\mathsf{SUF\text{-}t'}}_{\mathsf{PACA},\mathsf{Compl}}(\mathcal{A}) := \Pr[\mathsf{Expt}^{\mathsf{SUF\text{-}t'}}_{\mathsf{ePACA},\mathsf{Compl}}(\mathcal{A}) = 1]$$

in winning the game $\mathsf{Expt}^{\mathsf{SUF\text{-}t'}}_{\mathsf{ePACA},\mathsf{Compl}}$ *as described in Figure 6 is negligible in the security parameter* λ.

E. Security Conclusions for CTAP 2.1

After having defined security for ePACA protocols above, we now present the security statements for CTAP 2.1. We give the full proofs of our two theorems (against PPT and QPT adversaries) in [4]. Our first theorem shows the SUF-t$'$ security of CTAP 2.1 against PPT adversaries.

Theorem 2 (PPT security of CTAP 2.1)**.** *Let* ePACA $=$ (Reboot, Setup, Bind, Auth, Validate) *denote the CTAP 2.1 protocol described in Section V-B. Assume that* ePACA *supports* $\mathsf{puvProtocol}_i$ *for* $i \in \{1, 2, 3\}$. *If the hash function* H *is* $\epsilon_{\mathsf{H}}^{\mathsf{coll\text{-}res}}$ *collision resistant,* $\mathsf{H}_i : \{0,1\}^\star \rightarrow \{0,1\}^{l_i}$ *is modeled as independent random oracle for* $i \in \{1, ..., 7\}$, SKE_1 *is* $\epsilon_{\mathsf{SKE}_1}^{\mathsf{ind\text{-}1cpa\text{-}H_2}}$-IND-1CPA-H$_2$ *and* $\epsilon_{\mathsf{SKE}_1}^{\mathsf{ind\text{-}1\$pa\text{-}lpc}}$-IND-1\$PA-LPC *secure,* SKE_i *is* $\epsilon_{\mathsf{SKE}_i}^{\mathsf{ind\text{-}1cpa}}$-IND-1CPA *and* $\epsilon_{\mathsf{SKE}_i}^{\mathsf{ind\text{-}1\$pa\text{-}lpc}}$-IND-1\$PA-LPC *secure for* $i \in \{2, 3\}$, *and the* sCDH *problem over* ECDH *with prime order* q *is* $\epsilon_{\mathsf{ECDH}}^{\mathsf{sCDH}}$ *hard, then the advantage of any* PPT *adversary* $\mathcal{A}$

that breaks SUF-t′ *security of* ePACA *is bounded by*

$$\begin{aligned}
&\mathsf{Adv}^{\mathsf{SUF\text{-}t}'}_{\mathsf{PACA},\mathcal{A}}(1^\lambda)\\
\leq&(q_{\textsc{Setup}} + q_{\textsc{Execute}})\epsilon^{\mathsf{sCDH}}_{\mathsf{ECDH}} + \epsilon^{\mathsf{coll\text{-}res}}_{\mathsf{H}}\\
&+\binom{q_{\textsc{Setup}} + q_{\textsc{Execute}}}{2}(2^{2-\min\{l_1,l_3,l_5,l_6\}} + 2^{1-q})\\
&+q_{\textsc{NewU}}2^{-\alpha_{\mathfrak{D}}} + \binom{q_{\textsc{Send-Bind-T}}}{2}2^{-\min\{\mu,2,\mu'\}\lambda}\\
&+q_{\textsc{Setup}}\max\{\epsilon^{\mathsf{ind\text{-}1cpa\text{-}H_2}}_{\mathsf{SKE}_1}, \epsilon^{\mathsf{ind\text{-}1cpa}}_{\mathsf{SKE}_2}, \epsilon^{\mathsf{ind\text{-}1cpa}}_{\mathsf{SKE}_3}\}\\
&+q_{\textsc{Execute}}\max\{\epsilon^{\mathsf{ind\text{-}1\$pa\text{-}lpc}}_{\mathsf{SKE}_1}, \epsilon^{\mathsf{ind\text{-}1\$pa\text{-}lpc}}_{\mathsf{SKE}_2}, \epsilon^{\mathsf{ind\text{-}1\$pa\text{-}lpc}}_{\mathsf{SKE}_3}\}\\
&+q_{\textsc{Setup}}\mathsf{pinRetriesMax}2^{-\alpha_{\mathfrak{D}}}\\
&+q_{\textsc{Validate}}2^{-\min\{\mu\lambda,2\lambda,\mu'\lambda,l_2,l_4,l_7\}}
\end{aligned}$$

where $q_{\mathcal{O}}$ *denotes the number of queries to* $\mathcal{O} = \{\textsc{Setup}, \textsc{Execute}, \textsc{Validate}\}$ *and* q_i *denotes the number of queries to random oracle* H_i *for* $i \in \{1, ..., 7\}$.

Proof Sketch. The proof is divided into the following steps: (1) By the random oracle H_i for $i \in \{1, 3, 5, 6\}$ and the sCDH assumption on the underlying ECDH, all keys K derived from the encapsulation of the underlying puvProtocol in the obtainSharedSecret-C-end algorithm, which is only invoked in the $\textsc{Setup}$ and $\textsc{Execute}$ oracles, are distinct except probability $(q_{\textsc{Setup}}+q_{\textsc{Execute}})\epsilon^{\mathsf{sCDH}}_{\mathsf{ECDH}} + \binom{q_{\textsc{Setup}}+q_{\textsc{Execute}}}{2}2^{2-\min\{l_1,l_3,l_5,l_6\}}$. (2) By the entropy of the user PIN $\alpha_{\mathfrak{D}}$, none of the user PIN sampled in $\textsc{NewU}$ oracle is predicable except with probability $q_{\textsc{NewU}}2^{-\alpha_{\mathfrak{D}}}$. (3) By the collision-resistance of H and the entropy of the Diffie-Hellman public keys 2^q and of pt values $2^{\max\{\mu,2,\mu'\}\lambda}$, we have the all H(pin), Diffie-Hellman public keys, pt values, are respectively distinct except probability in total $\epsilon^{\mathsf{coll\text{-}res}}_{\mathsf{H}} + \binom{q_{\textsc{Setup}+\textsc{Execute}}}{2}2^{1-q} + \binom{q_{\textsc{Send-Bind-T}}}{2}2^{-\min\{\mu,2,\mu'\}\lambda}$. (4) By the IND-1CPA-H$_2$ security of SKE$_1$ and the IND-1CPA security of SKE$_2$ and SKE$_3$, the pins encrypted by the underlying puvProtocol in the setPIN-C algorithm, which is only invoked in the $\textsc{Setup}$ oracle, are indistinguishable from random except probability $q_{\textsc{Setup}}\max\{\epsilon^{\mathsf{ind\text{-}1cpa\text{-}H_2}}_{\mathsf{SKE}_1}, \epsilon^{\mathsf{ind\text{-}1cpa}}_{\mathsf{SKE}_2}, \epsilon^{\mathsf{ind\text{-}1cpa}}_{\mathsf{SKE}_3}\}$. (5) By the IND-1\$PA-LPC security of SKE$_i$ for $i \in \{1, 2, 3\}$, the pinHashs encrypted by the underlying puvProtocol in the obtainPinUvAuthToken-C-start algorithm, which is invoked only in the $\textsc{Execute}$ oracle, are indistinguishable from random except probability $q_{\textsc{Execute}}\max\{\epsilon^{\mathsf{ind\text{-}1\$pa\text{-}lpc}}_{\mathsf{SKE}_1}, \epsilon^{\mathsf{ind\text{-}1\$pa\text{-}lpc}}_{\mathsf{SKE}_2}, \epsilon^{\mathsf{ind\text{-}1\$pa\text{-}lpc}}_{\mathsf{SKE}_3}\}$.

Finally, the adversary $\mathcal{A}$ cannot trigger the flip of the win-SUF-t′ predicate in Figure 6 via condition (i) in Line 8, due to the design of CTAP 2.1, see validate-T algorithm in CTAP 2.1. (ii) in Line 9, due to the distinction of Diffie-Hellman public keys, (iii) in Line 10, due to the distinction of Diffie-Hellman public keys and pts, (iv) in Line 11-14, since $\mathcal{A}$ obtains no information about pins or pts and can only win by randomly guessing the pin in the $\textsc{Setup}$ oracle maximal pinRetriesMax times for each token session, or the pt values or the tags t in the Validate algorithm in the $\textsc{Validate}$ oracle, which happens with probability except $q_{\textsc{Setup}}\mathsf{pinRetriesMax}2^{-\alpha_{\mathfrak{D}}} + q_{\textsc{Validate}}2^{-\min\{\mu\lambda,2\lambda,\mu'\lambda,l_2,l_4,l_7\}}$ in total, modeling H$_7$ as a random oracle. □

```
Expt^{SUF-t'}_{ePACA,Compl}(A):
1   $\mathcal{L}_{\textsc{Auth}} \leftarrow \emptyset$
2   win-SUF-t' $\leftarrow 0$
3   $() \xleftarrow{\$} \mathcal{A}^{\mathcal{O}}(1^\lambda)$
4   return win-SUF-t'

bindPartner(T, i):
5   if $\exists(C, j)$ s.th. $\pi^i_T$.sid $= \pi^j_C$.sid
6     return $(C, j)$
7   return $(\bot, \bot)$

Win-SUF-t'(T, i, M, t, d):
8   if $d \neq$ accepted:  return 1
9   if $\exists(C_1, j_1), (C_2, j_2)$ s.t. $(C_1, j_1) \neq (C_2, j_2)$ and $\pi^{j_1}_{C_1}$.st_exe $= \pi^{j_2}_{C_2}$.st_exe =
    bindDone and $\pi^{j_1}_{C_1}$.sid $= \pi^{j_2}_{C_2}$.sid:  return 1
10  if $\exists(T_1, i_1), (T_2, i_2)$ s.t. $(T_1, i_1) \neq (T_2, i_2)$ and $\pi^{i_1}_{T_1}$.st_exe $= \pi^{i_2}_{T_2}$.st_exe =
    bindDone and $\pi^{i_1}_{T_1}$.sid $= \pi^{i_2}_{T_2}$.sid:  return 1
11  $(C, j) \leftarrow$ bindPartner$(T, i)$
12  if $(C, j, M, t) \notin \mathcal{L}_{\textsc{Auth}}$
13    if $(C, j) = (\bot, \bot)$ or $\pi^j_C$.compromised = false
14      if $\pi^i_T$.pinCorr = false: return 1
15  return 0

NewT(T, initialData):
16  if st_T $\neq \bot$: return $\bot$
17  (version, puvProtocolList) $\leftarrow$ initialData
18  st_T.version $\leftarrow$ version
19  st_T.puvProtocolList $\leftarrow$ puvProtocolList
20  Reboot(st_T)
21  return

NewU(U):
22  if pin_U $= \bot$
23    pin_U $\xleftarrow{\$} \mathcal{PIN}$
24  return

CorruptUser(U):
25  corr_U $\leftarrow$ true
26  return pin_U

Send-Bind-T(T, i, m):
27  if st_T $= \bot$ or $\pi^i_T = \bot$ or
    $\pi^i_T$.st_exe $\neq$ waiting or $\pi^i_T$.isValid = false
28    return $\bot$
29  $\pi^i_T$.pinCorr $\leftarrow$ corr_{st_T.user}
30  $m' \xleftarrow{\$}$ Bind-T$(\pi^i_T, m)$
31  $c_{pt}$ || calledReboot $\leftarrow m'$
32  if calledReboot = true
33    foreach $i'$ s.t. $\pi^{i'}_T \neq \bot$
34      $\pi^{i'}_T$.isValid $\leftarrow$ false
35  elseif $\pi^i_T$.st_exe = bindDone
36    foreach $i' \neq i$ and $\pi^i_T \neq \bot$
37      $\pi^{i'}_T$.isValid $\leftarrow$ false
38  return $m'$

Reboot(T):
39  if st_T $= \bot$: return
40  foreach $i$ s.t. $\pi^i_T \neq \bot$
41    $\pi^i_T$.isValid $\leftarrow$ false
42  Reboot(st_T)
43  return

Send-Bind-C(C, j, m):
44  if $\pi^j_C = \bot$ or $\pi^j_C$.st_exe $\neq$
    bindStart or $\pi^j_C$.isValid =
    false
45    return $\bot$
46  return Bind-C$(\pi^j_C, m)$

Setup(T, i, C, j, U):
47  if st_T $= \bot$ or $\pi^i_T \neq \bot$ or $\pi^j_C \neq \bot$
    or pin_U $= \bot$
48    return $\bot$
49  $\pi^i_T \leftarrow$ st_T
50  trans $\xleftarrow{\$}$ Setup$(\pi^i_T, \pi^j_C,$ pin_U$)$
51  $\pi^i_T$.isValid, $\pi^j_C$.isValid $\leftarrow$ false
52  st_T.user $\leftarrow U$
53  return trans

Execute(T, i, C, j, U):
54  if st_T $= \bot$ or $\pi^i_T \neq \bot$ or $\pi^j_C \neq$
    $\bot$ or pin_U $= \bot$
55    return $\bot$
56  $\pi^i_T \leftarrow$ st_T
57  trans, $m_C \leftarrow \bot$
58  while $\pi^j_C$.st_exe $\neq$ bindStart
59    $m_T \xleftarrow{\$}$ Bind-T$(\pi^i_T, m_C)$
60    $m_C \xleftarrow{\$}$ Bind-C$(\pi^j_C, U, m_T)$
61    trans $\leftarrow$ trans $\| m_T \| m_C$
62  return trans

Auth(C, j, M):
63  if $\pi^j_C = \bot$ or $\pi^j_C$.st_exe $\neq$ bindDone
64    return $\bot$
65  $(M, t) \xleftarrow{\$}$ auth-$C(\pi^j_C, M)$
66  $\mathcal{L}_{\textsc{Auth}} \leftarrow \mathcal{L}_{\textsc{Auth}} \cup \{(C, j, M, t)\}$
67  return $(M, t)$

Compromise(C, j):
68  if $\pi^j_C = \bot$ or $\pi^j_C$.st_exe $\neq$
    bindDone: return $\bot$
69  $\pi^j_C$.compromised $\leftarrow$ true
70  return $\pi^j_C$.bs

Validate(T, i, M, t, d):
71  if $\pi^i_T = \bot$ or $\pi^i_T$.st_exe $\neq$ bindDone or $\pi^i_T$.isValid = false
72    return $\bot$
73  status $\leftarrow$ validate-$T(\pi^i_T, M, t, d)$
74  if status = accepted: win-SUF-t' $\leftarrow$ Win-SUF-t'$(T, i, M, t, d)$
75  return status
```

Fig. 6. Security Game for extended PIN-based Access Control Authenticators Protocol for ePACA = (Reboot, Setup, Bind, Auth, Validate), where $\mathcal{O} = \{\textsc{NewT}, \textsc{NewU}, \textsc{Compromise}, \textsc{CorruptUser}, \textsc{Reboot}, \textsc{Setup}, \textsc{Execute}, \textsc{Send-Bind-T}, \textsc{Send-Bind-C}, \textsc{Auth}, \textsc{Validate}\}$ and Compl $\in$ {PPT, QPT}. We highlight differences to the SUF-t security game from [2] in blue.

The above theorem proves that CTAP 2.1 only accepts messages under the user's approval, which is captured by winning condition 1. Winning conditions 2 and 3 capture the uniqueness of each session identifiers: if two sessions are partnered with each other, then they are each other's unique partners. Condition 4 ensures the token only accepts the authorization from a client that it binds to if (1) the binding phase is trusted, (2) the binding state (on the client side) is not compromised if available, and (3) the user PIN that sets up the token is not corrupted.

As is to be expected, the above theorem only holds when the token's user PINs have large enough entropy. If a user PIN is predictable, the attacker can perform active attacks and authorize malicious commands towards the token.

Moving to the security guarantees against quantum adversaries, we note that the asymmetric cryptographic primitives in $\mathsf{puvProtocol}_1$ and $\mathsf{puvProtocol}_2$ are simply ECDH, which is quantum-vulnerable. Therefore, $\mathcal{A}$ can trivially win SUF-t$'$ experiment by selecting the Pin/Uv Auth Protocol in a test session to be $\mathsf{puvProtocol}_1$ or $\mathsf{puvProtocol}_2$. The theorem below suggests the security of the test session if $\mathsf{puvProtocol}_3$ is selected as instantiation.

Theorem 3 (QPT security of CTAP 2.1)**.** *Let* ePACA $=$ (Reboot, Setup, Bind, Auth, Validate) *denote the CTAP 2.1 protocol described in Section V-B. Assume that the underlying* H *is* $\epsilon_{\mathsf{H}}^{\text{coll-res}}$*-collision resistant,* H_5 *is* $\epsilon_{\mathsf{H}_5}^{\text{swap}}$-swap *secure,* H_i *is* $\epsilon_{\mathsf{H}_i}^{\text{prf}}$-prf *secure for* $i \in \{6, 7\}$, SKE_3 *is* $\epsilon_{\mathsf{SKE}_3}^{\text{ind-1cpa}}$-IND-1CPA *and* $\epsilon_{\mathsf{SKE}_3}^{\text{ind-1\$pa-lpc}}$-IND-1\$PA-LPC *secure, and that the* KEM *in* $\mathsf{puvProtocol}_3$ *with public-key entropy* α_{pk} *and ciphertext entropy* α_c *is* $\epsilon_{\mathsf{KEM}}^{\text{ind-cca}}$-IND-CCA *secure. If there exists a* QPT *adversary* $\mathcal{A}$ *that breaks the* SUF-t$'$ *security of* ePACA *for a test session* π *that uses* $\mathsf{puvProtocol}_3$*, then we have that*

$$\begin{aligned}
&\mathsf{Adv}_{\mathsf{ePACA},\mathsf{QPT}}^{\mathsf{SUF\text{-}t'}}(\mathcal{A}) \\
\le&(q_{\textsc{Setup}} + q_{\textsc{Execute}})(\epsilon_{\mathsf{KEM}}^{\text{ind-cca}} + \epsilon_{\mathsf{H}_5}^{\text{swap}} + \epsilon_{\mathsf{H}_6}^{\text{prf}}) \\
&+\binom{q_{\textsc{Setup}} + q_{\textsc{Execute}}}{2} 2^{1-l_6} + \epsilon_{\mathsf{H}}^{\text{coll-res}} + q_{\textsc{NewU}} 2^{-\alpha_{\mathfrak{D}}} \\
&+\binom{q_{\textsc{Send-Bind-T}}}{2} 2^{-\mu'\lambda} + \binom{q_{\textsc{Execute}}}{2}(2^{-\alpha_{pk}} + 2^{-\alpha_c}) \\
&+q_{\textsc{Setup}}\epsilon_{\mathsf{SKE}_3}^{\text{ind-1cpa}} + q_{\textsc{Execute}}\epsilon_{\mathsf{SKE}_3}^{\text{ind-1\$pa-lpc}} \\
&+q_{\textsc{Setup}}\mathsf{pinRetriesMax}2^{-\alpha_{\mathfrak{D}}} + \binom{q_{\textsc{Execute}}}{2}(2^{-\alpha_{pk}} + 2^{-\alpha_c}) \\
&+q_{\textsc{Validate}}(2^{-\mu'\lambda} + \epsilon_{\mathsf{H}_7}^{\text{prf}} + 2^{-l_7})
\end{aligned}$$

where $q_{\mathcal{O}}$ *denotes the number of queries to* $\mathcal{O} = \{$Setup, Execute, Validate$\}$.

Proof Sketch. The proof is similar to the one for Theorem 2 and consists of following steps: (1) By the IND-CCA security of KEM, the swap security of H_5, and the prf security of H_6, all keys K derived in from the encapsulation of the underlying puvProtocol in the obtainSharedSecret-C-end algorithm, which is only invoked in the Setup and Execute oracles, are distinct except probability $(q_{\textsc{Setup}} + q_{\textsc{Execute}})\epsilon_{\mathsf{ECDH}}^{\text{sCDH}} + (q_{\textsc{Setup}} + q_{\textsc{Execute}})(\epsilon_{\mathsf{KEM}}^{\text{ind-cca}} + \epsilon_{\mathsf{H}_5}^{\text{swap}} + \epsilon_{\mathsf{H}_6}^{\text{prf}}) + \binom{q_{\textsc{Setup}}+q_{\textsc{Execute}}}{2}2^{1-l_6}$. (2) By the entropy of the user PIN $\alpha_{\mathfrak{D}}$, none of the user PIN sampled in NewU oracle is predicable except with probability $q_{\textsc{NewU}}2^{-\alpha_{\mathfrak{D}}}$. (3) By the collision-resistance of H and the entropy $2^{-\mu'\lambda}$ of pt values sampled in the Send-Bind-T oracle, we have all H(pin) and pt values respectively distinct except probability in total $\epsilon_{\mathsf{H}}^{\text{coll-res}} + \binom{q_{\textsc{Send-Bind-T}}}{2}2^{-\mu'\lambda}$. (4) By the IND-1CPA security of SKE_3, the pins encrypted by the underlying $\mathsf{puvProtocol}_3$ in the setPIN-C algorithm, which is only invoked in the Setup oracle, are indistinguishable from random except probability $q_{\textsc{Setup}}\epsilon_{\mathsf{SKE}_3}^{\text{ind-1cpa}}$. (5) By the IND-1\$PA-LPC security of SKE_3, the pinHashs encrypted by the underlying $\mathsf{puvProtocol}_3$ in the obtainPinUvAuthToken-C-start algorithm, which is invoked only in the Execute oracle, are indistinguishable from random except probability $q_{\textsc{Execute}}\epsilon_{\mathsf{SKE}_3}^{\text{ind-1\$pa-lpc}}$.

Finally, the adversary $\mathcal{A}$ cannot trigger the flip of the win-SUF-t$'$ predicate in Figure 6 via condition (i) in Line 8, due to the design of CTAP 2.1, see validate-T algorithm in CTAP 2.1, (ii) in Line 9, since the collision of KEM public keys or ciphertexts with entropy α_{pk} or α_c happens at most $\binom{q_{\textsc{Execute}}}{2}(2^{-\alpha_{pk}} + 2^{-\alpha_c})$, (iii) in Line 10, due to the pairwise distinct KEM public keys and pts in the tokens' session identifiers, (iv) in Line 11-14, since $\mathcal{A}$ obtains no information about pins or pts and can only win by randomly guessing the pin in the Setup oracle maximal pinRetriesMax times for each token session, or the pt values or the tags t in the Validate algorithm in the Validate oracle, which happens with probability except $q_{\textsc{Setup}}\mathsf{pinRetriesMax}2^{-\alpha_{\mathfrak{D}}} + q_{\textsc{Validate}}2^{-\mu'\lambda} + \epsilon_{\mathsf{H}_7}^{\text{prf}} + 2^{-l_7}$ in total, assuming the prf security of the underlying H_7. ☐

As such, we suggest to add our PQ instantiation $\mathsf{puvProtocol}_3$ of CTAP 2.1 to the specifications. As mentioned in Section V-C, we also suggest to increase the security parameter from 256 to 512, in order to preserve the current 256-bits level security.

VI. FIDO2 Composition

In this section, we analyze the security of the composition of WebAuthn 2 and CTAP 2.1. To provide a more generalized result, we first define the *user authentication* (ua) security model for the composition of any ePIA and ePACA protocols, which we refer to as ePIA+ePACA. Then, we formally reduce the ua security of ePIA+ePACA to the auth security of the underlying ePIA (see Section IV-C) and the SUF-t$'$ security of the underlying ePACA protocols (see Section V-D). In this section, we respectively use $\bar{\pi}$ and π to denote the ePIA and ePACA session, respectively, to distinguish them clearly.

A. Security Model of ePIA+ePACA

As before, to define the ua security property, we start with describing the trust model, oracles, and winning conditions.

a) Trust Model: The trust model for ua covers both the ones for auth and for SUF-t$'$. Additionally, we assume a server-to-client authenticated channel, which is in practice guaranteed by a TLS connection. As before, we assume "trust-on-first-use", which means, the Setup phase and the initialization of the Bind phase in ePACA and the Register phase in ePIA are authenticated.

b) Oracles: During the execution of the ua experiment, the adversary $\mathcal{A}$ has access to all oracles defined in the SUF-t′ experiment except AUTH and VALIDATE. Furthermore, $\mathcal{A}$ is allowed to query NEWS, NEWTPLA, and CORRUPT from the auth experiment, in addition to the following oracles:

REGISTER$((S,i),(T,j,j'),(C,k),\mathsf{tb},UV,d)$**:** This oracle simulates the honest registration between server S and token T via client C. This oracle is the same as the one in the auth experiment except that after the invocation of $(m_{\mathsf{rcom}}, m_{\mathsf{rcl}}) \leftarrow \mathsf{rCom}(\mathsf{id}_S, m_{\mathsf{rch}}, \mathsf{tb})$, additionally $(m_{\mathsf{rcom}}, t) \leftarrow \textsc{Auth}(C,k,m_{\mathsf{rcom}})$ and $\mathsf{status} \leftarrow \textsc{Validate}(T,j',m_{\mathsf{rcom}},t,d)$ are queried. Moreover, the game aborts if status $\neq$ accepted. Here, AUTH and VALIDATE are defined in the SUF-t′ experiment.

CHALLENGE$((S,i),(C,k),\mathsf{tb},UV)$**:** This oracle simulates the process of the server S generating a challenge nonce and sending it to the client C in an authenticated channel. This oracle is the same as in the auth experiment except that after the invocation of $(m_{\mathsf{rcom}}, m_{\mathsf{rcl}}) \leftarrow \mathsf{rCom}(\mathsf{id}_S, m_{\mathsf{rch}}, \mathsf{tb})$ we additionally query $(m_{\mathsf{rcom}}, t) \leftarrow \textsc{Auth}(C,k,m_{\mathsf{rcom}})$ and $\mathsf{status} \leftarrow \textsc{Validate}(T,j',m_{\mathsf{rcom}},t,d)$, and append tag t to the output.

RESPONSE$((T,j,j'),m_{\mathsf{acom}},t,d)$**:** This oracle simulates the token receiving messages from a client and producing its response. This oracle is the same as the one defined in the auth experiment except that we additionally query $\mathsf{status} \leftarrow \textsc{Validate}(T,j',m_{\mathsf{rcom}},t,d)$, and abort if status $\neq$ accepted.

COMPLETE$((S,i),m_{\mathsf{acl}},m_{\mathsf{arsp}})$**:** This oracle simulates the server verifying the response message and the client message. This oracle is the same as in the auth experiment except that the winning predicate is Win-ua defined in Figure 8.

It is important to note that AUTH and VALIDATE (from the SUF-t′ experiment) are embedded in the REGISTER, CHALLENGE, and RESPONSE oracles in Figure 7.

c) Winning Conditions: We say *user authentication* (ua) holds, if all of the following conditions hold when an ePIA server session $\bar{\pi}_S^i$ accepts a client message m_{acl} and a response message m_{arsp}:

1) The non-$\perp$ session identifiers of the ePIA token (resp., server) sessions do not collide with each other, see Line 37 - 40 in Figure 8.
2) The partnered token and server sessions must have the identical agreed content unless the registration context on the token is corrupted, see Line 42 in Figure 8.
3) The non-$\perp$ session identifiers of the ePACA token (resp., client) sessions that completed Bind, do not collide with each other, see Line 44 - 45 in Figure 8.
4) During registration, the ePIA token and server sessions must partner with each other and the authorized command message and tag must have been output by one of the non-compromised partners of the ePACA token session without corrupting its setup user, see Line 47 - 50 in Figure 8.
5) The token T that has been registered with S, must own an ePIA session $\bar{\pi}_T^i$ that is partnered with $\bar{\pi}_S^i$ and produce a response message unless T's registration context of S is corrupted, see Line 52 - 56 in Figure 8.
6) The above response message must be produced after an ePACA session $\pi_T^{j'}$ validates some authorized command m_{acom} and tag t with the approval from the user, see Line 58 - 58 in Figure 8.
7) The above command m_{acom} and tag t must be authorized by a client ePACA session π_C^k that is partnered with $\pi_T^{j'}$ for some challenge message m_{rch} that has been produced by the ePIA session $\bar{\pi}_S^i$, unless π_C^k is compromised or the PIN that sets up token T has been corrupted, see Line 61 - 66 in Figure 8.

```
$\mathsf{Expt}^{\mathsf{ua}}_{\mathsf{ePIA+ePACA},\mathsf{Compl}}(\mathcal{A})$:
    $\mathcal{L}_{\mathsf{frsh}}, \mathcal{L}_{\textsc{Auth}} \leftarrow \emptyset$ // as in auth and SUF-t′ experiments
    $\mathcal{L}_{\textsc{Register}}, \mathcal{L}_{\textsc{Challenge}}, \mathcal{L}_{\textsc{Response}} \leftarrow \emptyset$
    $\mathsf{win\text{-}ua} \leftarrow \mathsf{false}$
    $() \xleftarrow{\$} \mathcal{A}^{\mathcal{O}}(1^\lambda)$
    return $\mathsf{win\text{-}ua}$

$\textsc{Register}((S,i),(T,j,j'),(C,k),\mathsf{tb},UV,d)$:
    if $\mathsf{pkCP}_S = \perp$ or $\mathsf{suppUV}_T = \perp$ or $\bar{\pi}_S^i \neq \perp$ or $\bar{\pi}_T^j \neq \perp$ or $\mathsf{rc}_T[S] \neq \perp$:
        return $\perp$
    $\bar{\pi}_S^i.\mathsf{pkCP} \leftarrow \mathsf{pkCP}_S$, $\bar{\pi}_T^j.\mathsf{suppUV} \leftarrow \mathsf{suppUV}_T$
    $m_{\mathsf{rch}} \xleftarrow{\$} \mathsf{rChall}(\bar{\pi}_S^i, \mathsf{tb}, UV)$
    $(m_{\mathsf{rcom}}, m_{\mathsf{rcl}}) \xleftarrow{\$} \mathsf{rCom}(\mathsf{id}_S, m_{\mathsf{rch}}, \mathsf{tb})$
    $(m_{\mathsf{rcom}}, t) \leftarrow \textsc{Auth}(C, k, m_{\mathsf{rcom}})$
    $\mathsf{status} \leftarrow \textsc{Validate}(T, j', m_{\mathsf{rcom}}, t, d)$
    if $\mathsf{status} \neq \mathsf{accepted}$: return $(m_{\mathsf{rch}}, m_{\mathsf{rcl}}, m_{\mathsf{rcom}}, t, \perp, \perp)$
    $(m_{\mathsf{rrsp}}, \mathsf{rc}_T) \xleftarrow{\$} \mathsf{rRsp}(\bar{\pi}_T^j, m_{\mathsf{rcom}})$
    $(\mathsf{rc}_S, d') \leftarrow \mathsf{rVrfy}(\bar{\pi}_S^i, m_{\mathsf{rcl}}, m_{\mathsf{rrsp}})$
    $\mathcal{L}_{\mathsf{frsh}} \leftarrow \mathcal{L}_{\mathsf{frsh}} \cup (S, T)$
    $\mathcal{L}_{\textsc{Register}} \leftarrow \mathcal{L}_{\textsc{Register}} \cup \{(S,i,T,j,j',C,k,m_{\mathsf{rch}},m_{\mathsf{rcl}},m_{\mathsf{rcom}},t,m_{\mathsf{rrsp}})\}$
    return $(m_{\mathsf{rch}}, m_{\mathsf{rcl}}, m_{\mathsf{rcom}}, t, m_{\mathsf{rrsp}}, d')$
$\textsc{Challenge}((S,i),(C,k),\mathsf{tb},UV)$:
    if $\mathsf{pkCP}_S = \perp$ or $\bar{\pi}_S^i \neq \perp$: return $\perp$
    $\bar{\pi}_S^i.\mathsf{pkCP} \leftarrow \mathsf{pkCP}_S$
    $m_{\mathsf{ach}} \xleftarrow{\$} \mathsf{aChall}(\bar{\pi}_S^i, \mathsf{tb}, UV)$
    $(m_{\mathsf{acom}}, m_{\mathsf{acl}}) \leftarrow \mathsf{aCom}(\mathsf{id}_S, m_{\mathsf{ach}}, \mathsf{tb})$
    $(m_{\mathsf{acom}}, t) \leftarrow \textsc{Auth}(C, k, m_{\mathsf{acom}})$
    $\mathcal{L}_{\textsc{Challenge}} \leftarrow \mathcal{L}_{\textsc{Challenge}} \cup \{(S,i,C,k,m_{\mathsf{ach}},m_{\mathsf{acl}},m_{\mathsf{acom}},t)\}$
    return $(m_{\mathsf{ach}}, m_{\mathsf{acl}}, m_{\mathsf{acom}}, t)$
$\textsc{Response}((T,j,j'),m_{\mathsf{acom}},t,d)$:
    $\mathsf{status} \leftarrow \textsc{Validate}(T, j', m_{\mathsf{acom}}, t, d)$
    if $\mathsf{status} \neq \mathsf{accepted}$: return $\perp$
    if $\mathsf{suppUV}_T = \perp$ or $\bar{\pi}_T^j \neq \perp$: return $\perp$
    $\bar{\pi}_T^j.\mathsf{suppUV} \leftarrow \mathsf{suppUV}_T$
    $(m_{\mathsf{arsp}}, \mathsf{rc}_T) \xleftarrow{\$} \mathsf{aRsp}(\bar{\pi}_T^j, \mathsf{rc}_T, m_{\mathsf{acom}})$
    $\mathcal{L}_{\textsc{Response}} \leftarrow \mathcal{L}_{\textsc{Response}} \cup \{(T,j,j',m_{\mathsf{acom}},t,d,m_{\mathsf{arsp}})\}$
    return $m_{\mathsf{arsp}}$
$\textsc{Complete}((S,i),m_{\mathsf{acl}},m_{\mathsf{arsp}})$:
    if $\bar{\pi}_S^i = \perp$ or $\bar{\pi}_S^i.\mathsf{st}_{\mathsf{exe}} \neq \mathsf{running}$: return $\perp$
    $(\mathsf{rc}_S, d) \xleftarrow{\$} \mathsf{aVrfy}(\bar{\pi}_S^i, \mathsf{rc}_S, m_{\mathsf{acl}}, m_{\mathsf{arsp}})$
    if $d = 1$: $\mathsf{win\text{-}ua} \leftarrow \mathsf{Win\text{-}ua}(S,i)$
    return $d$
```

Fig. 7. The ua security experiment for ePIA+ePACA. The winning condition Win-ua is defined in Figure 8. The AUTH and VALIDATE oracles are defined in $\mathsf{Expt}^{\mathsf{SUF\text{-}t'}}_{\mathsf{ePACA},\mathsf{Compl}}$ experiment in Figure 6.

Definition 3 (ua security for ePIA+ePACA)**.** *Let* Compl $\in$ $\{$PPT, QPT$\}$, ePACA *be an extended PIN-based access control for authenticators protocol, and* ePIA *be an extended passwordless authentication protocol. We say that the composition*

```
Win-ua(S, i):
//The non-$\perp$ session identifiers of ePIA token (resp. server) sessions do not collide with each other
if $\exists(T_1, j_1), (T_2, j_2)$ s.t. $(T_1, j_1) \neq (T_2, j_2)$ and $\pi_{T_1}^{j_1}.\mathsf{sid} = \pi_{T_2}^{j_2}.\mathsf{sid} \neq \perp$
  return 1
if $\exists(S_1, i_1), (S_2, i_2)$ s.t. $(S_1, i_1) \neq (S_2, i_2)$ and $\pi_{S_1}^{i_1}.\mathsf{sid} = \pi_{S_2}^{i_2}.\mathsf{sid} \neq \perp$
  return 1
//In ePIA, the partnered session have the identical agreed content unless the registration context on the token is corrupted
if $\exists(S', i'), (T', j')$ s.t. $\bar{\pi}_{S'}^{i'}.\mathsf{sid} = \bar{\pi}_{T'}^{j'}.\mathsf{sid} \neq \perp$ and $(S', T') \in \mathcal{L}_{\mathsf{frsh}}$
    and $\bar{\pi}_{S'}^{i'}.\mathsf{agCon} \neq \bar{\pi}_{T'}^{j'}.\mathsf{agCon}$: return 1
//The non-$\perp$ session identifiers of ePACA token (resp. client) sessions that completed Bind algorithm don't collide with each other
if $\exists(C_1, k_1), (C_2, k_2)$ s.t. $(C_1, k_1) \neq (C_2, k_2)$ and $\pi_{C_1}^{k_1}.\mathsf{st}_{\mathsf{exe}} =$
    $= \pi_{C_2}^{k_2}.\mathsf{st}_{\mathsf{exe}} = \mathsf{bindDone}$ and $\pi_{C_1}^{j_1}.\mathsf{sid} = \pi_{C_2}^{j_2}.\mathsf{sid}$: return 1
if $\exists(T_1, j_1'), (T_2, j_2')$ s.t. $(T_1, j_1') \neq (T_2, j_2')$ and $\pi_{T_1}^{j_1'}.\mathsf{st}_{\mathsf{exe}} =$
    $= \pi_{T_2}^{j_2'}.\mathsf{st}_{\mathsf{exe}} = \mathsf{bindDone}$ and $\pi_{T_1}^{j_1'}.\mathsf{sid} = \pi_{T_2}^{j_2'}.\mathsf{sid}$: return 1
//The ePIA or ePACA sessions used in the registration phase must partner with each other.
foreach $(S', x, T', y, y', C', z, m_{\mathsf{rch}}, m_{\mathsf{rcl}}, m_{\mathsf{rcom}}, t_{\mathsf{rcom}}, m_{\mathsf{rrsp}}) \in \mathcal{L}_{\text{REGISTER}}$
  if $\bar{\pi}_{T'}^{y}.\mathsf{sid} \neq \bar{\pi}_{S'}^{x}.\mathsf{sid}$: return 1
  $(C'', z') \leftarrow \mathsf{bindPartner}(T', y')$
  if $(C'', z', m_{\mathsf{rcom}}, t_{\mathsf{rcom}}) \notin \mathcal{L}_{\text{AUTH}}$ and $\Big((C'', z') = (\perp, \perp)$
    or $\pi_{C''}^{z'}.\mathsf{compromised} = \mathsf{false}\Big)$ and $\pi_{T'}^{y'}.\mathsf{pinCorr} = \mathsf{false}$: return 1
//A response message $m_{\mathsf{arsp}}$ must be output by $T$ that registered with $S$, unless $T$'s registration context of $S$ is corrupted
$T \leftarrow \mathsf{regPartner}(S, i)$
if $\nexists j$ s.t. $\bar{\pi}_{S}^{i}.\mathsf{sid} = \bar{\pi}_{T}^{j}.\mathsf{sid}$
  if $(S, T) \in \mathcal{L}_{\mathsf{frsh}}$: return 1
elseif $\nexists(j', m_{\mathsf{acom}}, t, d, m_{\mathsf{arsp}})$ s.t. $(T, j, j', m_{\mathsf{acom}}, t, d, m_{\mathsf{arsp}}) \in \mathcal{L}_{\text{RESPONSE}}$
  return 1
//Above $m_{\mathsf{arsp}}$ must be output under user approval
elseif $d \neq \mathsf{accepted}$: return 1
else
//Above $m_{\mathsf{arsp}}$ must be output after above $T$ validates above message-tag pair $(m_{\mathsf{acom}}, t)$, which encodes $m_{\mathsf{ach}}$ output by session $\bar{\pi}_{S}^{i}$
  $(C, k) \leftarrow \mathsf{bindPartner}(T, j')$
  if $(C, k, m_{\mathsf{acom}}, t) \notin \mathcal{L}_{\text{AUTH}}$
    if $(C, k) = (\perp, \perp)$ or $\pi_{C}^{k}.\mathsf{compromised} = \mathsf{false}$
      if $\pi_{T}^{j'}.\mathsf{pinCorr} = \mathsf{false}$: return 1
  elseif $\nexists(m_{\mathsf{ach}}, m_{\mathsf{acl}})$ s.t. $(S, i, C, k, m_{\mathsf{ach}}, m_{\mathsf{acl}}, m_{\mathsf{acom}}, t) \in \mathcal{L}_{\text{CHALLENGE}}$
    return 1
return 0
```

Fig. 8. The Win-ua in ua security experiment for ePIA+ePACA. The regPartner and bindPartner predicates are defined in Figure 2 and Figure 6, respectively.

ePIA+ePACA *has* user authentication*, or is* ua*-secure for short, if for all* Compl *adversaries* $\mathcal{A}$ *the advantage*

$$\mathsf{Adv}^{\mathsf{ua}}_{\mathsf{ePIA+ePACA,Compl}}(\mathcal{A}) := \Pr[\mathsf{Expt}^{\mathsf{ua}}_{\mathsf{ePIA+ePACA,Compl}}(\mathcal{A}) = 1]$$

in winning the game $\mathsf{Expt}^{\mathsf{ua}}_{\mathsf{ePIA+ePACA,Compl}}$ *as described in Figure 7 is negligible in the security parameter* λ.

We can reduce the security of the ePIA+ePACA protocol to the security of the ePIA and the ePACA protocol as stated in the next theorem. We give the full proof in [4].

Theorem 4 (PPT/QPT security of the composition)**.** *Let* $\mathsf{Compl} \in \{\mathsf{PPT}, \mathsf{QPT}\}$*. Let* Σ *denote an* ePIA *protocol and* Π *denote an* ePACA *protocol. If there exists a* Compl *adversary* $\mathcal{A}$ *that breaks the* ua *security of the composition* $\Sigma + \Pi$*, then there must exist* Compl *adversaries* $\mathcal{A}_1$ *and* $\mathcal{A}_2$ *that respectively break the* auth *security of* Σ *and the* SUF-t′ *security of* Π *such that*

$$\mathsf{Adv}^{\mathsf{ua}}_{\Sigma+\Pi,\mathsf{Compl}}(\mathcal{A}) \leq \mathsf{Adv}^{\mathsf{auth}}_{\Sigma,\mathsf{Compl}}(\mathcal{A}_1) + \mathsf{Adv}^{\mathsf{SUF\text{-}t'}}_{\Pi,\mathsf{Compl}}(\mathcal{A}_2).$$

In particular, the winning condition 1 and 3 capture the uniqueness of each WebAuthn 2 and CTAP 2.1 session identifiers. If two sessions are partnered with each other, then they are each other's unique partners. The winning condition 2 ensures that if the credential private key between the partnered token and server sessions is not corrupted, then both sessions must agree on the server identifier id_S, the $\mathsf{H}(\mathsf{ch}, \mathsf{tb})$ hash of the challenge nonce and the token binding states, the local counter n, and the user presence UP and verification UV conditions. Furthermore, if the underlying hash function H is collision resistant, the token and server sessions also implicitly agree on the token binding state tb. Our theorem proves partnership of WebAuthn 2 sessions in the authenticated registration phase and the resilience of man-in-the-middle attacks against WebAuthn 2 in the authentication phase unless the corruption of the registration context on the token, which are captured by winning conditions 4 and 5. The messages between every partnered token and server session in WebAuthn 2 must be authorized by the client, which is connected to the server over an authenticated channel in WebAuthn 2 and bound to the token in CTAP 2.1 unless the adversary make certain corruptions, which is captured by wining conditions 4,6,7.

VII. Related work

The only published in-depth formal analysis of FIDO2 is Barbosa et al. [2], which we address in-depth. We note that a recently released manuscript [10] also analyzes aspects of FIDO2, but their work focuses on WebAuthn's privacy aspects, and introducing the possibility of revocation, notably in the context of cryptocurrency wallets. Our work is essentially orthogonal to [10] in terms of focus, and we consider the newer versions of both sub-protocols.

To provide context for our comparison to [2], we first revisit the largest changes in CTAP 2.1 compared to CTAP 2.0.

A. Comparison between CTAP 2.0 and CTAP 2.1

Compared to the expired proposed standard of CTAP 2.0 [7], the latest draft review of CTAP 2.1 [6] has a number of differences, mainly from the following four aspects:

1) The definition of CTAP 2.0 is directly based on the concrete primitives such as the Diffie-Hellman key exchange and hash functions, while CTAP 2.1 is based on a so-called "PIN/UV Auth Protocol" abstract scheme, denoted by puvProtocol for short, which leads CTAP 2.1 to be PQ ready. Up to date, two instantiations of puvProtocol are officially announced, where CTAP 2.1 instantiated by the $\mathsf{puvProtocol}_1$ is close to CTAP 2.0. In particular, CTAP 2.1 instantiated with our hybrid construction $\mathsf{puvProtocol}_3$ proposed in Section V-C is provably PQ secure, as proven in Theorem 3.
2) In CTAP 2.0, the binding state that is used for the client's authorization and the token's validation is defined as so-called *pinToken*, which has the length of multiple of 128 bits and can be of unlimited length. In CTAP 2.1, the binding state is defined as so-called *pinUvAuthToken*, the length of which is however fixed: either 128 or 256 bits.
3) In CTAP 2.0, the pinToken is sampled during the reboot phase and then repeatedly re-used until the next invocation of the reboot algorithm. In contrast, the pinUvAuthToken in CTAP 2.1 is one-time – it is re-sampled after every usage. This difference exerts a great influence on the security: While

CTAP 2.0 only satisfies $\mathsf{UF\text{-}t}$ security as proven by Barbosa et al. [2], CTAP 2.1 provably satisfies $\mathsf{SUF\text{-}t}'$ security, see Theorem 2.

4) CTAP 2.0 allows tokens and clients to share a pinUvAuthToken only when the users provide their correct pin, which is called *clientPIN method*. Instead, CTAP 2.1 additionally enables users to input their biometric information such as fingerprint if the built-in on-device user verification is physically supported by the token, which is called *built-in user verification method*. Notably, the built-in user verification method is always the preferred option when it is supported by the token. The biometric information is assumed to be unique and unpredictable for each user and is input to the token without any intermediary (therefore, the transmission can be considered to be authenticated). In our model, built-in user verification can be viewed as the simplified CTAP 2.1 using clientPIN method without the transmission of the encryption of pinHash.

B. Comparison with Barbosa et al. [2]

As mentioned before, our work builds on the first formal FIDO2 analysis in [2], and we compare several aspects.

WebAuthn comparison:

1) **Different analysis target**: The analysis of [2] assumes attestation type `Basic` such that "the server is assumed to know the attestation public key that uniquely identifies the authenticator" [2]. However, the token's attestation key pair is generated in the factory and at least 100,000 tokens should share same attestation key pair to ensure privacy ([1, Section 14.4], [11, Section 14.4.1]). Thus, Barbosa et al.'s assumption does not hold in practice. In contrast, we investigate WebAuthn with the default attestation type `None`, and our Theorem 1 also applies to WebAuthn with attestation type `Basic`.
2) **Fine-grained abstraction**: Our WebAuthn abstraction is more detailed than [2]. For example, we include the supported signature list pkCP of the server, the optional UV-support of the token, and the token binding state $\mathtt{tb}$. Our theorem implies that the server and token ultimately agree on these values, which is crucial for the desired security. Furthermore, the supported schemes list enables us to to exhibit a downgrade attack against WebAuthn and specify a security notion "Algorithm Agreement" for the corresponding protection.
3) **Active interference**: The security model of WebAuthn in [2] seems to allow active interference during the registration. This is true in [2]'s model because it assumes that each token has a unique attestation key pair and the server knows in advance which public key to use for signature verification; yet this is not true in practice by design, as mentioned previously. The official specification [11, Section 13.4.4] clearly acknowledges the MitM attack on registration, contradicting the implication of [2].
4) **Stronger adversary capability**: Barbosa et al. assume the tokens to be tamper-proof, i.e., the adversary is prevented from corrupting the internal state of any token. Our model, instead, includes a corruption oracle that enables an adversary to reveal the private signing key, capturing the real world scenario in which some tokens might be stolen and the private keys compromised.

CTAP comparison:

1) **Different analysis target**: Barbosa et al. analyzed CTAP 2.0 [7], while we investigate CTAP 2.1 [6]. As explained in Section VII-A, these two versions have numerous differences. Our paper carefully explores the abstraction gaps between CTAP 2.0 and CTAP 2.1.
2) **Improved security model**: We refine the Barbosa et al. PACA security model. For example, the token binding states may be reset in REBOOT or SEND oracle. However, Barbosa et al. only mark the token sessions invalid in the REBOOT oracle but forgot the ones in the SEND oracle[8]. Furthermore, the PACA definition of invalidity is not suitable for CTAP 2.1, as the previous binding states of a token are reset after not only reboot but also the establishment of a new session. In this work, we define a code-based $\mathsf{SUF\text{-}t}'$ security, which refines and generalizes $\mathsf{SUF\text{-}t}$ security in [2].
3) **Proof gaps**: Although Barbosa et al. proved the security of CTAP 2.0, their proof has several technical gaps. To address this, we base the $\mathsf{SUF\text{-}t}'$ security of CTAP 2.1 on novel assumptions and provide a detailed proof. We summarize the gaps and shortcomings of the proofs from [2] in [4], and show how we solve each for our work.

The Composition of WebAuthn and CTAP:

1) **Different security model**: The security of the composition of WebAuthn 2 and CTAP 2.1 relies on the respective security guarantees. The differences between the syntax and the security models of both WebAuthn and CTAP compared to [2] propagate into a different security model for the composition, and we provide a fully detailed proof.

VIII. Limitations and Future Work

While our work covers many core aspects of CTAP and WebAuthn beyond the state-of-the-art, it remains an abstraction. Some of our main current limitations include that we do not yet model some of the new CTAP 2.1 features for enterprise customers, and do not make formal statements about the unlinkability of credentials or other detailed privacy statements. We leave the proof methodology for tighter upper bounds in all theorems in our paper as an open question.

Acknowledgments

We are very grateful to Jacqueline Brendel for initiating the project, initial analysis of the standards, and valuable support throughout.

[8]Recall that [2] defines the invalidity of a session such that "*if a token is rebooted, its binding states got reset and hence become invalid*" [2]

References

[1] Dirk Balfanz, Alexei Czeskis, Jeff Hodges, J.C. Jones, Michael B. Jones, Akshay Kumar, Angelo Liao, Rolf Lindemann, Emil Lundberg, Vijay Bharadwaj, Arnar Birgisson, Hubert Le Van Gong, Christiaan Brand, Langley Adam, Giridhar Mandyam, Mike West, and Jeffrey Yasskin. *Web authentication: An API for accessing public key credentials level 1 – W3C recommendation*. https://www.w3.org/TR/2019/REC-webauthn-1-20190304/. March 2019.

[2] Manuel Barbosa, Alexandra Boldyreva, Shan Chen, and Bogdan Warinschi. "Provable Security Analysis of FIDO2". In: *CRYPTO 2021, Part III*. Vol. 12827. LNCS. Virtual Event: Springer, Heidelberg, Aug. 2021.

[3] Mihir Bellare, Anand Desai, Eric Jokipii, and Phillip Rogaway. "A Concrete Security Treatment of Symmetric Encryption". In: *38th FOCS*. IEEE Computer Society Press, Oct. 1997.

[4] Nina Bindel, Cas Cremers, and Mang Zhao. *FIDO2, CTAP 2.1, and WebAuthn 2: Provable Security and Post-Quantum Instantiation*. Cryptology ePrint Archive, Paper 2022/1029. https://eprint.iacr.org/2022/1029. 2022.

[5] Nina Bindel, Udyani Herath, Matthew McKague, and Douglas Stebila. "Transitioning to a Quantum-Resistant Public Key Infrastructure". In: *Post-Quantum Cryptography - 8th International Workshop, PQCrypto 2017*. Springer, Heidelberg, 2017.

[6] John Bradley, Jeff Hodges, Michael B. Jones, Akshay Kumar, Rolf Lindemann, Johan Verrept, Chad Armstrong, Konstantinos Georgantas, Fabian Kaczmarczyck, Nina Satragno, and Nuno Sung. *Client to Authenticator Protocol (CTAP) – Proposed Standard, June 15, 2021*. https://fidoalliance.org/specs/fido-v2.1-ps-20210615/fido-client-to-authenticator-protocol-v2.1-ps-20210615.html. 2021.

[7] Christiaan Brand, Alexei Czeskis, Ehrensvärd Jakob, Michael B. Jones, Akshay Kumar, Rolf Lindemann, Adam Powers, and Johan Verrept. *Client to Authenticator Protocol (CTAP) – Proposed Standard, January 30, 2019*. https://fidoalliance.org/specs/fido-v2.0-ps-20190130/fido-client-to-authenticator-protocol-v2.0-ps-20190130.html. 2019.

[8] Lily Chen, Stephen Jordan, Yi-Kai Liu, Dustin Moody, Rene Peralta, Ray Perlner, and Daniel Smith-Tone. *NISTIR 8105 Report on Post-Quantum Cryptography*. Tech. rep. National Institute for Standards and Technology (NIST), 2016.

[9] William F Ehrsam, Carl HW Meyer, John L Smith, and Walter L Tuchman. *Message verification and transmission error detection by block chaining*. US Patent 4,074,066. 1978.

[10] Lucjan Hanzlik, Julian Loss, and Benedikt Wagner. *Token meets Wallet: Formalizing Privacy and Revocation for FIDO2*. Cryptology ePrint Archive, Report 2022/084. https://ia.cr/2022/084. 2022.

[11] Jeff Hodges, J.C. Jones, Michael B. Jones, Akshay Kumar, Emil Lundberg, John Bradley, Christiaan Brand, Langley Adam, Giridhar Mandyam, Nina Satragno, Nick Steele, Jiewen Tan, Shane Weeden, Mike West, and Jeffrey Yasskin. *Web authentication: An API for accessing public key credentials level 2 – W3C recommendation*. https://www.w3.org/TR/2021/REC-webauthn-2-20210408/. April 2021.

[12] B. Kaliski, J. Jonsson, and A. Rusch. *PKCS #1: RSA Cryptography Specifications Version 2.2*. RFC 8017 (Informational). Internet Engineering Task Force, Nov. 2016.

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

Appendix A
Preliminaries

Definition 4. *We say* $\mathsf{F} : \mathcal{K} \times \mathcal{M} \to \mathcal{O}$ *is* ϵ-prf *secure, if for any efficient adversaries* $\mathcal{A}$*, any* $k \xleftarrow{\$} \mathcal{K}$*, and any truly random function* $\mathsf{R} : \mathcal{M} \to \mathcal{O}$*, it holds that,*

$$\mathsf{Adv}^{\mathsf{prf}}_{\mathsf{F},\mathcal{A}} := \left|\Pr[\mathcal{A}^{\mathsf{F}(k,\cdot)} = 1] - \Pr[\mathcal{A}^{\mathsf{R}(\cdot)} = 1]\right| \leq \epsilon$$

We say F *is* ϵ-swap *secure, if the function* $\bar{\mathsf{F}}$ *defined below is* ϵ-prf *secure.*

$$\bar{\mathsf{F}} : \mathcal{M} \times \mathcal{K} \to \mathcal{O}, \bar{\mathsf{F}}(m, k) := \mathsf{F}(k, m)$$

Definition 5. *Let* $\mathsf{SKE} = (\mathsf{KG}, \mathsf{Enc}, \mathsf{Dec})$ *be a symmetric key encryption scheme with symmetric key space* $\mathcal{K}$*. We say* SKE *is* ϵ*-one time* IND-CPA *secure with respect to function* H *(denoted by* IND-1CPA-H*) secure, if the blow defined advantage of every (potential quantum) adversary* $\mathcal{A}$ *against* $\mathsf{Expt}^{\text{IND-1CPA-H}}_{\mathsf{SKE}}$ *experiment in Figure 9 is bounded by,*

$$\mathsf{Adv}^{\text{IND-1CPA-H}}_{\mathsf{SKE}}(\mathcal{A}) := \left|\Pr[\mathsf{Expt}^{\text{IND-1CPA-H}}_{\mathsf{SKE}}(\mathcal{A}) = 1] - \frac{1}{2}\right| \leq \epsilon.$$

Definition 6. *Let* $\mathsf{SKE} = (\mathsf{KG}, \mathsf{Enc}, \mathsf{Dec})$ *be a symmetric key encryption scheme with symmetric key space* $\mathcal{K}$*. We say* SKE *is* ϵ-IND-1\$PA-LPC *secure, if the blow defined advantage of every (potential quantum) adversary* $\mathcal{A}$ *against* $\mathsf{Expt}^{\text{IND-1\$PA-LPC}}_{\mathsf{SKE}}$ *experiment in Figure 9 is bounded by,*

$$\mathsf{Adv}^{\text{IND-1\$PA-LPC}}_{\mathsf{SKE}}(\mathcal{A}) := \left|\Pr[\mathsf{Expt}^{\text{IND-1\$PA-LPC}}_{\mathsf{SKE}}(\mathcal{A}) = 1] - \frac{1}{2}\right| \leq \epsilon.$$

```
$\mathsf{Expt}^{\text{IND-1CPA-H}}_{\mathsf{SKE}}(\mathcal{A})$:
    $\mathsf{b} \xleftarrow{\$} \{0,1\}$
    $K \xleftarrow{\$} \mathsf{KG}()$
    $(m_0^\star, m_1^\star) \xleftarrow{\$} \mathcal{A}()$
    if $|m_0^\star| \neq |m_1^\star|$
        return $0$
    $c^\star \xleftarrow{\$} \mathsf{Enc}(K, m_{\mathsf{b}}^\star)$
    $t^\star \leftarrow \mathsf{H}(K, c^\star)$
    $\mathsf{b}' \xleftarrow{\$} \mathcal{A}(c^\star, t^\star)$
    return $[\![\mathsf{b} = \mathsf{b}']\!]$

$\mathsf{Expt}^{\text{IND-1\$PA-LPC}}_{\mathsf{SKE}}(\mathcal{A})$:
    $\mathsf{b} \xleftarrow{\$} \{0,1\}$
    $K \xleftarrow{\$} \mathsf{KG}()$
    $(m_0^\star, m_1^\star) \xleftarrow{\$} \mathcal{A}()$
    if $|m_0^\star| \neq |m_1^\star|$
        return $0$
    $c^\star \xleftarrow{\$} \mathsf{Enc}(K, m_{\mathsf{b}}^\star)$
    $\mathsf{b}' \xleftarrow{\$} \mathcal{A}^{\textsc{Rand},\textsc{Lpc}}(c^\star)$
    return $[\![\mathsf{b} = \mathsf{b}']\!]$

$\textsc{Rand}(l)$:
    $m_0' \xleftarrow{\$} \{0,1\}^l$
    $m_1' \xleftarrow{\$} \{0,1\}^l$
    $c' \xleftarrow{\$} \mathsf{Enc}(K, m_{\mathsf{b}}')$
    return $(m_0', m_1', c')$
$\textsc{Lpc}(c)$:
    if $c = c^\star$
        return $0$
    return $[\![m_0^\star = \mathsf{Dec}(K, c)]\!]$
```

Fig. 9. IND-1CPA-H and IND-1\$PA-LPC experiments for $\mathsf{SKE} = (\mathsf{KG}, \mathsf{Enc}, \mathsf{Dec})$.

Appendix B
CBC Mode and IND-1\$PA-LPC Security

The Cipher Block Chaining (CBC) mode is a block cipher mode of operation invented by Ehrsam et al. in 1976 [9]. The CBC can be divided into two categories: CBC_0, whose initial vector is a string of zero bits, and CBC_R, whose initial vector is a random bit string. We first recall CBC as an instance of symmetric key encryption. Let $\mathcal{K} := \{0,1\}^{f_1(\lambda)}$, $\mathcal{M} := \{0,1\}^{f_2(\lambda)}$, and $\mathcal{O} := \{0,1\}^{f_2(\lambda)}$ respectively denote the symmetric key space, message space, and output space of an invertible function $\mathsf{F} : \mathcal{K} \times \mathcal{M} \to \mathcal{O}$, where f_1 and f_2 denote arbitrary polynomial functions. Then, both CBC_0 and CBC_R are defined in Figure 10. Here, we simply assume that the input message m of the encryption algorithm always has the length of a multiple of $f_2(\lambda)$. It is straightforward that CBC_0 is a deterministic encryption scheme.

The IND-1\$PA security of the deterministic CBC_0 was proven by Barbosa et al. [2]. Moreover, the IND-CPA security of the randomized CBC_R was proven by Bellare et al. [3]. Below, we prove the IND-1\$PA-LPC security of both CBC_0 and CBC_R based on above two security conclusions.

Theorem 5 (IND-CPA $\implies$ IND-1\$PA)**.** *Let* $\mathsf{SKE} = (\mathsf{KG}, \mathsf{Enc}, \mathsf{Dec})$ *denote a symmetric encryption scheme. If* SKE *is* $\epsilon^{\text{ind-cpa}}_{\mathsf{SKE}}$-IND-CPA *secure, then* SKE *is* $\epsilon^{\text{ind-1\$pa}}_{\mathsf{SKE}}$-IND-1\$PA *secure such that* $\epsilon^{\text{ind-1\$pa}}_{\mathsf{SKE}} \leq \epsilon^{\text{ind-cpa}}_{\mathsf{SKE}}$.

Theorem 6 (IND-1\$PA $\implies$ IND-1\$PA-LPC)**.** *Let* $\mathsf{SKE} = (\mathsf{KG}, \mathsf{Enc}, \mathsf{Dec})$ *denote* CBC_0 *or* CBC_R*. If* SKE *is* $\epsilon^{\text{ind-1\$pa}}_{\mathsf{SKE}}$-IND-1\$PA *secure and the underlying function* $\mathsf{F} : \{0,1\}^{f_1(\lambda)} \times$

```
$\mathsf{KG}(1^\lambda)$:
$K \xleftarrow{\$} \mathcal{K}$
return $K$
$\mathsf{Enc}(K, m)$:
$x_1 \| \ldots \| x_n \leftarrow m$ s.t. $|x_i| = f_2(\lambda)\ \forall i \in \{1, \cdots, n\}$
$y_0 \xleftarrow{\$} \mathsf{SetIV}()$
for $i = 1, \ldots, n$
    $y_i \leftarrow \mathsf{F}(K, y_{i-1} \oplus x_i)$
$y \leftarrow y_0 \| \cdots \| y_n$
return $y$

$\mathsf{Dec}(K, c)$:
$y_0 \| \ldots \| y_n \leftarrow c$ s.t. $|y_i| = f_2(\lambda)\ \forall i \in \{0, \cdots, n\}$
for $i = 1, \ldots, n$
    $x_i \leftarrow y_{i-1} \oplus \mathsf{F}^{-1}(K, y_i)$
$m \leftarrow x_1 \| \cdots \| x_n$
return $m$
```

Fig. 10. CBC mode $\mathsf{SKE} = (\mathsf{KG}, \mathsf{Enc}, \mathsf{Dec})$ with symmetric key space $\mathcal{K} := \{0,1\}^{f_1(\lambda)}$ for arbitrary polynomial function f_1. If $\mathsf{SKE} = \mathsf{CBC}_0$, then $\mathsf{SetIV}()$ outputs a string of zero bits of length $f_2(\lambda)$. If $\mathsf{SKE} = \mathsf{CBC}_R$, then $\mathsf{SetIV}()$ outputs a random string of length $f_2(\lambda)$.

$\{0,1\}^{f_2(\lambda)} \rightarrow \{0,1\}^{f_2(\lambda)}$ *is* $\epsilon_{\mathsf{F}}^{\mathsf{prp}}$-prp *secure, then* SKE *is* $\epsilon_{\mathsf{SKE}}^{\mathsf{ind\text{-}1\$pa\text{-}lpc}}$-IND-1\$PA-LPC *secure such that*

$$\epsilon_{\mathsf{SKE}}^{\mathsf{ind\text{-}1\$pa\text{-}lpc}} \le 2\epsilon_{\mathsf{F}}^{\mathsf{prp}} + q_{\text{LPC}} 2^{-f_2(\lambda)} + q_{\text{RAND}} \lceil \frac{l_{\max}}{f_2(\lambda)} \rceil 2^{-f_2(\lambda)} + \epsilon_{\mathsf{SKE}}^{\mathsf{ind\text{-}1\$pa}}$$

where $q_{\mathcal{O}}$ *denotes the maximal number of queries to* $\mathcal{O} \in \{\text{RAND}, \text{LPC}\}$ *oracles and* $l_{\max}$ *denotes the maximal input to the* RAND *oracle.*

Appendix C
Detailed Description of WebAuthn 2

In Figure 11, the security parameter $\lambda = 128$. For each server S, the associated identifier id_S is its effective domain. The official supported signature algorithms are RSASSA–PKCS1–v1_5 and RSASSA–PSS. As discussed in Section IV-D, the list of signature schemes can be extended by PQ compatible hybrid signature scheme. The underlying hash function H is SHA-256. We assume that each token has a unique user and can be registered at most once per server. The $\mathsf{Register} = (\mathsf{rChall}, \mathsf{rCom}, \mathsf{rRsp}, \mathsf{rVrfy})$ sub-protocol is executed as follows.

- $\mathsf{rChall}(\pi_S^i, \mathsf{tb}, UV)$: The server S samples a random challenge nonce $\pi_S^i.\mathsf{ch}$ and a user identifier $\pi_S^i.\mathsf{uid}$ and initializes the token biding state $\pi_S^i.\mathsf{tb}$ and user verification condition $\pi_S^i.UV$. Finally, S sets $\pi_S^i.\mathsf{st_{exe}}$ to running and outputs a challenge message, see Line 3.
- $\mathsf{rCom}(\mathsf{id}_S, m_{\mathsf{rch}}, \mathsf{tb})$: The client parses m_{rch} into a server identifier id, a challenge nonce ch, a user identifier uid, a supported signature list pkCP and a user verification condition UV. Next, the client aborts if $\mathsf{id} \neq \mathsf{id}_S$. Otherwise, the client sets the user presence condition UP to true and computes the hash h of client message m_{rcl}, which is defined in Line 8. Finally, the client outputs the client and command messages m_{rcl} and m_{rcom}, respectively, see Line 10.
- $\mathsf{rRsp}(\pi_T^j, m_{\mathsf{rcom}})$: The token T first parses m_{rcom} into a server identifier id, a user identifier uid, a hash value h, a signature list pkCP, and the user presence and user verification conditions UP and UV, respectively. Next, T picks one supported signature scheme Σ in pkCP with the highest preference, i.e., with the smallest index possible. Afterwards, T checks whether it can support the required user verification condition UV. If either step fails, the token aborts. Otherwise, T generates a public-private key pair using the key generation algorithm of Σ, initializes the counter n to 0, samples a random credential identifier cid, and sets its execution state to accepted. Finally, T extends the registration context as in Line 20, and outputs it together with a response message m_{rrsp}, as defined in Line 18. The agreed content includes the server identifier id, the hash value h, the credential identifier cid, the counter n, the list pkCP, the public key pk, the signature scheme Σ, and the user presence UP and verification UV conditions. The session identifier is the tuple of the hash of server identifier id, the credential identifier cid, and the counter n.
- $\mathsf{rVrfy}(\pi_S^i, m_{\mathsf{rcl}}, m_{\mathsf{rrsp}})$: The server S parses the client message m_{rcl} and the response message m_{rrsp} and executes a few checks as in Line 26. It outputs abort and decision $d = 0$ if any check fails. Otherwise, S sets the execution state to accepted. Finally, S extends the registration context as in Line 28 and outputs it together with decision $d = 1$. The agreed content and the session identifier are defined as the ones in the rRsp algorithm.

$\mathsf{Authenticate} = (\mathsf{aChall}, \mathsf{aCom}, \mathsf{aRsp}, \mathsf{aVrfy})$ is defined next.

- $\mathsf{aChall}(\pi_S^i, \mathsf{tb}, UV)$: The server S samples a random challenge nonce $\pi_S^i.\mathsf{ch}$ and initializes its token binding state $\pi_S^i.\mathsf{tb}$ and user condition $\pi_S^i.UV$. Finally, S sets π_S^i to running and outputs a challenge message, see Line 34.
- $\mathsf{aCom}(\mathsf{id}_S, m_{\mathsf{ach}}, \mathsf{tb})$: The client parses m_{ach} into an identifier id, a challenge nonce ch, and user verification condition UV. Next, the client aborts if $\mathsf{id} \neq \mathsf{id}_S$. Otherwise, the client sets the user presence condition UP to true and compute the hash h of the client message m_{acl}, which is defined in Line 39. Finally, the client outputs the client message m_{acl} and command message m_{acom}, see Line 41.
- $\mathsf{aRsp}(\pi_T^j, \mathsf{rc}_T, m_{\mathsf{acom}})$: The token T first parses the command message m_{acom} into a server identifier id, a hash value h, and user presence and user verification conditions UP and UV. Next, T checks whether the corresponding registration context exists and whether it can satisfy the user verification requirement. T aborts if either of the above steps fails. Then, T increments the counter $\mathsf{rc}_T[\mathsf{id}].n$ by 1 and defines the associated data ad that includes the hash of id, the counter $\mathsf{rc}_T[\mathsf{id}].n$, and the conditions UP and UV, followed by computing a signature σ on ad and h using the signing key $\mathsf{rc}_T[\mathsf{id}].sk$. Finally, T sets its execution state to accepted and outputs the response message m_{arsp} defined in Line 49 along with rc_T. The agreed context is defined as the tuple of the server identifier id, the value h, the counter $\mathsf{rc}_T[\mathsf{id}].n$, and the user conditions UV and UP. The session identifier is defined as the tuple of the hash of the server identifier id, the credential identifier $\mathsf{rc}_T[\mathsf{id}].\mathsf{cid}$, the hash value h, and the counter n.
- $\mathsf{aVrfy}(\pi_S^i, \mathsf{rc}_S, m_{\mathsf{acl}}, m_{\mathsf{arsp}})$: The server S parses the client message m_{acl} and the response message m_{arsp} and executes checks as in Line 58 if the corresponding registration context exists. It aborts and produces decision $d = 0$ if any check

Register

```
rChall($\pi_S^i$, tb, $UV$): // 1. Server
    $\pi_S^i.\mathsf{ch} \xleftarrow{\$} \{0,1\}^{\geq\lambda}$, $\pi_S^i.\mathsf{tb} \leftarrow \mathsf{tb}$, $\pi_S^i.UV \leftarrow UV$
    $\pi_S^i.\mathsf{uid} \xleftarrow{\$} \{0,1\}^{\leq 4\lambda}$
    $m_{\mathsf{rch}} \leftarrow (\mathsf{id}_S, \pi_S^i.\mathsf{ch}, \pi_S^i.\mathsf{uid}, \pi_S^i.\mathsf{pkCP}, \pi_S^i.UV)$
    $\pi_S^i.\mathsf{st}_{\mathsf{exe}} \leftarrow \mathsf{running}$
    return $m_{\mathsf{rch}}$
rCom($\mathsf{id}_S$, $m_{\mathsf{rch}}$, tb): // 2. Client
    $(\mathsf{id}, \mathsf{ch}, \mathsf{uid}, \mathsf{pkCP}, UV) \leftarrow m_{\mathsf{rch}}$
    if $\mathsf{id} \neq \mathsf{id}_S$: return $\perp$
    $m_{\mathsf{rcl}} \leftarrow (\mathsf{ch}, \mathsf{tb})$
    $UP \leftarrow \mathsf{true}$, $h \leftarrow \mathsf{H}(m_{\mathsf{rcl}})$
    $m_{\mathsf{rcom}} \leftarrow (\mathsf{id}, \mathsf{uid}, h, \mathsf{pkCP}, UP, UV)$
    return $(m_{\mathsf{rcom}}, m_{\mathsf{rcl}})$
rRsp($\pi_T^j$, $m_{\mathsf{rcom}}$): // 3. Token
    $(\mathsf{id}, \mathsf{uid}, h, \mathsf{pkCP}, UP, UV) \leftarrow m_{\mathsf{rcom}}$
    if at least one algorithm in pkCP is supported
        $\Sigma \leftarrow \mathsf{pkCP}[i]$ with smallest $i$ possible
    else return $(\perp, \perp)$
    if $\pi_T^j.\mathsf{suppUV} = \mathsf{false}$ and $UV = \mathsf{true}$: return $(\perp, \perp)$
    $(pk, sk) \xleftarrow{\$} \Sigma.\mathsf{KG}(1^\lambda)$, $\mathsf{cid} \xleftarrow{\$} \{0,1\}^{\geq\lambda}$, $n \leftarrow 0$
    $m_{\mathsf{rrsp}} \leftarrow (\mathsf{H}(\mathsf{id}), n, \mathsf{cid}, pk, \Sigma, UP, UV)$
    $\boxed{\mathsf{h_{CP}} \leftarrow \mathsf{H}(\mathsf{pkCP})}$
    $\mathsf{rc}_T[\mathsf{id}] \leftarrow (\mathsf{uid}, \mathsf{cid}, sk, n, \Sigma, \boxed{\mathsf{h_{CP}}})$
    $\pi_T^j.\mathsf{agCon} \leftarrow (\mathsf{id}, h, \mathsf{cid}, n, \mathsf{pkCP}, pk, \Sigma, UV, UP)$
    $\pi_T^j.\mathsf{sid} \leftarrow (\mathsf{H}(\mathsf{id}), \mathsf{cid}, n)$
    $\pi_T^j.\mathsf{st}_{\mathsf{exe}} \leftarrow \mathsf{accepted}$
    return $(m_{\mathsf{rrsp}}, \mathsf{rc}_T)$
rVrfy($\pi_S^i$, $m_{\mathsf{rcl}}$, $m_{\mathsf{rrsp}}$): // 4. Server
    $(\mathsf{ch}, \mathsf{tb}) \leftarrow m_{\mathsf{rcl}}$, $(h, n, \mathsf{cid}, pk, \Sigma, UP, UV) \leftarrow m_{\mathsf{rrsp}}$
    if $h \neq \mathsf{H}(\mathsf{id}_S)$ or $n \neq 0$ or $\mathsf{ch} \neq \pi_S^i.\mathsf{ch}$ or $\mathsf{tb} \neq \pi_S^i.\mathsf{tb}$ or $\Sigma \notin \pi_S^i.\mathsf{pkCP}$
    or $UP \neq \mathsf{true}$ or $UV \neq \pi_S^i.UV$: return $(\perp, 0)$
    $\boxed{\mathsf{h_{CP}} \leftarrow \mathsf{H}(\pi_S^i.\mathsf{pkCP})}$
    $\mathsf{rc}_S[\mathsf{cid}] \leftarrow (\pi_S^i.\mathsf{uid}, pk, n, \Sigma, \boxed{\mathsf{h_{CP}}})$
    $\pi_S^i.\mathsf{agCon} \leftarrow (\mathsf{id}_S, \mathsf{H}(m_{\mathsf{rcl}}), \mathsf{cid}, n, \pi_S^i.\mathsf{pkCP}, pk, \Sigma, UV, UP)$
    $\pi_S^i.\mathsf{sid} \leftarrow (\mathsf{H}(\mathsf{id}), \mathsf{cid}, n)$
    $\pi_S^i.\mathsf{st}_{\mathsf{exe}} \leftarrow \mathsf{accepted}$
    return $(\mathsf{rc}_S, 1)$
```

Authenticate

```
aChall($\pi_S^i$, tb, $UV$): // 1. Server
    $\pi_S^i.\mathsf{ch} \xleftarrow{\$} \{0,1\}^{\geq\lambda}$, $\pi_S^i.\mathsf{tb} \leftarrow \mathsf{tb}$, $\pi_S^i.UV \leftarrow UV$
    $m_{\mathsf{ach}} \leftarrow (\mathsf{id}_S, \pi_S^i.\mathsf{ch}, \pi_S^i.UP, \pi_S^i.UV)$
    $\pi_S^i.\mathsf{st}_{\mathsf{exe}} \leftarrow \mathsf{running}$
    return $m_{\mathsf{ach}}$
aCom($\mathsf{id}_S$, $m_{\mathsf{ach}}$, tb): // 2. Client
    $(\mathsf{id}, \mathsf{ch}, UV) \leftarrow m_{\mathsf{ach}}$
    if $\mathsf{id} \neq \mathsf{id}_S$: return $\perp$
    $m_{\mathsf{acl}} \leftarrow (\mathsf{ch}, \mathsf{tb})$
    $UP \leftarrow \mathsf{true}$, $h \leftarrow \mathsf{H}(m_{\mathsf{acl}})$
    $m_{\mathsf{acom}} \leftarrow (\mathsf{id}, h, UP, UV)$
    return $(m_{\mathsf{acom}}, m_{\mathsf{acl}})$
aRsp($\pi_T^j$, $\mathsf{rc}_T$, $m_{\mathsf{acom}}$): // 3. Token
    $(\mathsf{id}, h, UP, UV) \leftarrow m_{\mathsf{acom}}$
    if $\mathsf{rc}_T[\mathsf{id}] = \perp$: return $(\perp, \mathsf{rc}_T)$
    if $\pi_T^j.\mathsf{suppUV} = \mathsf{false}$ and $UV = \mathsf{true}$: return $(\perp, \mathsf{rc}_T)$
    $\mathsf{rc}_T[\mathsf{id}].n \leftarrow \mathsf{rc}_T[\mathsf{id}].n + 1$
    $ad \leftarrow (\mathsf{H}(\mathsf{id}), \mathsf{rc}_T[\mathsf{id}].n, UP, UV)$
    $\sigma \xleftarrow{\$} \mathsf{rc}_T[\mathsf{id}].\Sigma.\mathsf{Sign}(\mathsf{rc}_T[\mathsf{id}].sk, (ad, h))$
    $m_{\mathsf{arsp}} \leftarrow (\mathsf{rc}_T[\mathsf{id}].\mathsf{cid}, ad, \boxed{\mathsf{rc}_T[\mathsf{id}].\mathsf{h_{CP}},}\ \sigma, \mathsf{rc}_T[\mathsf{id}].\mathsf{uid})$
    $\pi_T^j.\mathsf{agCon} \leftarrow (\mathsf{id}, h, \mathsf{rc}_T[\mathsf{id}].n, \boxed{\mathsf{rc}_T[\mathsf{id}].\mathsf{h_{CP}},}\ UV, UP)$
    $\pi_T^j.\mathsf{sid} \leftarrow (\mathsf{H}(\mathsf{id}), \mathsf{rc}_T[\mathsf{id}].\mathsf{cid}, h, n)$
    $\pi_T^j.\mathsf{st}_{\mathsf{exe}} \leftarrow \mathsf{accepted}$
    return $(m_{\mathsf{arsp}}, \mathsf{rc}_T)$
aVrfy($\pi_S^i$, $\mathsf{rc}_S$, $m_{\mathsf{acl}}$, $m_{\mathsf{arsp}}$): // 4. Server
    $(\mathsf{ch}, \mathsf{tb}) \leftarrow m_{\mathsf{acl}}$, $(\mathsf{cid}, ad, \boxed{\mathsf{h_{CP}},}\ \sigma, \mathsf{uid}) \leftarrow m_{\mathsf{arsp}}$
    $(h, n, UP, UV) \leftarrow ad$
    if $\mathsf{rc}_S[\mathsf{cid}] = \perp$: return $(\mathsf{rc}_S, 0)$
    $\boxed{\text{if } \mathsf{h_{CP}} \neq \mathsf{rc}_S[\mathsf{cid}].\mathsf{h_{CP}}\text{: } \mathsf{rc}_S[\mathsf{cid}] \leftarrow \perp \text{ and return } (\mathsf{rc}_S, 0)}$
    if $\pi_S^i.\mathsf{ch} \neq \mathsf{ch}$ or $\pi_S^i.\mathsf{tb} \neq \mathsf{tb}$ or $h \neq \mathsf{H}(\mathsf{id}_S)$ or $UP \neq \mathsf{true}$ or $UV \neq \pi_S^i.UV$
    or $\mathsf{rc}_S[\mathsf{cid}].\Sigma.\mathsf{Vfy}(\mathsf{rc}_S[\mathsf{cid}].pk, (ad, \mathsf{H}(m_{\mathsf{acl}})), \sigma) = 0$ or $n \leq \mathsf{rc}_S[\mathsf{cid}].n$:
    return $(\mathsf{rc}_S, 0)$
    $\mathsf{rc}_S[\mathsf{cid}].n \leftarrow n$
    $\pi_S^i.\mathsf{agCon} \leftarrow (\mathsf{id}_S, \mathsf{H}(m_{\mathsf{acl}}), n, \boxed{\mathsf{h_{CP}},}\ UV, UP)$
    $\pi_S^i.\mathsf{sid} \leftarrow (h, \mathsf{cid}, \mathsf{H}(m_{\mathsf{acl}}), n)$
    $\pi_S^i.\mathsf{st}_{\mathsf{exe}} \leftarrow \mathsf{accepted}$
    return $(\mathsf{rc}_S, 1)$
```

Fig. 11. Instantiation of ePIA = (Register, Authenticate) with WebAuthn 2 (and WebAuthn 2$^+$ that includes boxed operations) with attestation type `None`, where Register = (rChall, rCom, rRsp, rVrfy) and Authenticate = (aChall, aCom, aRsp, aVrfy).

fails. Otherwise, S updates the counter in the registration context and sets the execution state to accepted and outputs rc_S together with decision $d = 1$. The agreed context and the session identifier are the same as in aRsp.

Appendix D
Detailed Description of CTAP 2.1

A. Description of CTAP 2.1 Algorithms

authPowerUp-T: inputs a token state st_T and resets each underlying Pin/Uv Auth Protocol puvProtocol. The counter m for the consecutive tries for binding phase is set to its maximum of 3.

getInfo-T: inputs a token session π_T^i and outputs its version and the list of the supported Pin/Uv Auth Protocol. We write $\mathsf{info} \leftarrow \mathsf{getInfo}\text{-}T(\pi_T^i)$.

obtainSharedSecret-C-start: inputs a client session π_C^j and token information info = (version, puvProtocolList) and aborts if version = 2.0. Otherwise, the client session π_C^j selects a Pin/Uv Auth Protocol puvProtocol from the list puvProtocolList and initializes it locally. The execution state of π_C^j is set to waiting. Finally, this algorithm outputs the selected Pin/Uv Auth Protocol puvProtocol. We write $\mathsf{puvProtocol} \xleftarrow{\$} \mathsf{obtainSharedSecret}\text{-}C\text{-}\mathsf{start}(\pi_C^j, \mathsf{info})$.

obtainSharedSecret-T: inputs a token session π_T^i and a Pin/Uv Auth Protocol puvProtocol aborts if puvProtocol is not supported by the token T. Otherwise, this algorithm simply outputs the public key of the local instance of puvProtocol. During the execution, the status of the token session is set to waiting. We write $pk \leftarrow \mathsf{obtainSharedSecret}\text{-}T(\pi_T^i, \mathsf{puvProtocol})$.

obtainSharedSecret-C-end: inputs a client session π_C^j and a public key pk. During the execution, the client session produces a shared secret K and a ciphertext c, followed by storing the secret K locally in $\pi_C^j.\mathsf{K}$. This algorithm outputs the ciphertext c. We write $c \xleftarrow{\$} \mathsf{obtainSharedSecret}\text{-}C\text{-}\mathsf{end}(\pi_C^j, pk)$.

setPIN-C: inputs a client session π_C^j and a PIN pin and aborts if pin is not in the PIN domain $\mathcal{PIN}$. Otherwise, π_C^j encrypts this pin and authenticates the encryption using the selected Pin/Uv Auth Protocol and the locally stored shared

```
authPowerUp-$T$(st$_T$):
  foreach puvProtocol $\in$ st$_T$.puvProtocolList
    st$_T$.puvProtocol.initialize()
  st$_T.m \leftarrow 3$
obtainSharedSecret-$C$-start($\pi_C^j$, info):
  Parse (version, puvProtocolList) $\leftarrow$ info
  if version $= 2.0$: return $\perp$
  select puvProtocol $\leftarrow$ puvProtocolList
  $\pi_C^j$.selectedpuvProtocol $\leftarrow$ puvProtocol
  $\pi_C^j$.selectedpuvProtocol.initialize()
  $\pi_C^j$.st$_{\text{exe}}$ $\leftarrow$ waiting
  $\pi_C^j$.sid $\leftarrow$ $\pi_C^j$.sid $\|$ info $\|$ puvProtocol
  return puvProtocol
obtainSharedSecret-$T$($\pi_T^i$, puvProtocol) :
  if puvProtocol $\notin$ st$_T$.puvProtocolList: return $\perp$
  $pk_T$ $\leftarrow$ st$_T$.puvProtocol.getPublicKey()
  $\pi_T^i$.st$_{\text{exe}}$ $\leftarrow$ waiting
  $\pi_T^i$.sid $\leftarrow$ $\pi_T^i$.sid $\|$ puvProtocol $\|$ $pk_T$
  return $pk_T$
obtainSharedSecret-$C$-end($\pi_C^j$, $pk$) :
  $(c, K) \xleftarrow{\$}$ $\pi_C^j$.selectedpuvProtocol.encapsulate($pk$)
  $\pi_C^j$.K $\leftarrow K$
  $\pi_C^j$.sid $\leftarrow$ $\pi_C^j$.sid $\|$ $pk$ $\|$ $c$
  return $c$
setPIN-$C$($\pi_C^j$, pin) :
  if pin $\notin \mathcal{PIN}$: return $\perp$
  $c_p \xleftarrow{\$}$ $\pi_C^j$.selectedpuvProtocol.encrypt($\pi_C^j$.K, pin)
  $t_p \xleftarrow{\$}$ $\pi_C^j$.selectedpuvProtocol.authenticate($\pi_C^j$.K, $c_p$)
  return $(c_p, t_p)$
setPIN-$T$($\pi_T^i$, puvProtocol, $c, c_p, t_p$):
  if puvProtocol $\notin$ st$_T$.puvProtocolList $\vee$ st$_T$.pinHash $\neq \perp$: return $\perp$
  $K \leftarrow$ st$_T$.puvProtocol.decapsulate($c$)
  if $K = \perp \vee$ st$_T$.puvProtocol.verify($K, c_p, t_p$) = false: return $\perp$
  pin $\leftarrow$ st$_T$.puvProtocol.decrypt($K, c_p$)
  if pin $\notin \mathcal{PIN}$: return $\perp$
  st$_T$.pinHash $\leftarrow$ H(pin)
  st$_T$.pinRetries $\leftarrow$ pinRetriesMax
  return accepted
```

```
getInfo-$T$($\pi_T^i$):
  info $\leftarrow$ (st$_T$.version, st$_T$.puvProtocolList)
  $\pi_T^i$.sid $\leftarrow$ $\pi_T^i$.sid $\|$ info
  return info
obtainPinUvAuthToken-$C$-start($\pi_C^j$, pin):
  pinHash $\leftarrow$ H(pin)
  $c_{ph} \xleftarrow{\$}$ $\pi_C^j$.selectedpuvProtocol.encrypt($\pi_C^j$.K, pinHash)
  $\pi_C^j$.st$_{\text{exe}}$ $\leftarrow$ bindStart
  $\pi_C^j$.sid $\leftarrow$ $\pi_C^j$.sid $\|$ $c_{ph}$
  return $c_{ph}$
obtainPinUvAuthToken-$T$($\pi_T^i$, puvProtocol, $c, c_{ph}$):
  if puvProtocol $\notin$ st$_T$.puvProtocolList $\vee$ st$_T$.pinRetries $= 0$
    return ($\perp$, false)
  $K \leftarrow$ st$_T$.puvProtocol.decapsulate($c$)
  if $K = \perp$: return ($\perp$, false)
  st$_T$.pinRetries $\leftarrow$ st$_T$.pinRetries $- 1$
  pinHash $\leftarrow$ st$_T$.puvProtocol.decrypt($K, c_{ph}$)
  if pinHash $\neq$ st$_T$.pinHash
    st$_T$.puvProtocol.regenerate()
    if st$_T.m = 0$: authPowerUp-$T$(st$_T$): return ($\perp$, true)
  st$_T.m \leftarrow 3$, st$_T$.pinRetries $\leftarrow$ pinRetriesMax
  foreach puvProtocol$'$ $\in$ st$_T$.puvProtocolList
    st$_T$.puvProtocol$'$.resetpuvToken()
  $\pi_T^i$.bs $\leftarrow$ $\pi_T^i$.puvProtocol.$pt$
  $c_{pt} \xleftarrow{\$}$ st$_T$.puvProtocol.encrypt($K$, $\pi_T^i$.bs)
  $\pi_T^i$.st$_{\text{exe}}$ $\leftarrow$ bindDone
  $\pi_T^i$.sid $\leftarrow$ $\pi_T^i$.sid $\|$ puvProtocol $\|$ $c$ $\|$ $c_{ph}$ $\|$ $c_{pt}$ $\|$ false
  return ($c_{pt}$, false)
obtainPinUvAuthToken-$C$-end($\pi_C^j$, $c_{pt}$):
  $\pi_C^j$.bs $\leftarrow$ $\pi_C^j$.selectedpuvProtocol.decrypt($\pi_C^j$.K, $c_{pt}$)
  $\pi_C^j$.st$_{\text{exe}}$ $\leftarrow$ bindDone
  $\pi_C^j$.sid $\leftarrow$ $\pi_C^j$.sid $\|$ $c$
  return
auth-$C$($\pi_C^j$, $M$):
  $t \xleftarrow{\$}$ $\pi_C^j$.selectedpuvProtocol.authenticate($\pi_C^j$.bs, $M$)
  return $(M, t)$
validate-$T$($\pi_T^i$, $M, t, d$):
  if st$_T$.puvProtocol.verify($\pi_T^i$.bs, $M, t$) = true: return $d$
  return rejected
```

Fig. 12. CTAP 2.1 is an ePACA = (Reboot, Setup, Bind, Auth, Validate) protocol. The flow of ePACA protocol is given in Figure 4.

secret π_C^j.K. This algorithm outputs the ciphertext c and the authentication tag t. We write $(c, t) \xleftarrow{\$}$ setPIN-$C(\pi_C^j$, pin).

setPIN-T: inputs a token session π_T^i, a Pin/Uv Auth Protocol puvProtocol, two ciphertexts c and c_p, and an authentication tag t_p. It aborts if puvProtocol is not supported or the local pinHash has been set. Then, the token decapsulates c for a shared secret K and verifies the ciphertext c_p and tag t using K. If K cannot be correctly decapsulated or the verification falls, then this algorithm aborts. If a PIN pin can be correctly decrypted, then the local pinHash π_T^i.pinHash is set to hash of pin and the local counter pinRetries is set to the maximum. Otherwise, this algorithm aborts. In the end, this algorithm outputs a status status $\in$ {accepted, rejected} indicating success or failure. [9] We write status $\leftarrow$ setPIN-$T(\pi_T^i$, puvProtocol, $c, c_p, t_p)$.

obtainPinUvAuthToken-C-start: inputs a client session π_C^j and a PIN pin. The client session π_C^j computes the hash of pin and encrypts it using the selected Pin/Uv Auth Protocol and the locally stored share secret π_C^j.K. This algorithm outputs the encryption c. During the execution, the status of the client session is set to bindStart. We write $c \xleftarrow{\$}$ obtainPinUvAuthToken-C-start(π_C^j, pin).

obtainPinUvAuthToken-T: inputs a token session π_T^i, a Pin/Uv Auth Protocol puvProtocol, and two ciphertexts c and c_{ph}. It aborts if puvProtocol is not supported by T or if the local counter pinRetries is 0. Otherwise, session π_T^i decapsulates c for a key K and aborts if a failure happens during the decapsulation. Then, π_T^i decrements the counter pinRetries by 1 and decrypts c_ph using K for a hash value pinHash. If pinHash matches the locally stored st$_T$.pinHash, then the counter m and pinRetries is set to their maximum. Otherwise, the local instance puvProtocol regenerates its key pair. If the counter for the consecutive retries reaches 0, then the token is rebooted. In all cases, the token resets the pts in all Pin/Uv Auth Protocol instances. Then, the session π_T^i sets the pt underlying puvProtocol as the binding state π_T^i.bs and encrypts it using K for a ciphertext c_{pt}. This algorithm outputs c_{pt} and a boolean value calledReboot indicating whether authPowerUp-T is invoked or not. After the successful completion, the status of the token session is set to bindDone. We write (c_{pt}, calledReboot) $\xleftarrow{\$}$ obtainPinUvAuthToken-$T(\pi_T^i$, puvProtocol, $c, c_{ph})$.

[9]In practice, the user confirmation is required in this step. Here, we simply assume the user confirmation and omit it in the algorithm.

obtainPinUvAuthToken-C-end**:** inputs a client session π_C^j and a ciphertext c_{pt}. During the execution, the client decrypts the binding state π_C^j.bs from c_{pt} and the status of the client session is set to bindDone.

auth-C**:** inputs a client session π_C^j and a command M. The client session authenticates M using the selected Pin/Uv Auth Protocol and the local binding state for a tag t. This algorithm then outputs M and an authorized tag t[10]. We write $(M, t) \xleftarrow{\$} \mathsf{auth}\text{-}C(\pi_C^j, M)$.

validate-T**:** inputs a token session π_T^i, a command M, an authorized tag t, and a user decision $d \in \{\mathsf{accepted}, \mathsf{rejected}\}$, and outputs status status $=$ accepted if $d =$ accepted and M and t can be verified using the binding state π_T^i.bs and the Pin/Uv Auth Protocol, which is specified by the tag t (Cf. footnote 10); and rejected otherwise.

B. *Official Instances of Pin/Uv Auth Protocol*

CTAP 2.1 officially introduces two instantiations of Pin/Uv Auth Protocol puvProtocol, as in Fig. 13 and 14. The first, $\mathsf{puvProtocol}_1$, runs $\mathsf{initialize}_1$ by simply invoking $\mathsf{regenerate}_1$ and $\mathsf{resetpuvToken}_1$, which further samples a public-private key pair from ECDH over curve NIST P-256 and samples a random pt with length $\mu\lambda$ for $\mu \in \{1, 2\}$ and $\lambda = 128$ bits. $\mathsf{getPublicKey}_1$ outputs the internal public key pk. $\mathsf{encapsulate}_1$ computes the key exchange using as input ECDH public key and its internal private key and applies $\mathsf{H}_1 = \text{SHA-256}$ to the x-coordinate of the key exchange result for a shared K, followed by outputting its internal public key and K. $\mathsf{decapsulate}_1$ recovers the shared secret K from ciphertext c using its internal private key sk. $\mathsf{encrypt}_1$ encrypts a message m using SKE_1 and a symmetric key K, where SKE_1 denotes AES-256-CBC encryption using an all-zero initial vector IV. $\mathsf{decrypt}_1$ recovers the message from ciphertext c by using SKE_1 and key K. $\mathsf{authenticate}_1$ authenticates a message m using K' by applying H_2 to both, where H_2 runs HMAC-SHA-256 and truncates the result to the first 128 bits. verify_1 outputs true if $t = \mathsf{H}_2(K', m)$, and false otherwise[11].

The second instantiation $\mathsf{puvProtocol}_2$ runs $\mathsf{initialize}_2$, $\mathsf{regenerate}_2$, and $\mathsf{getPublicKey}_2$ identical to the ones in $\mathsf{puvProtocol}_1$. The $\mathsf{resetpuvToken}_2$ algorithm outputs a pt with fixed 256 bits length. The algorithm $\mathsf{encapsulate}_2$ first computes the x coordinate of the ECDH exchange of input public key and internal private key, denoted by Z, followed by applying H_3 to Z and "CTAP2 HMAC key" for a HMAC key K_1 and to Z and "CTAP2 AES key" for a AES key K_2. Finally, $\mathsf{encapsulate}_2$ outputs its internal public key as ciphertext as well as K_1 and K_2. $\mathsf{decapsulate}_2$ recovers HMAC key K_1 and AES key K_2 from the input ciphertext c using its internal private key. $\mathsf{encrypt}_2$ splits the input K into two sub-keys K_1 and K_2 where K_1 has length of 256 bits. Then, it encrypts a message m using SKE_2 on key K_2, where SKE_2 denotes AES-256-CBC encryption using a randomized initial vector IV. $\mathsf{decrypt}_2$ recovers the message m from ciphertext c using the key K_2, where K_2 discards the first 256 bits of K. $\mathsf{authenticate}_2$ applies H_4 to key K_1' and a message m to produce a tag t, where H_4 denotes HMAC-SHA-256 and K_1' is the first 256 bits of the input K'. verify_2 on a key K', a message m, and a tag t, verifies whether the tag t matches $\mathsf{H}_4(K_1', m)$, where K_1' is the first 256 bits of K'.

```
$\underline{\mathsf{initialize}_1():}$
    $\mathsf{regenerate}_1()$
    $\mathsf{resetpuvToken}_1()$
$\underline{\mathsf{regenerate}_1():}$
    $(pk, sk) \xleftarrow{\$} \mathsf{ECDH.KG}()$
$\underline{\mathsf{resetpuvToken}_1():}$
    $pt \xleftarrow{\$} \{0,1\}^{\mu\lambda}$
$\underline{\mathsf{encapsulate}_1(pk'):}$
    $Z \leftarrow \mathsf{XCoordinateOf}(sk \cdot pk')$
    $K \leftarrow \mathsf{H}_1(Z)$ , $c \leftarrow pk$
    $\textbf{return } (c, K)$
$\underline{\mathsf{decapsulate}_1(c):}$
    $Z \leftarrow \mathsf{XCoordinateOf}(sk \cdot c)$
    $K \leftarrow \mathsf{H}_1(Z)$
    $\textbf{return } K$
$\underline{\mathsf{getPublicKey}_1():}$
    $\textbf{return } pk$
$\underline{\mathsf{encrypt}_1(K, m):}$
    $c \leftarrow \mathsf{SKE}_1.\mathsf{Enc}(K, m)$
    $\textbf{return } c$
$\underline{\mathsf{decrypt}_1(K, c):}$
    $m \leftarrow \mathsf{SKE}_1.\mathsf{Dec}(K, c)$
    $\textbf{return } m$
$\underline{\mathsf{authenticate}_1(K', m):}$
    $t \leftarrow \mathsf{H}_2(K', m)$
    $\textbf{return } t$
$\underline{\mathsf{verify}_1(K', m, t):}$
    $t' \leftarrow \mathsf{H}_2(K', m)$
    $\textbf{return } [\![t = t']\!]$
```

Fig. 13. The first instantiation of PIN/UV Auth Protocol $\mathsf{puvProtocol}_1$. The operation $\cdot$ denotes scalar multiplication.

```
$\underline{\mathsf{initialize}_2():}$
    $\mathsf{regenerate}_2()$
    $\mathsf{resetpuvToken}_2()$
$\underline{\mathsf{regenerate}_2():}$
    $(pk, sk) \xleftarrow{\$} \mathsf{ECDH.KG}()$
$\underline{\mathsf{resetpuvToken}_2():}$
    $pt \xleftarrow{\$} \{0,1\}^{2\lambda}$
$\underline{\mathsf{encapsulate}_2(pk'):}$
    $Z \leftarrow \mathsf{XCoordinateOf}(sk \cdot pk')$
    $K_1 \leftarrow \mathsf{H}_3(Z, \text{"CTAP2 HMAC key"})$
    $K_2 \leftarrow \mathsf{H}_3(Z, \text{"CTAP2 AES key"})$
    $K \leftarrow (K_1, K_2)$
    $c \leftarrow pk$
    $\textbf{return } (c, K)$
$\underline{\mathsf{decapsulate}_2(c):}$
    $Z \leftarrow \mathsf{XCoordinateOf}(sk \cdot c)$
    $K_1 \leftarrow \mathsf{H}_3(Z, \text{"CTAP2 HMAC key"})$
    $K_2 \leftarrow \mathsf{H}_3(Z, \text{"CTAP2 AES key"})$
    $K \leftarrow (K_1, K_2)$
    $\textbf{return } K$
$\underline{\mathsf{getPublicKey}_2():}$
    $\textbf{return } pk$
$\underline{\mathsf{encrypt}_2(K, m):}$
    Parse $(K_1, K_2) \leftarrow K$
        s.t. $|K_1| = 2\lambda$
    $c \leftarrow \mathsf{SKE}_2.\mathsf{Enc}(K_2, m)$
    $\textbf{return } c$
$\underline{\mathsf{decrypt}_2(K, c):}$
    Parse $(K_1, K_2) \leftarrow K$
        s.t. $|K_1| = 2\lambda$
    $m \leftarrow \mathsf{SKE}_2.\mathsf{Dec}(K_2, c)$
    $\textbf{return } m$
$\underline{\mathsf{authenticate}_2(K', m):}$
    Parse $(K_1', K_2') \leftarrow K'$
        s.t. $|K_1'| = 2\lambda$
    $t \leftarrow \mathsf{H}_4(K_1', m)$
    $\textbf{return } t$
$\underline{\mathsf{verify}_2(K', m, t):}$
    Parse $(K_1', K_2') \leftarrow K'$
        s.t. $|K_1'| = 2\lambda$
    $t' \leftarrow \mathsf{H}_4(K_1', m)$
    $\textbf{return } [\![t = t']\!]$
```

Fig. 14. The 2nd instantiation of PIN/UV Auth Protocol $\mathsf{puvProtocol}_2$.

[10]In practice, this authorized tag t also includes information that specifies the index of Pin/Uv Auth Protocol. Here, we omit this.

[11]In practice, if $K' = pt$, then verify_1 also outputs fails if pt is not in-use. Note that the usage time of the pt is out of the scope of this paper. We omit this here and in the following verify_2 in $\mathsf{puvProtocol}_2$.