Provable Security Analysis of FIDO2

Manuel Barbosa[1], Alexandra Boldyreva[2], Shan Chen[3], and Bogdan Warinschi[4]

[1] University of Porto (FCUP) and INESC TEC, Portugal
mbb@fc.up.pt
[2] Georgia Institute of Technology, USA
sasha@gatech.edu
[3] Technische Universität Darmstadt, Germany*
dragoncs16@gmail.com
[4] University of Bristol, UK and Dfinity, Switzerland
csxbw@bristol.ac.uk

Abstract. We carry out the first provable security analysis of the new FIDO2 protocols, the promising FIDO Alliance's proposal for a standard for *passwordless* user authentication. Our analysis covers the core components of FIDO2: the W3C's Web Authentication (WebAuthn) specification and the new Client-to-Authenticator Protocol (CTAP2).
Our analysis is *modular*. For WebAuthn and CTAP2, in turn, we propose appropriate security models that aim to capture their intended security goals and use the models to analyze their security. First, our proof confirms the authentication security of WebAuthn. Then, we show CTAP2 can only be proved secure in a weak sense; meanwhile, we identify a series of its design flaws and provide suggestions for improvement. To withstand stronger yet realistic adversaries, we propose a generic protocol called sPACA and prove its strong security; with proper instantiations, sPACA is also more efficient than CTAP2. Finally, we analyze the overall security guarantees provided by FIDO2 and WebAuthn+sPACA based on the security of their components.
We expect that our models and provable security results will help clarify the security guarantees of the FIDO2 protocols. In addition, we advocate the adoption of our sPACA protocol as a substitute for CTAP2 for both stronger security and better performance.

1 Introduction

Motivation. Passwords are pervasive yet insecure. According to some studies, the average consumer of McAfee has 23 online accounts that require a password [17], and the average employee using LastPass has to manage 191 passwords [22]. Not only are the passwords difficult to keep track of, but it is well-known that achieving strong security while relying on passwords is quite difficult (if not impossible). According to the Verizon Data Breach Investigations Report [34], 81% of hacking-related breaches relied on either stolen and/or

* Shan Chen did most of his work while at Georgia Institute of Technology.

weak passwords. What some users may consider an acceptable password, may not withstand sophisticated and powerful modern password cracking tools. Moreover, even strong passwords may fall prey to phishing attacks and identity fraud. According to Symantec, in 2017, phishing emails were the most widely used means of infection, employed by 71% of the groups that staged cyber attacks [31].

An ambitious project which tackles the above problem is spearheaded by the Fast Identity Online (FIDO) Alliance. A truly international effort, the alliance has working groups in the US, China, Europe, Japan, Korea and India and has brought together many companies and types of vendors, including Amazon, Google, Microsoft, Apple, RSA, Intel, Yubico, Visa, Samsung, major banks, etc.

The goal is to enable user-friendly passwordless authentication secure against phishing and identity fraud. The core idea is to rely on security devices (controlled via biometrics and/or PINs) which can then be used to register and later seamlessly authenticate to online services. The various standards defined by FIDO formalize several protocols, most notably Universal Authentication Framework (UAF), the Universal Second Factor (U2F) protocols and the new FIDO2 protocols: W3C's Web Authentication (WebAuthn) and FIDO Alliance's Client-to-Authenticator Protocol v2.0 (CTAP2[5]).

FIDO2 is moving towards wide deployment and standardization with great success. Major web browsers including Google Chrome and Mozilla Firefox have implemented WebAuthn. In 2018, Client-to-Authenticator Protocol (CTAP)[6] was recognized as international standards by the International Telecommunication Union's Telecommunication Standardization Sector (ITU-T). In 2019, WebAuthn became an official web standard. Also, Android and Windows Hello earned FIDO2 Certification. Although the above deployment is backed-up by highly detailed description of the security goals and a variety of possible attacks and countermeasures, these are informal [21].

Our Focus. We provide the first provable security analysis of the FIDO2 protocols. Our focus is to clarify the formal trust model assumed by the protocols, to define and prove their exact security guarantees, and to identify and fix potential design flaws and security vulnerabilities that hinder their widespread use. Our analysis covers the actions of human users authorizing the use of credentials via *gestures* and shows that, depending on the capabilities of security devices, such gestures enhance the security of FIDO2 protocols in different ways. We concentrate on the FIDO2 authentication properties and leave the study of its arguably less central anonymity goals for future work.

Related Work. Some initial work in this direction already exists. Hu and Zhang [25] analyzed the security of FIDO UAF 1.0 and identified several vulnerabilities in different attack scenarios. Later, Panos *et al.* [32] analyzed FIDO UAF 1.1 and explored some potential attack vectors and vulnerabilities. However, both works were informal. FIDO U2F and WebAuthn were analyzed using the applied pi-calculus and ProVerif tool [23, 27, 33]. Regarding an older version

[5] The older version is called CTAP1/U2F.
[6] CTAP refers to both versions: CTAP1/U2F and CTAP2.

of FIDO U2F, Pereira *et al.* [33] presented a server-in-the-middle attack and Jacomme and Kremer [27] further analyzed it with a structured and fine-grained threat model for malware. Guirat and Halpin [23] confirmed the authentication security provided by WebAuthn while pointed out that the claimed privacy properties (i.e., account unlinkability) failed to hold due to the same attestation key pair used for different servers.

However, *none* of the existing work employs the cryptographic provable security approach to the FIDO2 protocols in the course of deployment. In particular, there is no analysis of CTAP2, and the results for WebAuthn [27] are limited in scope: as noted by the authors themselves, their model "makes a number of simplifications and so much work is needed to formally model the complete protocol as given in the W3C specification". The analysis in [27] further uses the *symbolic* model (often called the Dolev-Yao model [18]), which captures weaker adversarial capabilities than those in computational models (e.g., the Bellare-Rogaway model [10]) employed by the provable security approach we adopt here.

The works on two-factor authentication (e.g., [16,29]) are related to our work, but the user in such protocols has to use the password *and* the two-factor device during each authentication/login. With FIDO2, *there is no password* during user registration or authentication. The PIN used in FIDO2 is meant to authorize a client (e.g., a browser) access to an authenticator device (e.g., an authentication token); the server does not use passwords at all.[7] Some two-factor protocols can also generate a binding cookie after the first login to avoid using the two-factor device or even the password for future logins. However, this requires trusting the client, e.g., a malicious browser can log in as the user without having the two-factor device (or the password). FIDO2 uses the PIN to prevent an attacker with a stolen device from authenticating to a server from a new client.

Our work is not directly applicable to federated authentication protocols such as Kerberos, OAuth, or OpenID. FIDO2 allows the user to keep a single hardware token that it can use to authenticate to multiple servers without having to use a federated identity. The only trust anchor is an attestation key pair for the token. To the best of our knowledge, there are no complete and formal security models for federated authentication in the literature, but such models would differ significantly from the ones we consider here. It is interesting to see how FIDO2 and federated authentication can be used securely together; we leave this as an interesting direction for future work. Our work could, however, be adapted to analyze some second-factor authentication protocols like Google 2-step [2].

FIDO2 Overview. FIDO2 consists of two core components (see Fig. 1 for the communication channels and Fig. 2 for the simplified FIDO2 flow).

WebAuthn is a web API that can be built into browsers to enable web applications to integrate user authentication. At its heart, WebAuthn is a *passwordless* "challenge-response" scheme between a server and a user. The user relies on a trusted authenticator device (e.g., a security token or a smartphone) and a possibly untrusted client (e.g., a browser or an operating system installed on

[7] Some form of prior user authentication method is required for registration of a new credential, but this is a set-up assumption for the protocol.

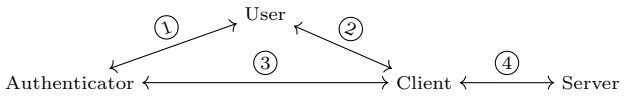

Fig. 1. Communication channels

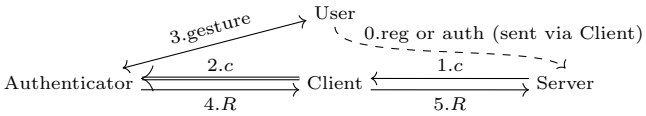

Fig. 2. FIDO2 flow (simplified): double arrow = CTAP2 authorized message.

the user's laptop). Such a device-assisted "challenge-response" scheme works as follows (details in Section 5). First, in the registration phase, the server sends a random challenge to the security device through the client. In this phase, the device signs the challenge using its long-term embedded attestation secret key, along with a new public key credential to use in future interactions; the credential is included in the response to the server. In the subsequent interactions, which correspond to user authentication, the challenge sent by the server is signed by the device using the secret key corresponding to the credential. In both cases, the signature is verified by the server.

The other FIDO2 component, CTAP2, specifies the communication between an authenticator device and the client (usually a browser). Its goal is to guarantee that the client can only use the authenticator with the user's permission, which the user gives by 1) entering a PIN when the authenticator powers up and 2) directly using the authenticator interface (e.g., a simple push-button) to authorize registration and authentication operations. CTAP2 specifies how to configure an authenticator with a user's PIN. Roughly speaking, its security goal is to "bind" a trusted client to the set-up authenticator by requiring the user to provide the correct PIN, such that the authenticator accepts only messages sent from a "bound" client. We remark that, surprisingly, CTAP2 relies on the (unauthenticated) Diffie-Hellman key exchange. The details are in Section 7.

Our Contributions. We perform the first thorough cryptographic analysis of the authentication properties guaranteed by FIDO2 using the provable security approach. Our analysis is conducted in a *modular* way. That is, we first analyze WebAuthn and CTAP2 components separately and then derive the overall security of a typical use of FIDO2. We note that our models, although quite different, follow the Bellare-Rogaway model [10] that was proposed to analyze key exchange protocols, which defines oracle queries to closely simulate the real-world adversarial abilities. Its extensions (like ours) have been widely used to analyze real-world protocols such as TLS 1.3 [13, 19], Signal [14], etc.

Provable security of WebAuthn. We start our analysis with the simpler base protocol, WebAuthn. We define the class of *passwordless authentication (PlA)* protocols that capture the syntax of WebAuthn. Our PlA model considers an authenticator and a server (often referred to as a relying party) communicating through a client, which consists of two phases. The server is assumed to know

the attestation public key that uniquely identifies the authenticator. In the *registration* phase the authenticator and the server communicate with the intention to establish some joint state corresponding to this registration session: this joint state fixes a credential, which is bound to the authenticator's attestation public key vk and a server identity id_S (e.g., a server domain name). The server gets the guarantee that the joint state is stored in a specific authenticator, which is assumed to be tamper-proof. The joint state can then be used in the *authentication* phase. Here, the authenticator and the server engage in a message exchange where the goal of the server is to verify that it is interacting with the same authenticator that registered the credential bound to (vk, id_S).

Roughly speaking, a PlA protocol is secure if, whenever an authentication/registration session completes on the server side, there is a unique partnered registration/authentication session which completed successfully on the authenticator side. For authentication sessions, we further impose that there is a unique associated registration session on both sides, and that these registration sessions are also uniquely partnered. This guarantees that registration contexts (i.e., the credentials) are *isolated* from one another; moreover, if a server session completes an authentication session with an authenticator, then the authenticator must have completed a registration session with the server earlier. We use the model thus developed to prove the security of WebAuthn under the assumption that the underlying hash function is collision-resistant and the signature scheme is unforgeable. Full details can be found in Section 5.

Provable security of CTAP2. Next we study the more complex CTAP2 protocol. We define the class of *PIN-based access control for authenticators (PACA)* protocols to formalize the general syntax of CTAP2. Although CTAP2 by its name may suggest a two-party protocol, our PACA model involves the user as an additional participant and therefore captures human interactions with the client and the authenticator (e.g., the user typing its PIN into the browser window or rebooting the authenticator). A PACA protocol runs in three phases as follows. First, in the authenticator setup phase, the user "embeds" its PIN into the authenticator via a client and, as a result, the authenticator stores a PIN-related long-term state. Then, in the binding phase, the user authorizes the client to "bind" itself to the authenticator (using the same PIN). At the end of this phase, the client and the authenticator end up with a (perhaps different) binding state. Finally, in the access channel phase, the client is able to send any authorized message (computed using its binding state) to the authenticator, which verifies it using its own binding state. Note that the final established access channel is *unidirectional*, i.e., it only guarantees authorized access from the client to the authenticator but not the other way.

Our model captures the security of the access channels between clients and authenticators. The particular implementation of CTAP2 operates as follows. In the binding phase, the authenticator privately sends its associated secret called pinToken (generated upon power-up) to the trusted client and the pinToken is then stored on the client as the binding state. Later, in the access channel phase, that binding state is used by the bound client to authenticate messages sent to

the authenticator. We note that, by the CTAP2 design, each authenticator is associated with a *single* pinToken per power-up, so multiple clients establish multiple access channels with the same authenticator using the *same* pinToken. This limits the security of CTAP2 access channels: for a particular channel from a client to an authenticator to be secure (i.e., no attacker can forge messages sent over that channel), *none* of the clients bound to the same authenticator during the same power-up can be compromised.

Motivated by the above discussion, we distinguish between unforgeability (UF) and strong unforgeability (SUF) for PACA protocols. The former corresponds to the weak level of security discussed above. The latter, captures *strong* fine-grained security where the attacker can compromise any clients except those involved in the access channels for which we claim security. As we explain later (Section 6), SUF also covers certain *forward secrecy* guarantees for authentication. For both notions, we consider a powerful attacker that can manipulate the communication between parties, compromise clients (that are not bound to the target authenticator) to reveal the binding states, and corrupt users (that did not set up the target authenticator) to learn their secret PINs.

Even with the stronger trust assumption (made in UF) on the bound clients, we are unable to prove that CTAP2 realizes the expected security model: we describe an attack that exploits the fact that CTAP2 uses unauthenticated Diffie-Hellman. Since it is important to understand the limits of the protocol, we consider a further refinement of the security models which makes stronger trust assumptions on the binding phase of the protocol. Specifically, in the *trusted binding* setting the attacker cannot launch active attacks against the client during the binding phase, but it may try to do so against the authenticator, i.e., it cannot launch man-in-the-middle (MITM) attacks but it may try to impersonate the client to the authenticator. We write UF-t and SUF-t for the security levels which consider trusted binding and the distinct security goals outlined above. In summary we propose four notions: by definition SUF is the strongest security notion and UF-t is the weakest one. Interestingly, UF and SUF-t are *incomparable* as established by our separation result discussed in Section 7 and Section 8. Based on our security model, we prove that CTAP2 achieves the weakest UF-t security and show that it is not secure regarding the three stronger notions. Finally, we identify a series of design flaws of CTAP2 and provide suggestions for improvement.

Improving CTAP2 security. CTAP2 cannot achieve UF security because in the binding phase it uses unauthenticated Diffie-Hellman key exchange which is vulnerable to MITM attacks. This observation suggests a change to the protocol which leads to stronger security. Specifically, we propose a generic sPACA protocol (for strong PACA), which replaces the use of unauthenticated Diffie-Hellman in the binding phase with a *password-authenticated key exchange (PAKE)* protocol. Recall that PAKE takes as input a common password and outputs the same random session key for both parties. The key observation is that the client and the authenticator share a value (derived from the user PIN) which can be viewed as a password. By running PAKE with this password as input, the client

and the authenticator obtain a strong key which can be used as the binding state to build the access channel. Since each execution of the PAKE (with different clients) results in a fresh independent key, we can prove that sPACA is a SUF-secure PACA protocol. Furthermore, we compare the performance of CTAP2 and sPACA (with proper PAKE instantiations). The results show that our sPACA protocol is also more efficient, so it should be considered for adoption.

Composed security of CTAP2 and WebAuthn. Finally, towards our main goal of the analysis of full FIDO2 (by full FIDO2 we mean the envisioned usage of the two protocols), we study the composition of PlA and PACA protocols (cf. Section 9). The composed protocol, which we simply call PlA+PACA, is defined naturally for an authenticator, user, client, and server. The composition, and the intuition that underlies its security, is as follows. Using PACA, the user (via a client) sets a PIN for the authenticator. This means that only clients that obtain the PIN from the user can "bind" to the authenticator and issue commands that it will accept. In other words, PACA establishes the access channel from the bound client to the authenticator. Then, the challenge-response protocols of PlA run between the server and the authenticator, via a PACA-bound client. The server-side guarantees of PlA are preserved, but now the authenticator can control client access to its credentials using PACA; this composition result is intuitive and easy to prove given our modular formalization.

Interestingly, we formalize an even stronger property that shows that FIDO2 gives end-to-end mutual authentication guarantees between the server and the authenticator when clients and servers are connected by an authenticated server-to-client channel (e.g., a TLS connection). The mutual authentication guarantees extend the PlA guarantees: authenticator, client, and server must all be using the same registration context for authentication to succeed. We note that Transport Layer Security (TLS) provides a server-to-client authenticated channel, and hence this guarantee applies to the typical usage of FIDO2 over TLS. Our results apply to WebAuthn+CTAP2 (under a UF-t adversarial model) and WebAuthn+sPACA (under a SUF adversarial model).

We conclude with an analysis of the role of user gestures in FIDO2. We first show that SUF security offered by sPACA allows the user, equipped with an authenticator that can display a simple session identifier, to detect and prevent attacks from malware that may compromise the states of PACA clients previously bound to the authenticator. (This is not possible for the current version of CTAP2.) We also show how simple gestures can allow a human user to keep track of which server identity is being used in PlA sessions.

Summary. Our analyses clarify the security guarantees FIDO2 should provide for the various parties involved in the most common usage scenario where: 1) the user owns a simple hardware token that is capable of accepting push-button gestures and, optionally, to display a session identifier code (akin to bluetooth pairing codes); 2) the user configures the token with a PIN using a trusted machine; 3) the user connects/disconnects the token on multiple machines, some trusted, some untrusted, and uses it to authenticated to multiple servers.

In all these interactions, the server is assured that during authentication it can recognize if the same token was used to register a key, and that this token was bound to the client it is talking to since the last power-up (this implies entering the correct PIN *recently*). This guarantee assumes that the client is not corrupted (i.e., the browser window where the user entered the PIN is isolated from malicious code and can run the CTAP2 protocol correctly) and that an active attack against the client via the CTAP2 API to guess the user entered PIN is detected (we know this is the case on the token side, as CTAP2 defines a blocking countermeasure).

Assuming a server-to-client authenticated channel, the user is assured that while it is in possession of the PIN, no one can authenticate on her behalf, except if she provides the PIN to a corrupted browser window. Moreover, the scope of this possible attack is limited to the current power-up period. If we assume that registration was conducted via an honest client, then we know that all authentication sessions with honest clients are placed to the correct server. Finally, if the token is stolen, the attacker still needs to guess the PIN (without locking the token) in order to impersonate the user.

With our proposed modifications, FIDO2 will meet this level of security. Without them, these guarantees will only hold assuming weaker client corruption capabilities and more importantly, the attacker cannot perform active man-in-the-middle attacks during all binding sessions, which may be unrealistic.

2 Preliminaries

In the full version of this paper [6], we recall the definitions of pseudorandom functions (PRFs), collision-resistant hash function families, message authentication codes (MACs), signature schemes, the computational Diffie-Hellman (CDH) problem and strong CDH (sCDH) problem, as well as the corresponding advantage measures $\mathbf{Adv}^{\mathsf{prf}}$, $\mathbf{Adv}^{\mathsf{coll}}$, $\mathbf{Adv}^{\mathsf{euf\text{-}cma}}$, $\mathbf{Adv}^{\mathsf{euf\text{-}cma}}$, $\mathbf{Adv}^{\mathsf{cdh}}$, $\mathbf{Adv}^{\mathsf{scdh}}$. There we also recall the syntax for PAKE and its security of perfect forward secrecy and explicit authentication.

3 Execution Model

The protocols we consider involve four disjoint sets of parties. Formally, the set of parties $\mathcal{P}$ is partitioned into four disjoint sets of users $\mathcal{U}$, authenticators (or tokens for short) $\mathcal{T}$, clients $\mathcal{C}$, and servers $\mathcal{S}$. Each party has a well-defined and non-ambiguous identifier, which one can think of as being represented as an integer; we typically use P, U, T, C, S for identifiers bound to a party in a security experiment and id for the case where an identifier is provided as an input in the protocol syntax.

For simplicity, we do not consider certificates or certificate checks but assume the public key associated with a party is supported by a *public key infrastructure (PKI)* and hence certified and bound to the party's identity. This issue arises explicitly only for attestation public keys bound to authenticators in Section 4.

The possible communication channels are represented as double-headed arrows in Fig. 1. In FIDO2, the client is a browser and the user-client channel is the browser window, which keeps no long-term state. The authenticator is a hardware token or mobile phone that is connected to the browser via an untrusted link that includes the operating system, some authenticator-specific middleware, and a physical communication channel that connects the authenticator to the machine hosting the browser. The authenticator exposes a simple interface to the user that allows it to perform a "gesture", confirming some action; ideally the authenticator should also be able to display information to the user (this is natural when using a mobile phone as an authenticator but not so common in USB tokens or smartcards). Following the intuitive definitions of *human-compatible communications* by Boldyreva *et al.* [12], we require that messages sent to the user be *human-readable* and those sent by the user be *human-writable*.[8] The user PIN needs to be *human-memorizable*.

We assume authenticators have a good source of random bits and keep volatile and static (or long-term) storage. Volatile storage is erased every time the device goes through a power-down/power-up cycle, which we call a *reboot*. Static storage is assumed to be initialized using a procedure carried out under special setup trust assumptions; in the case of this paper we will consider the setup procedures to generate an attestation key pair for the authenticator and to configure a user PIN, i.e., to "embed" the PIN in the authenticator.

Trust model. For each of the protocols we analyze in the paper we specify a trust model, which justifies our proposed security models. Here we state the trust assumptions that are always made throughout the paper. First, human communications (①②) are authenticated and private. This in practice captures the direct human-machine interaction between the human user and the authenticator device or the client terminal, which involves physical senses and contact that we assume cannot be eavesdropped or interrupted by an attacker. Second, client-authenticator communications (③) are not protected, i.e., neither authenticated nor private. Finally, authenticators are assumed to be tamper-proof, so our models will not consider corruption of their internal state.

Modeling users and their gestures. We do not include in our protocol syntaxes and security models explicit state keeping and message passing for human users, i.e., there are no *session oracles* for users in the security experiments. We shortly explain why this is the case. The role of the user in these protocols is to a) first check that the client is operating on correct inputs, e.g., by looking at the browser window to see if the correct server identity is being used; b) possibly (if the token has the capability to display information) check that the token and client are operating on consistent inputs; and c) finally confirm to the token that this is the case. Therefore, the user itself plays the role of an out-of-band secure channel via which the consistency of information exchanged between the client and the token can be validated.

[8] We regard understandable information displayed on a machine as human-readable and typing in a PIN or rebooting an authenticator as human-writable.

We model this with a public gesture predicate G that captures the semantics of the user's decision. Intuitively, the user decision $d \in \{0,1\}$ is given by $d = \mathsf{G}(x,y)$, where x and y respectively represent the information conveyed to the user by the client and the token in step b) above. Note that x, y may not be input by the user. Tokens with different user interface capabilities give rise to different classes of gesture predicates. For example, if a user can observe a server domain name id on the token display before pressing a button, then we can define the gesture of checking that the token displayed an identifier id that matches the one displayed by the client id^* as $\mathsf{G}(\mathsf{id}^*, \mathsf{id}) = (\mathsf{id}^* \stackrel{?}{=} \mathsf{id})$.

User actions are hardwired into the security experiments as direct inputs to either a client or a token, which is justified by our assumption that users interact with these entities via fully secure channels. We stress that here G is a modeling tool, which captures the sequence of interactions a), b), c) above. Providing a gesture means physical possession of the token, so an attacker controlling only some part of the client machine (e.g., malware) is *not* able to provide a gesture. Moreover, requiring a gesture from the user implies that the user can detect when some action is requested from the token.

4 Passwordless Authentication

We start our analysis with the simpler FIDO2 component protocol, WebAuthn. In order to analyze the authentication security of WebAuthn we first define the syntax and security model for *passwordless authentication (PlA)* protocols.

4.1 Protocol Syntax

A PlA protocol is an interactive protocol among three parties: a token (representing a user), a client, and a server. The token is associated with an attestation public key that is pre-registered to the server. The protocol defines two types of interactions: registration and authentication. In registration the server requests the token to register some initial authentication parameters. If this succeeds, the server can later recognize the same token using a challenge-response protocol.

The possible communication channels are as shown in Fig. 1, but we do not include the user. Servers are accessible to clients via a communication channel that models Internet communications.

The state of token T, denoted by st_T, is partitioned into the following (static) components: i) an attestation key pair (vk_T, ak_T) and ii) a set of registration contexts $\mathsf{st}_T.\mathsf{rct}$. A server S also keeps its registration contexts $\mathsf{st}_S.\mathsf{rcs}$. Clients do not keep long-term state.[9] All states are initialized to the empty string ε.

A PlA protocol consists of the following algorithms and subprotocols:

Key Generation: This algorithm, denoted by Kg, is executed *at most once* for each authenticator; it generates an attestation key pair (vk, ak).

[9] Some two-factor protocols may have a "trust this computer" feature that requires the client to store some long-term states. This is not included in our model as to the best of our knowledge FIDO2 does not have that feature.

Register: This subprotocol is executed among a token, a client, and a server. The token inputs its attestation secret key ak_T; the client inputs an intended server identity $\hat{\mathsf{id}}_S$; and the server inputs its identity id_S (e.g., a server domain name) and the token's attestation public key vk_T. At the end of the subprotocol, each party that successfully terminates obtains a new registration context, and sets its *session identifier* that can be used to uniquely name a (registration or authentication) session. Note that the token may successfully complete the subprotocol while the server may fail to, in the same run.

Authenticate: This subprotocol is executed between a token, a client, and a server. The token inputs its registration contexts; the client inputs an intended server identity $\bar{\mathsf{id}}_S$; and the server inputs its identity id_S and registration contexts. At the end of the subprotocol, the server *accepts* or *rejects*. Each party on success sets its session identifier and updates the registration contexts.

Restricted class of protocols. For both Register and Authenticate, we focus on 2-pass challenge-response protocols with the following structure:

- Server-side computation is split into four procedures: rchallenge and rcheck for registration, achallenge and acheck for authentication. The challenge algorithms are probabilistic, which take the server's input to the Register or Authenticate subprotocol and return a challenge. The check algorithms get the same input, the challenge, and a response. rcheck outputs the updated registration contexts rcs that are later input by acheck; acheck outputs a bit b (1 for accept and 0 for reject) and updates rcs.
- Client-side computation is modeled as two deterministic functions rcommand and acommand that capture possible checks and translations performed by the client before sending the challenges to the token. These algorithms output commands denoted by M_r, M_a respectively, which they generate from the input intended server identity and the challenge. The client may append some information about the challenge to the token's response before sending it to the server, which is an easy step that we do not model explicitly.
- Token-side computation is modeled as two probabilistic algorithms rresponse and aresponse that, on input a command and the token's input to the Register or Authenticate subprotocol, generate a response and update the registration contexts rct. In particular, rresponse outputs the updated registration contexts rct that are later input by aresponse; aresponse may also update rct.

Correctness. Correctness imposes that for any server identities $\mathsf{id}_S, \hat{\mathsf{id}}_S, \bar{\mathsf{id}}_S$ the following probability is 1:

$$\Pr\left[\, b = ((\mathsf{id}_S \stackrel{?}{=} \hat{\mathsf{id}}_S) \wedge (\mathsf{id}_S \stackrel{?}{=} \bar{\mathsf{id}}_S)) \;\middle|\; \begin{array}{l} (ak, vk) \stackrel{\$}{\leftarrow} \mathsf{Kg}(\,) \\ c_r \stackrel{\$}{\leftarrow} \mathsf{rchallenge}(\mathsf{id}_S, vk) \\ M_r \leftarrow \mathsf{rcommand}(\hat{\mathsf{id}}_S, c_r) \\ (R_r, \mathsf{rct}) \stackrel{\$}{\leftarrow} \mathsf{rresponse}(ak, M_r) \\ \mathsf{rcs} \leftarrow \mathsf{rcheck}(\mathsf{id}_S, vk, c_r, R_r) \\ c_a \stackrel{\$}{\leftarrow} \mathsf{achallenge}(\mathsf{id}_S, \mathsf{rcs}) \\ M_a \leftarrow \mathsf{acommand}(\bar{\mathsf{id}}_S, c_a) \\ (R_a, \mathsf{rct}) \stackrel{\$}{\leftarrow} \mathsf{aresponse}(\mathsf{rct}, M_a) \\ (b, \mathsf{rcs}) \leftarrow \mathsf{acheck}(\mathsf{id}_S, \mathsf{rcs}, c_a, R_a) \end{array} \right]$$

Intuitively, correctness requires that the server always accepts an authentication that is consistent with a prior registration, if and only if the client's input intended server identities match the server identity received from the server. Note that the latter check is performed by the client rather than the human user. It helps to prevent a so-called server-in-the-middle attack identified in [33].

4.2 Security Model

Trust model. Before defining security we clarify that there are no security assumptions on the communication channels shown in Fig. 1. Again, authenticators are assumed to be tamper-proof, so the model will not consider corruption of their internal state. (Note that clients and servers keep *no* secret state.) We assume the key generation stage, where the attestation key pair is created and installed in the token, is either carried out within the token itself, or performed in a trusted context that leaks nothing about the attestation secret key.

Session oracles. As with the Bellare-Rogaway model [10], to capture multiple sequential and parallel PlA executions (or instances), we associate each party $P \in \mathcal{T} \cup \mathcal{S}$ with a set of session oracles $\{\pi_P^{i,j}\}_{i,j}$, which models two types of PlA instances corresponding to registration and authentication. We omit session oracles for clients, since all they do can be performed by the adversary. For servers and tokens, session oracles are structured as follows: $\pi_P^{i,0}$ refers to the i-th registration instance of P, whereas $\pi_P^{i,j}$ for $j \geq 1$ refers to the j-th authentication instance of P associated with $\pi_P^{i,0}$ after this registration completed. A party's static storage is maintained by the security experiment and shared among all of its session oracles.

Security experiment. The security experiment is run between a challenger and an adversary $\mathcal{A}$. At the beginning of the experiment, the challenger runs $(ak_T, vk_T) \xleftarrow{\$} \mathsf{Kg}(\,)$ for all $T \in \mathcal{T}$ to generate their attestation key pairs and assign unique identities $\{\mathsf{id}_S\}_{S \in \mathcal{S}}$ to all servers. The challenger also manages the attestation public keys $\{vk_T\}_{T \in \mathcal{T}}$ and provides them to the server oracles as needed. The adversary $\mathcal{A}$ is given all attestation public keys and server identities and then allowed to interact with session oracles via the following queries:

- $\mathsf{Start}(\pi_S^{i,j})$. The challenger instructs a specified server oracle $\pi_S^{i,j}$ to execute $\mathsf{rchallenge}$ (if $j = 0$) or $\mathsf{achallenge}$ (if $j > 0$) to start the Register or Authenticate subprotocol and generate a challenge c, which is given to $\mathcal{A}$.
- $\mathsf{Challenge}(\pi_T^{i,j}, M)$. The challenger delivers a specified command M to a specified token oracle $\pi_T^{i,j}$, which processes the command using $\mathsf{rresponse}$ (if $j = 0$) or $\mathsf{aresponse}$ (if $j > 0$) and returns the response to $\mathcal{A}$.
- $\mathsf{Complete}(\pi_S^{i,j}, T, R)$. The challenger delivers a specified token response R to a specified server oracle $\pi_S^{i,j}$, which processes the response using rcheck and vk_T (if $j = 0$) or acheck (if $j > 0$) and returns the result to $\mathcal{A}$.

We assume without loss of generality that each query is only called once for each instance and allow the adversary to get the full state of the server via Start and $\mathsf{Complete}$ queries.

Partners. We follow the seminal work by Bellare *et al.* [9] to define partnership via *session identifiers*. A server registration oracle $\pi_S^{i,0}$ and a token registration oracle $\pi_T^{k,0}$ are each other's *partner* if they agree on the same session identifier, which indicates a "shared view" that must be defined by the analyzed protocol and must be the same for both parties, usually as a function of the communication trace. A server authentication oracle $\pi_S^{i,j}$ $(j > 0)$ and a token authentication oracle $\pi_T^{k,l}$ $(l > 0)$ are each other's *partner* if: i) they agree on the session identifier and ii) $\pi_S^{i,0}$ and $\pi_T^{k,0}$ are each other's partner.

We note that a crucial aspect of this definition is that the authentication session partnership holds only if the token and the server are also partnered for the associated registration sessions: a credential registered in a server should not be used to authenticate a token using another credential.

Advantage measure. Let Π be a PlA protocol. We define the passwordless authentication advantage $\mathbf{Adv}_{\Pi}^{\mathsf{pla}}(\mathcal{A})$ as the probability that a server oracle accepts but it is not uniquely partnered with a token oracle. In other words, a secure PlA protocol guarantees that, if a server oracle accepts, then there exists a unique token oracle that has derived the same session identifier, and no other server oracle has derived the same session identifier.

5 The W3C Web Authentication Protocol

In this section, we present the cryptographic core of W3C's Web Authentication (WebAuthn) protocol [15] of FIDO2 and analyze its security.

Protocol Description. We show the core cryptographic operations of WebAuthn in Fig. 3 in accordance with PlA syntax.[10] For WebAuthn, a server identity is an effective domain (e.g., a hostname) of the server URL. The attestation key pair is generated by the key generation algorithm Kg of a signature scheme $\mathsf{Sig} = (\mathsf{Kg}, \mathsf{Sign}, \mathsf{Ver})$. (Note that WebAuthn supports the RSASSA-PKCS1-v1_5 and RSASSA-PSS signature schemes [30].) In Fig. 3, we use H to denote the SHA-256 hash function and λ to denote the default parameter 128 (in order to accommodate potential parameter changes). WebAuthn supports two types of operations: Registeration and Authentication (cf. Fig. 1 and Fig. 2 in [15]), respectively corresponding to the PlA Register and Authenticate subprotocols. In the following description, we assume each token is registered at most once for a server; this is without loss of generality since otherwise one can treat the one token as several tokens sharing the same attestation key pair.

– In registration, the server generates a random string rs of length at least $\lambda = 128$ bits and a random 512-bit user id uid, forms a challenge cc with rs, uid and its identity id_S, and then sends it to the client. Then, the client checks if the received server identity matches its input (i.e., the intended server), then passes the received challenge (where the random string is hashed) to the token.

[10] We do not include the WebAuthn explicit reference to user interaction/gestures at this point, as this will be later handled by our PACA protocol.

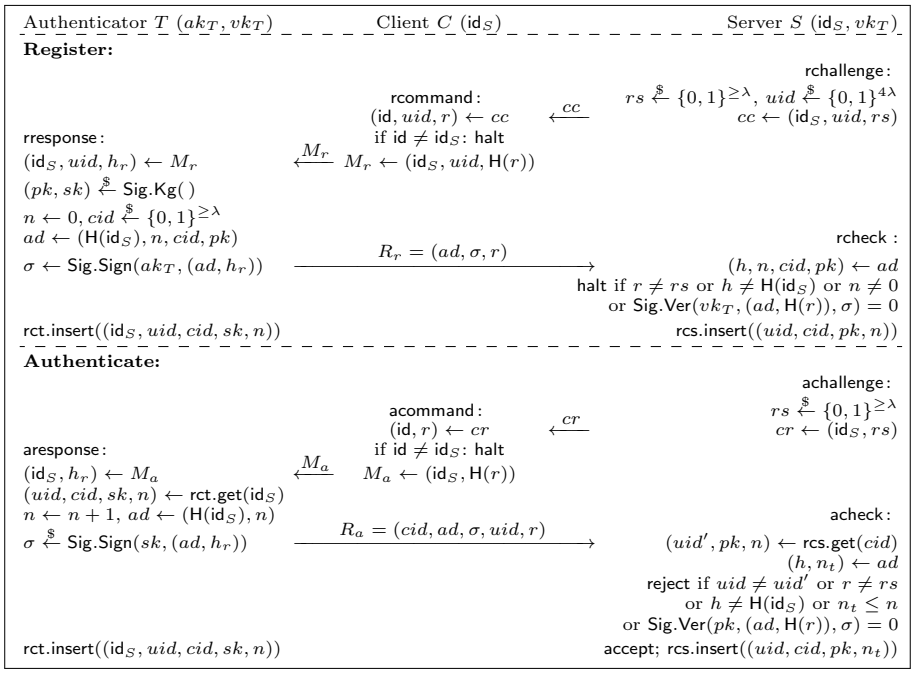

Fig. 3. The WebAuthn protocol

The token generates a key pair (pk, sk) with $\mathsf{Sig.Kg}$, sets the signature counter n to 0,[11] and samples a credential id cid of length at least $\lambda = 128$ bits; it then computes an attestation signature (on $\mathsf{H}(\mathsf{id}_S), n, cid, pk$ and the random string hash h_r) and sends the signed (public) credential and signature to the client as a response; the token also inserts the generated credential into its registration contexts. Upon receiving the response, the server checks the validity of the attestation signature and inserts the credential into its registration contexts.

– In authentication, the server also generates a random string rs, but no uid is sampled; it then forms a challenge cr with rs and its identity id_S, and sends it to the client. Then, the client checks if the received id_S matches its input and passes the challenge (where the random string is hashed) to the token. The token retrieves the credential associated with the authenticating server id_S from its registration contexts, increments the signature counter n, computes an authentication signature (on $\mathsf{H}(\mathsf{id}_S), n$ and the random string hash h_r), and sends it to the client together with $\mathsf{H}(\mathsf{id}_S), n$ and the retrieved credential id cid and user id uid; the token also updates the credential with the new signature counter. Upon receiving the response, the server retrieves the credential associated with the credential id cid and checks the validity of the signature counter and the signature; if all checks pass, it accepts and updates the credential with the new signature counter.

[11] The signature counter is mainly used to detect cloned tokens, but it also helps in preventing replay attacks (if such attacks are possible).

It is straightforward to check that WebAuthn is a correct PlA protocol.

WebAuthn Analysis. The following theorem (proved in the full version [6]) assesses PlA security of WebAuthn uses $(ad, \mathsf{H}(r))$ as the session identifier.

Theorem 1. *For any efficient adversary $\mathcal{A}$ that makes at most q_{S} queries to* Start *and q_{C} queries to* Challenge*, there exist efficient adversaries $\mathcal{B}, \mathcal{C}$ such that (recall $\lambda = 128$):*

$$\mathbf{Adv}^{\mathsf{pla}}_{\mathrm{WebAuthn}}(\mathcal{A}) \leq \mathbf{Adv}^{\mathsf{coll}}_{\mathsf{H}}(\mathcal{B}) + q_{\mathsf{S}}\mathbf{Adv}^{\mathsf{euf\text{-}cma}}_{\mathsf{Sig}}(\mathcal{C}) + (q_{\mathsf{S}}^2 + {q_{\mathsf{C}}}^2) \cdot 2^{-\lambda}.$$

The security guarantees for the WebAuthn instantiations follow from the results proving RSASSA-PKCS1-v1_5 and RSASSA-PSS to be EUF-CMA in the random oracle model under the RSA assumption [11,28] and the assumption that SHA-256 is collision-resistant.

6 PIN-Based Access Control for Authenticators

In this section, we define the syntax and security model for *PIN-based access control for authenticators* (PACA) protocols. The goal of the protocol is to ensure that after PIN setup and possibly an arbitrary number of authenticator reboots, the user can employ the client to issue PIN-authorized commands to the token, which the token can use for access control, e.g., to unlock built-in functionalities that answer client commands.

6.1 Protocol Syntax

A PACA protocol is an interactive protocol involving a human user, an authenticator (or token for short), and a client. The state of token T, denoted by st_T, consists of static storage $\mathsf{st}_T.\mathsf{ss}$ that remains intact across reboots and volatile storage $\mathsf{st}_T.\mathsf{vs}$ that gets reset after each reboot. $\mathsf{st}_T.\mathsf{ss}$ is comprised of: i) a private secret $\mathsf{st}_T.\mathsf{s}$ and ii) a public retries counter $\mathsf{st}_T.\mathsf{n}$, where the latter is used to limit the maximum number of consecutive failed active attacks (e.g., PIN guessing attempts) against the token. $\mathsf{st}_T.\mathsf{vs}$ consists of: i) power-up state $\mathsf{st}_T.\mathsf{ps}$ and ii) binding states $\mathsf{st}_T.\mathsf{bs}_i$ (together denoted by $\mathsf{st}_T.\mathsf{bs}$). A client C may also keep binding states, denoted by $\mathsf{bs}_{C,j}$. All states are initialized to the empty string ε.

A PACA protocol is associated with an arbitrary public gesture predicate G and consists of the following algorithms and subprotocols, all of which can be executed a number of times, except if stated otherwise:

Reboot: This algorithm represents a power-down/power-up cycle and it is executed by the authenticator *with mandatory user interaction*. We use $\mathsf{st}_T.\mathsf{vs} \xleftarrow{\$} \mathsf{reboot}(\mathsf{st}_T.\mathsf{ss})$ to denote the execution of this algorithm, which inputs its static storage and resets all *volatile* storage. Note that one should always run this algorithm to power up the token at the beginning of PACA execution.

Setup: This subprotocol is executed *at most once* for each authenticator. The user inputs a PIN through the client and the token inputs its volatile storage. In the end, the token sets up its *static* storage and the client (and through it the user) gets an indication of whether the subprotocol completed successfully.

Bind: This subprotocol is executed by the three parties to establish an access channel over which commands can be issued. The user inputs its PIN through the client, whereas the token inputs its static storage and power-up state. At the end of the subprotocol, each of the token and client that successfully terminates gets a (volatile) binding state and sets the session identifier. In either case (success or not), the token may update its static retries counter.[12] We assume the client always initiates this subprotocol once it gets the PIN from the user.

Authorize: This algorithm allows a client to generate authorized commands for the token. The client inputs a binding state $\mathsf{bs}_{C,j}$ and a command M. We denote $(M,t) \xleftarrow{\$} \mathsf{authorize}(\mathsf{bs}_{C,j}, M)$ as the generation of an authorized command.

Validate: This algorithm allows a token to verify authorized commands sent by a client with respect to a user decision (where the human user inputs the public gesture predicate G). The token inputs a binding state $\mathsf{st}_T.\mathsf{bs}_i$, an authorized command (M,t), and a user decision $d = \mathsf{G}(x,y)$. We denote $b \leftarrow \mathsf{validate}(\mathsf{st}_T.\mathsf{bs}_i, (M,t), d)$ as the validation performed by the token to obtain an accept or reject indication.

Correctness. For an arbitrary public predicate G, we consider any token T and any sequence of PACA subprotocol executions that includes the following (which may not be consecutive): i) a Reboot of T; ii) a successful Setup using PIN fixing $\mathsf{st}_T.\mathsf{ss}$ via some client; iii) a Bind with PIN creating token-side binding state $\mathsf{st}_T.\mathsf{bs}_i$ and client-side binding state $\mathsf{bs}_{C,j}$ at a client C; iv) authorization of command M by C as $(M,t) \xleftarrow{\$} \mathsf{authorize}(\mathsf{bs}_{C,j}, M)$; and v) validation by T as $b \leftarrow \mathsf{validate}(\mathsf{st}_T.\mathsf{bs}_i, (M,t), d)$. If no Reboot of T is executed after iii), then correctness requires that $b = 1$ if and only if $\mathsf{G}(x,y) = 1$ (i.e., $d = 1$) holds.

Remark. The above PACA syntax may seem overly complex but it is actually difficult (if not impossible) to decompose. First, Setup and Bind share the same power-up state generated by Reboot so cannot be separated into two independent procedures. Then, although Authorize and Validate together can independently model an access channel, detaching them from PACA makes it difficult to define security in a general way: Bind may not establish random symmetric keys; it could, for instance, output asymmetric key pairs.

6.2 Security Model

Trust model. Before defining our security model, we first state the assumed security properties for the involved communication channels, as shown in Fig. 1 excluding the client-server channel. We assume that Setup is carried out over an authenticated channel where the adversary can only eavesdrop communications between the client and authenticator; this is a necessary assumption, as there are no pre-established authentication parameters between the parties.

[12] When such an update is possible, the natural assumption often made in cryptography requires that incoming messages are processed in an atomic way by the token, which avoids concurrency issues. Note that Bind executions could still be concurrent.

Session oracles. To capture multiple sequential and parallel PACA executions, each party $P \in \mathcal{T} \cup \mathcal{C}$ is associated with a set of session oracles $\{\pi_P^i\}_i$, where π_P^i models the i-th PACA instance of P. For clients, session oracles are totally independent from each other and they are assumed to be available throughout the protocol execution. For tokens, the static storage and power-up state are maintained by the security experiment and shared by all oracles of the same token. Token oracles keep only binding states (if any). If a token is rebooted, its binding states got reset and hence become *invalid*, i.e., those states will be no longer accessible to anyone including the adversary.

Security experiment. The security experiment is executed between a challenger and an adversary $\mathcal{A}$. At the beginning of the experiment, the challenger fixes an arbitrary distribution $\mathscr{D}$ over a PIN dictionary $\mathcal{PIN}$ associated with PACA; it then samples independent user PINs according to $\mathscr{D}$, denoted by $\langle \mathsf{pin}_U \xleftarrow{\mathscr{D}} \mathcal{PIN} \rangle_{U \in \mathcal{U}}$. Without loss of generality, we assume each user holds only one PIN. The challenger also initializes states of all oracles to the empty string. Then, $\mathcal{A}$ is allowed to interact with the challenger via the following queries:

- $\mathsf{Reboot}(T)$. The challenger runs Reboot for token T, marking all previously used instances π_T^i (if any) as *invalid*[13] and setting $\mathsf{st}_T.\mathsf{vs} \xleftarrow{\$} \mathsf{reboot}(\mathsf{st}_T.\mathsf{ss})$.
- $\mathsf{Setup}(\pi_T^i, \pi_C^j, U)$. The challenger inputs pin_U through π_C^j and runs Setup between π_T^i and π_C^j; it returns the trace of communications to $\mathcal{A}$. After this query, T is *set up*, i.e., $\mathsf{st}_T.\mathsf{ss}$ is set and available, for the rest of the experiment. Oracles created in this query, i.e., π_T^i and π_C^j, must never have been used before and are always marked *invalid* after Setup completion.[14]
- $\mathsf{Execute}(\pi_T^i, \pi_C^j)$. The challenger runs Bind between π_T^i and π_C^j using the same pin_U that set up T; it returns the trace of communications to $\mathcal{A}$. This query allows the adversary to access honest Bind executions in which it can only take passive actions, i.e., eavesdropping. The resulting binding states on both sides are kept as $\mathsf{st}_T.\mathsf{bs}_i$ and $\mathsf{bs}_{C,j}$ respectively.
- $\mathsf{Connect}(T, \pi_C^j)$. The challenger asks π_C^j to initiate the Bind subprotocol with T using the same pin_U that set up T; it returns the first message sent by π_C^j to $\mathcal{A}$. Note that no client oracles can be created for active attacks if $\mathsf{Connect}$ queries are disallowed, since we assume the client is the initiator of Bind. This query allows the adversary to launch an active attack against a client oracle.
- $\mathsf{Send}(\pi_P^i, m)$. The challenger delivers m to π_P^i and returns its response (if any) to $\mathcal{A}$. If π_P^i completes the Bind subprotocol, then the binding state is kept as $\mathsf{st}_T.\mathsf{bs}_i$ for a token oracle and as $\mathsf{bs}_{C,i}$ for a client oracle. This query allows the adversary to launch an active attack against a token oracle or completing an active attack against a client oracle.
- $\mathsf{Authorize}(\pi_C^j, M)$. The challenger asks π_C^j to authorize command M; it returns the authorized command $(M, t) \xleftarrow{\$} \mathsf{authorize}(\mathsf{bs}_{C,j}, M)$.

[13] All queries are ignored if they refer to an oracle π_P^i marked as invalid.

[14] Session oracles used for Setup are separated since they may cause ambiguity in defining session identifiers for binding sessions.

• $\mathsf{Validate}(\pi_T^i, (M,t))$. The challenger asks π_T^i (that received a user decision d) to validate (M,t); it returns the validation result $b \leftarrow \mathsf{validate}(\mathsf{st}_T.\mathsf{bs}_i, (M,t), d)$.
• $\mathsf{Compromise}(\pi_C^j)$. The challenger returns $\mathsf{bs}_{C,j}$ and marks π_C^i as *compromised*.
• $\mathsf{Corrupt}(U)$. The challenger returns pin_U and marks pin_U as *corrupted*.

Partners. We say a token oracle π_T^i and a client oracle π_C^j in binding sessions are each other's *partner* if they have both completed their Bind executions and agree on the same session identifier. As with our PIA model, session identifiers must be properly defined by the analyzed protocol. Moreover, we also say π_C^j is T's *partner* (and hence T may have multiple partners). Note that, as mentioned before, if a token is rebooted then all of its existing session oracles (if any) are invalidated. A *valid* partner refers to a valid session oracle.

Security goals. We define 4 levels of security for a PACA protocol Π. All advantage measures define PAKE-like security: the adversary's winning probability should be negligibly larger than that of the trivial attack of guessing the user PIN (known as *online dictionary attacks* with more details in the full version [6]).

Unforgeability (UF). We define $\mathbf{Adv}_{\Pi}^{\mathsf{uf}}(\mathcal{A})$ as the probability that there exists a token oracle π_T^i that accepts an authorized command (M,t) for gesture G and at least one of the following conditions does *not* hold:

1) G approves M, i.e., $\mathsf{G}(x,y) = 1$;
2) (M,t) was output by one of T's valid partners π_C^j.

The adversary must be able to trigger this event without: i) corrupting pin_U that was used to set up T, before π_T^i accepted (M,t); or ii) compromising any of T's partners created after T's last reboot and before π_T^i accepted (M,t).

The above captures the attacks where the attacker successfully makes a token accept a forged command, without corrupting the user PIN used to set up the token or compromising any of the token's partners. In other words, a UF-secure PACA protocol protects the token from unauthorized access even if it is stolen and possessed by an attacker. Nevertheless, UF considers only weak security for access channels, i.e., compromising one channel could result in compromising all channels (with respect to the same token after its last reboot).

Unforgeability with trusted binding (UF-t). We define $\mathbf{Adv}_{\Pi}^{\mathsf{uf\text{-}t}}(\mathcal{A})$ the same as $\mathbf{Adv}_{\Pi}^{\mathsf{uf}}(\mathcal{A})$ except that the adversary is *not* allowed to make $\mathsf{Connect}$ queries.

As mentioned before, the attacker is now forbidden to launch active attacks against clients (that input user PINs) during binding; it can still, however, perform active attacks against tokens. This restriction captures the minimum requirement for proving the security of CTAP2 (using our model), which is the main reason we define UF-t. Clearly, UF security implies UF-t security.

Strong unforgeability (SUF). We define $\mathbf{Adv}_{\Pi}^{\mathsf{suf}}(\mathcal{A})$ as the UF advantage, with one more condition captured:

3) π_T^i and π_C^j are each other's unique valid partner.

More importantly, the adversary considered in this strong notion is allowed to compromise T's partners, provided that it has not compromised π_C^j. It is also allowed to corrupt pin_U used to set up T even before the command is accepted, *as long as* π_T^i has set its binding state.

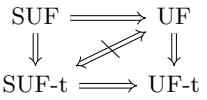

Fig. 4. Relations between PACA security notions.

The above captures similar attacks considered in UF but in a strong sense, where the attacker is allowed to compromise the token's partners. This means SUF considers strong security for access channels, i.e., compromising any channel does not affect other channels. It hence guarantees a unique binding between an accepted command and an access channel (created by uniquely partnered token and client oracles running Bind), which explains condition 3). Finally, the attacker is further allowed to corrupt the user PIN *immediately* after the access channel establishment. This guarantees *forward secrecy* for access channels, i.e., once the channel is created its security will no longer be affected by later PIN corruption. Note that SUF security obviously implies UF security.

Strong unforgeability with trusted binding (SUF-t). For completeness we can also define $\mathbf{Adv}_{\Pi}^{\mathsf{suf\text{-}t}}(\mathcal{A})$, where the adversary is *not* allowed to make $\mathsf{Connect}$ queries. Again, it is easy to see that SUF security implies SUF-t security.

Relations between PACA security notions. Fig. 4 shows the implication relations among our four defined notions. Note that UF and SUF-t do not imply each other, for which we will give separation examples in Sections 7 and 8.

Improving (S)UF-t security with user confirmation. Trusted binding excludes active attacks against the client (during binding), but online dictionary attacks are still possible against the token. Such attacks can be mitigated by requiring user confirmation (e.g., pressing a button) for Bind execution, such that only honest Bind executions will be approved when the token is possessed by an honest user. We argue that the confirmation overhead is quite small for CTAP2-like protocols since the user has to type its PIN into the client anyway; the security gain is meaningful as now *no* online dictionary attacks (that introduce non-negligible adversarial advantage) can happen to unstolen tokens.

A practical implication of SUF security. We note that SUF security has a practical meaning: an accepted command can be traced back to a unique access channel. This means that an authenticator that allows a human user to confirm a session identifier (that determines the channel) for a command can allow a human user to detect rogue commands issued by an adversary (e.g., malware) that compromised one of the token's partners (e.g., browsers).

PACA security bounds. In our theorems for PACA security shown later, we fix q_{S} (i.e., the number of Setup queries) as one adversarial parameter to bound the adversary's success probability of online dictionary attacks (e.g., the first bound term in Theorem 2 and the PAKE advantage term in Theorem 3), while for PAKE security the number of SEND queries q_s is used (see [9] or the full version [6] for example). This is because PACA has a token-side retries counter to limit the total number of failed PIN guessing attempts (across reboots).

7 The Client to Authenticator Protocol v2.0

In this section, we present the cryptographic core of the FIDO Alliance's CTAP2, analyze its security using PACA model, and make suggestions for improvement.

Protocol Description. CTAP2's cryptographic core lies in its authenticator API[15] which we show in Fig. 5 in accordance with PACA syntax. One can also refer to its specification (Fig. 1, [1]) for a command-based description.[16] The PIN dictionary $\mathcal{PIN}$ of CTAP2 consists of 4∼63-byte strings.[17] In Fig. 5, the client inputs an arbitrary user PIN $\mathsf{pin}_U \in \mathcal{PIN}$. We use $\mathsf{ECKG}_{\mathbb{G},G}$ to denote the key generation algorithm of the NIST P-256 elliptic-curve Diffie-Hellman (ECDH) [26], which samples an elliptic-curve secret and public key pair (a, aG), where G is an elliptic-curve point that generates a cyclic group $\mathbb{G}$ of prime order $|\mathbb{G}|$ and a is chosen at random from the integer set $\{1, \ldots, |\mathbb{G}|-1\}$. Let H denote the SHA-256 hash function and H' denote SHA-256 with output truncated to the first $\lambda = 128$ bits; $\mathsf{CBC}_0 = (\mathcal{K}, \mathsf{E}, \mathsf{D})$ denotes the (deterministic) encryption scheme AES-256-CBC [20] with fixed IV = 0; HMAC' denotes the MAC HMAC-SHA-256 [8] with output truncated to the first $\lambda = 128$ bits. Note that we use the symbol λ to denote the block size in order to accommodate parameter changes in future versions of CTAP2.

- Reboot generates $\mathsf{st}_T.\mathsf{ps}$ by running $\mathsf{ECKG}_{\mathbb{G},G}$, sampling a $k\lambda$-bit pinToken pt (where $k \in \mathbb{N}_+$ can be any fixed parameter, e.g., $k = 2$ for a 256-bit pt), and resetting the mismatch counter $m \leftarrow 3$ that limits the maximum number of consecutive mismatches. It also erases the binding state $\mathsf{st}_T.\mathsf{bs}$ (if any).
- Setup is essentially an unauthenticated ECDH followed by the client transmitting the (encrypted) user PIN to the token. The shared encryption key is derived from hashing the x-coordinate of the ECDH result. A HMAC' tag of the encrypted PIN is also attached for authentication; but as we will show this is actually useless. The token checks if the tag is correct and if the decrypted PIN pin_U is valid; if so, it sets the static secret $\mathsf{st}_T.\mathsf{s}$ to the PIN hash and sets the retries counter $\mathsf{st}_T.\mathsf{n}$ to the default value 8.
- Bind also involves an unauthenticated ECDH but followed by the transmission of the encrypted PIN hash. First, if $\mathsf{st}_T.\mathsf{n} = 0$, the token blocks further access unless being reset to factory default state, i.e., erasing all static and volatile state. Otherwise, the token decrements $\mathsf{st}_T.\mathsf{n}$ and checks if the decrypted PIN

[15] The rest of CTAP2 does not focus on security but specifies transport-related behaviors like message encoding and transport-specific bindings.

[16] There the command used for accessing the retries counter $\mathsf{st}_T.\mathsf{n}$ is omitted because PACA models it as public state. Commands for PIN resets are also omitted and left for future work, but capturing those is not hard by extending our analysis since CTAP2 changes PIN by simply running the first part of Bind (to establish the encryption key and verify the old PIN) followed by the last part of Setup (to set a new PIN). Without PIN resets, our analysis still captures CTAP2's core security aspects and our PACA model becomes more succinct.

[17] PINs memorized by users are at least 4 Unicode characters and of length at most 63 bytes in UTF-8 representation.

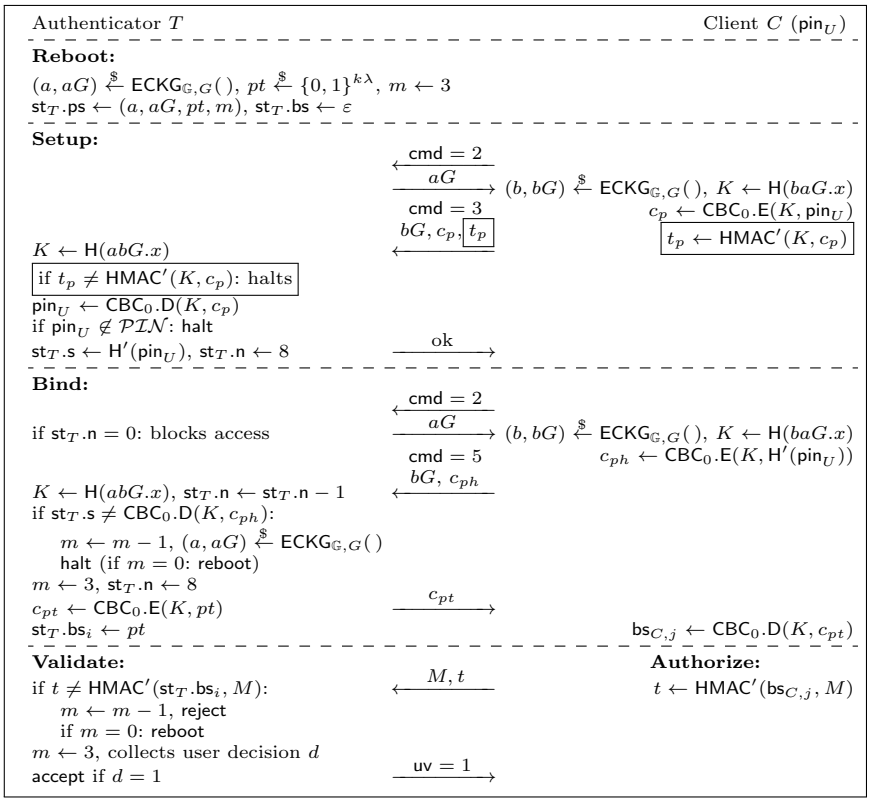

Fig. 5. The CTAP2 protocol (and CTAP2* that excludes the boxed contents).

hash matches its stored static secret. If the check fails, it decrements the mismatch counter m, generates a new key pair, then halts; if $m = 0$, it further requires a reboot to enforce user interaction (and hence user detectability). If the check passes, it resets the retries counter, sends back the encrypted pinToken, and uses its pinToken as the binding state $\mathsf{st}_T.\mathsf{bs}_i$; the client then uses the decrypted pinToken as its binding state $\mathsf{bs}_{C,j}$.

– Authorize generates an authorized command by attaching a HMAC' tag.
– Validate accepts the command if and only if the tag is correct and the user gesture approves the command. The default CTAP2 gesture predicate G_1 always returns true, since only physical user presence is required. The mismatch counter is also updated to trigger user interaction.

It is straightforward to check that CTAP2 is a correct PACA protocol.

CTAP2 Analysis. The session identifier of CTAP2 is defined as the full communication trace of the Bind execution.

Insecurity of CTAP2. It is not hard to see that CTAP2 is not UF-secure (and hence not SUF-secure). An attacker can query Connect to initiate the Bind execution of a client oracle that inputs the user PIN, then impersonate the token to get the PIN hash, and finally use it to get the secret binding state pt from the

token. CTAP2 is not SUF-t-secure either because compromising any partner of the token reveals the common binding state pt used to access all token oracles.

UF-t security of CTAP2. The following theorem (proved in the full version [6]) confirms CTAP2's UF-t security, by modeling the hash function H (with fixed 256-bit input) and truncated HMAC HMAC' as random oracles $\mathcal{H}_1, \mathcal{H}_2$.

Theorem 2. *Let $\mathscr{D}$ be an arbitrary distribution over $\mathcal{PIN}$ with min-entropy $h_{\mathscr{D}}$. For any efficient adversary $\mathcal{A}$ making at most $q_{\mathsf{S}}, q_{\mathsf{E}}, q_{\mathsf{R}}, q_{\mathsf{V}}$ queries respectively to* Setup, Execute, Reboot, Validate*, and $q_{\mathcal{H}}$ random oracle queries to $\mathcal{H}_2$, there exist efficient adversaries $\mathcal{B}, \mathcal{C}, \mathcal{D}$ such that (recall $\lambda = 128$):*

$$\begin{aligned}\mathbf{Adv}^{\mathsf{uf\text{-}t}}_{\mathrm{CTAP2}}(\mathcal{A}) \leq\ & 8q_{\mathsf{S}} \cdot 2^{-h_{\mathscr{D}}} + (q_{\mathsf{S}} + q_{\mathsf{E}})\mathbf{Adv}^{\mathsf{scdh}}_{\mathbb{G},G}(\mathcal{B}) + \mathbf{Adv}^{\mathsf{coll}}_{\mathsf{H}'}(\mathcal{C}) \\ & + 2(q_{\mathsf{S}} + q_{\mathsf{E}})\mathbf{Adv}^{\mathsf{prf}}_{\text{AES-256}}(\mathcal{D}) + q_{\mathsf{V}} \cdot 2^{-k\lambda} + q_{\mathsf{S}}q_{\mathcal{H}} \cdot 2^{-2\lambda} \\ & + (12q_{\mathsf{S}} + 2|\mathcal{U}|q_{\mathsf{R}}q_{\mathsf{E}} + q_{\mathsf{R}}^2 q_{\mathsf{E}} + (k+1)^2 q_{\mathsf{E}} + q_{\mathsf{V}}) \cdot 2^{-\lambda}.\end{aligned}$$

We remark that for conciseness the above theorem does not show what security should be achieved by CBC_0 for CTAP2's UF-t security to hold, but directly reduces to the PRF security of the underlying AES-256 cipher. Actually, the proof of the above theorem also shows that it is sufficient for CBC_0 to achieve a novel security notion that we call *indistinguishability under one-time chosen and then random plaintext attack (IND-1$PA)*, which (defined in the full version [6]) we think would be of independent interest. We prove in the full version [6] that the IND-1$PA security of CBC_0 can be reduced to the PRF security of AES-256.

SUF-t $\not\Rightarrow$ UF. Note that we can modify CTAP2 to achieve SUF-t security by using independent pinTokens for each Bind execution, but this is not UF-secure due to unauthenticated ECDH. This shows that SUF-t does not imply UF.

CTAP2 improvement. Here we make suggestions for improving CTAP2 per se, but we advocate the adoption of our proposed efficient PACA protocol with stronger SUF security in Section 8.

Setup simplification. First, we notice that the Setup authentication procedures (boxed in Fig. 5) are useless, since there are no pre-established authentication parameters between the token and client. In particular, a MITM attacker can pick its own aG to compute the shared key K and generate the authentication tag. More importantly, CTAP2 uses the same key K for both encryption and authentication, which is considered bad practice and the resulting security guarantee is elusive; this is why we have to model HMAC' as a random oracle. Therefore, we suggest removing those redundant authentication procedures (or using checksums), then the resulting protocol, denoted by CTAP2*, is also UF-t-secure, with the proof in the full version [6] where HMAC' is treated as an EUF-CMA-secure MAC.[18] Furthermore, one can use a simple one-time pad (with appropriate key expansion) instead of CBC_0 to achieve the same UF-t security. This is because only one encryption is used in Setup and hence one-time security provided by a one-time pad is sufficient.

[18] Note that HMAC-SHA-256 has been proved to be a PRF (and hence EUF-CMA) assuming SHA-256's compression function is a PRF [7].

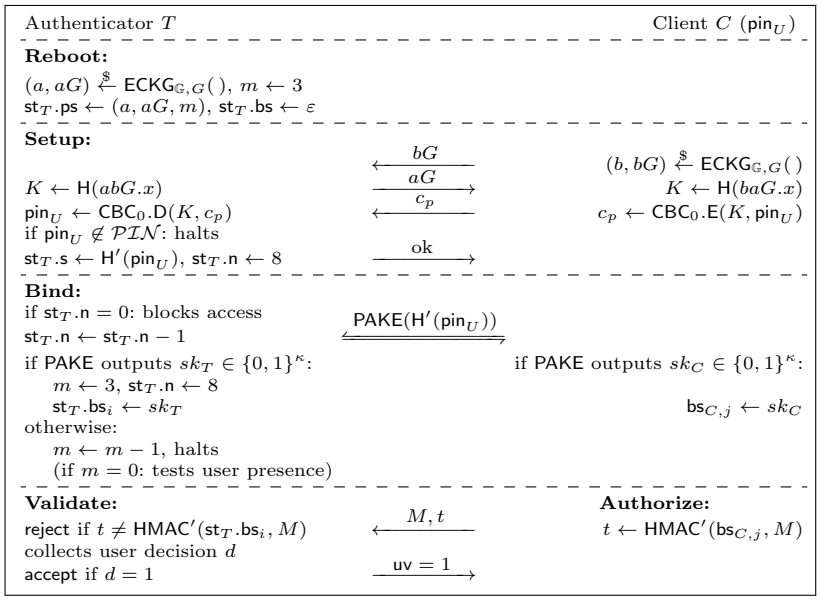

Fig. 6. The sPACA protocol

Unnecessary reboots. In order to prevent attacks that block the token without user interaction, CTAP2 requires a token reboot after 3 consecutive failed binding attempts. Such reboots do not enhance security as the stored PIN hash is not updated, but they could cause usability issues since reboots invalidate all established access channels by erasing the existing binding states. We therefore suggest replacing reboots with tests of user presence (e.g., pressing a button) that do not affect existing bindings. Note that reboots are also introduced for user interaction in Validate executions; this however is completely useless when CTAP2 already requires a test of user presence before accepting each command.

User confirmation for binding. As discussed at the end of Section 6, we suggest CTAP2 require user confirmation for Bind executions to improve security. Note that here user confirmation is used to detect and prevent malicious Bind executions rather than confirming honest ones.

8 The Secure PACA Protocol

In this section, we propose a generic PACA protocol that we call sPACA for *secure PACA*, prove its SUF security, and compare its performance with CTAP2 when instantiating the underlying PAKE of sPACA with CPace [24].

Protocol Description. We purposely design our sPACA protocol following CTAP2 such that the required modification is minimized if sPACA is adopted. As shown in Fig. 6, sPACA employs the same PIN dictionary $\mathcal{PIN}$ and cryptographic primitives as CTAP2 and additionally relies on a PAKE protocol PAKE initiated by the client. Compared to CTAP2, sPACA does not have pinTokens,

but instead establishes independent random binding states in Bind executions by running PAKE between the token and the client (that inputs the user PIN) on the shared PIN hash; it also excludes unnecessary reboots. We also note that the length of session keys $sk_T, sk_C \in \{0,1\}^\kappa$ established by PAKE is determined by the concrete PAKE instantiation; typically $\kappa \in \{224, 256, 384, 512\}$ when the keys are derived with a SHA-2 hash function.

sPACA Analysis. The session identifier of sPACA is simply that of PAKE.

SUF security of sPACA. The following theorem (proved in the full version [6]) confirms SUF security of sPACA by modeling H as a random oracle.

Theorem 3. *Let* PAKE *be a 3-pass protocol where the client is the initiator and let* $\mathscr{D}$ *be an arbitrary distribution over* $\mathcal{PIN}$ *with min-entropy* $h_{\mathscr{D}}$*. For any efficient adversary* $\mathcal{A}$ *making at most* $q_{\mathsf{S}}, q_{\mathsf{C}}, q_{\mathsf{E}}$ *queries respectively to* Setup, Connect, Execute*, there exist efficient adversaries* $\mathcal{B}, \mathcal{C}, \mathcal{D}, \mathcal{E}, \mathcal{F}$ *such that:*

$$\begin{aligned}\mathbf{Adv}^{\mathsf{suf}}_{\mathrm{sPACA}}(\mathcal{A}) &\leq q_{\mathsf{S}}\mathbf{Adv}^{\mathsf{cdh}}_{\mathbb{G},G}(\mathcal{B}) + \mathbf{Adv}^{\mathsf{coll}}_{\mathsf{H}'}(\mathcal{C}) + 2q_{\mathsf{S}}\mathbf{Adv}^{\mathsf{prf}}_{\text{AES-256}}(\mathcal{D}) \\ &+ \mathbf{Adv}_{\mathsf{PAKE}}(\mathcal{E}, 16q_{\mathsf{S}} + 2q_{\mathsf{C}}, h_{\mathscr{D}}) + (q_{\mathsf{C}} + q_{\mathsf{E}})\mathbf{Adv}^{\mathsf{euf\text{-}cma}}_{\mathsf{HMAC}'}(\mathcal{F}) + 12q_{\mathsf{S}} \cdot 2^{-\lambda}.\end{aligned}$$

Note that it is crucial for PAKE to guarantee *explicit* authentication, otherwise, the token might not be able to detect wrong PIN guesses and then decrement its retries counter to prevent exhaustive PIN guesses.[19] Also note that the PAKE advantage bound may itself include calls to an independent random oracle. PAKE can be instantiated with variants of CPace [24] or SPAKE2 [3, 5] that include explicit authentication. Both protocols were recently considered by the IETF for standardization and CPace was selected in the end.[20] They both meet the required security property, as they have been proved secure in the UC setting which implies the game-based security notion we use [4, 24].

UF $\not\Rightarrow$ SUF-t. Note that one can easily transform sPACA into a protocol that is still UF secure, but not SUF-t secure: similar to CTAP2, let the authenticator generate a global pinToken used as binding states for all its partners and send it (encrypted with the session key output by PAKE) to its partners at the end of Bind executions. This shows that UF does not imply SUF-t.

Performance comparison of CTAP2 and sPACA. It is straightforward to see from Fig. 5 and Fig. 6 that CTAP2 and sPACA differ mainly in their Bind executions, while sPACA has slightly better performance than CTAP2 in other subprotocols. We therefore compare their performance for binding (where sPACA is instantiated with CPace) in terms of message flows, computations (for group exponentiations, hashes, AES) on both sides, and communication complexity. Among these three factors, the number of flows reflects the network

[19] One does not actually need explicit token-to-client authentication in the proof, as clients do not have long-term secret to protect. This would allow removing the server-side authentication component from the PAKE instantiation for further efficiency. We do not propose to do this and choose to rely on the standard *mutual* explicit authentication property to enable direct instantiation of a standardized protocol.

[20] `https://mailarchive.ietf.org/arch/msg/cfrg/j88r8N819bw88xCOyntuw_Ych-I`

Table 1. Performance comparison of CTAP2 and sPACA for binding.

Protocol	Flow	Token			Client			Communication
		exp	hash	AES	exp	hash	AES	($\lambda = 128$)
CTAP2	4	2	1	$2k$	2	2	$2k$	$4\lambda + 2k\lambda$ (e.g., $k = 2$)
sPACA[CPace]	3	2	4	0	2	5	0	$4\lambda + 2\kappa$ (e.g., $\kappa = 256$)

latency cost that usually dominates the performance. Therefore, one can observe that sPACA (with CPace) is more efficient than CTAP2 from the results summarized in Table 1, which we explain as follows.

First, CPace needs 3 flows when explicit authentication is required and hence so does sPACA, while CTAP2 needs 4. Besides, if Bind is executed when the client already has a command to issue, the last CPace message can be piggybacked with the authorized command, leading to a very efficient 2-flow binding.[21] As shown in Fig. 5, CTAP2 requires two Diffie-Hellman group exponentiations and $2k$ AES computations (for pt of k-block length) on both sides; the token computes one hash while the client computes two (one for hashing PIN). For sPACA, CPace requires two Diffie-Hellman group exponentiations and four hashes on both sides; the client also needs to compute the PIN hash beforehand. In short, sPACA incurs 3 more hashes while CTAP2 involves $2k$ more AES computations. Note that the most expensive computations are group exponentiations, for which both protocols have two. Regarding communication complexity, both protocols exchange two group elements and two messages of the same length as the binding states, so they are equal if, say, $\kappa = k\lambda = 256$. Overall, sPACA (with CPace) is more efficient than CTAP2 due to less flows.

Finally, we note that the cryptographic primitives in sPACA could be instantiated with more efficient ones compared to those in CTAP2 without compromising security. For instance, as mentioned before, one can use a very efficient one-time pad (with appropriate key expansion) instead of CBC_0 in Setup.

9 Composed Security of PlA and PACA

In this section we discuss the composed security of PlA and PACA and the implications of this composition for FIDO2 and WebAuthn+sPACA. The composed protocol, which we simply refer to as PlA+PACA, is defined in the natural way, and it includes all the parties that appear in Fig. 1. We give a typical flow for registration in Fig. 7, where we assume PACA Setup and Bind have been correctly executed. The server's role is purely that of a PlA server. The client receives the server challenge via an authenticated channel (i.e., it knows the true server identity id_S when it gets a challenge from the server). It then authorizes the challenge using the PACA protocol and sends it to the authenticator. The authenticator first validates the PACA command (possibly using a user gesture) and, if successful, it produces a PlA response that is conveyed to the server.

[21] This piggy backing has the extra advantage of associating the end of the binding state with a user gesture by default, which helps detect online dictionary attacks against the token as stated in Section 6.

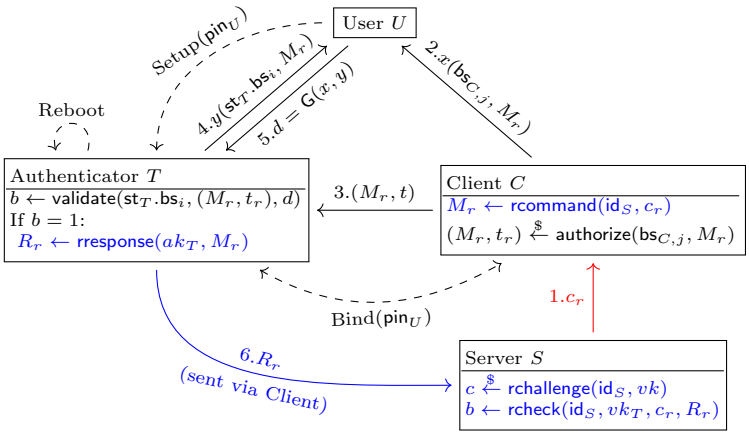

Fig. 7. Full PlA+PACA registration flow: black = PACA, blue = PlA, red = authenticated (e.g., TLS), dashed = PACA algorithms/subprotocols.

The flow for authentication looks exactly the same, apart from the fact that the appropriate PlA authentication algorithms are used instead. The requirement on the token is that it supports the combined functionalities of PlA and PACA protocols and that it is able to validate the correct authorization of two types of commands, (M_r, t_r) and (M_a, t_a), that correspond to PlA registration and authentication. These commands are used to control access to the PlA registration and authentication functionalities. In the full version of this paper [6] we formally give a syntax for such composed protocols.

A crucial aspect of our security results is that we convey the *two-sided* authentication guarantees offered by PlA+PACA, and not only the server-side guarantees. In fact, the server-side guarantees given by the composed protocol are almost those offered by PlA, as the server is simply a PlA server: if a token was used to register a key, then the server can recognize the same token in authentication; furthermore, PACA security requires that the authentication must have been carried by a PACA-bound client. But how do the client and user know which server they are registering at? What guarantees does a user have such that registered credentials cannot be used in a different server? What does a user know about how client security affects the effectiveness of access control for the token? We answer these questions next.

Security model. We give a very short description of the security model here (the details are in the full version [6]). We define a security property called *user authentication (UA)* for the composed protocol. We analyze the PlA+PACA composition in a trust model as with our PACA model but we further require a server-to-client explicit authentication guarantee. This captures a basic guarantee given by TLS, whereby the client knows the true identity of the server that generates the challenge and is ensured the integrity of the received challenge; it allows formalizing explicit server authentication guarantees given to the token and user by the composed protocol. We allow the adversary to create arbitrary

bindings between clients and tokens, used to deliver arbitrary commands to those created token oracles. We model server-to-token interactions via a unified query: the adversary can request challenges from server S, via client C aimed at a specific client-token PACA binding. We hardwire the server's true identity to the challenges, which is justified by our assumption of an authenticated channel from server to client. The token oracles are modeled in the obvious way: if a PACA command is accepted, then it is interpreted as in the PlA security experiment and the response is given to the adversary. Compromise of binding states and corruption of user PINs are modeled as in the PACA security experiment.

Security guarantees. The security goal we define for the composed protocol requires that a server oracle that accepts is uniquely partnered with a token oracle, which is associated with a unique PACA-bound client oracle (that has established an access channel), and these oracles agree on the exchanged messages in all passes of the challenge-response authentication session; this also holds for the associated registration session. We show that such server-side security for the composed protocol follows from security of its PlA and PACA components. Then, it is not hard to see that PlA correctness guarantees the above token and client oracles agree on the accepting server's identity and that PlA correctness and server-to-client explicit authentication (e.g., offered by TLS) guarantees that user approval (i.e., $d = 1$) via an uncompromised access channel implies that only the intended server can be authenticated to.

We now give a brief intuition on how the server-side result can be proved assuming the underlying PlA and PACA components are secure. Suppose a server authentication oracle $\pi_S^{i,j}$ $(j > 0)$ accepts and its associated server registration oracle $\pi_S^{i,0}$ took as input the attestation public key of token T:

- PlA security guarantees a unique partner oracle in T, which determines two partner token oracles: $\pi_T^{k,0}$ for registration and $\pi_T^{k,l}$ $(l > 0)$ for authentication.
- Token oracles are, by construction, created on acceptance of PACA commands. Therefore, token T must have accepted PACA commands to create the above PlA partner token oracles.
- PACA security binds a PACA command accepted by the token to a unique PACA partner client oracle (in the SUF/SUF-t corruption model) or to a set of PACA partner client oracles (in the UF/UF-t corruption model).
- PlA security also guarantees unique server-side partnered oracles $\pi_S^{i,0}$ and $\pi_S^{i,j}$ (which generated a challenge that is consistent with the token's view); this implies that the two accepted PACA commands are produced respectively by unique PACA partner client oracles π_C^m and π_C^n (in either corruption model), i.e., π_C^m has a consistent view with $\pi_S^{i,0}$ and $\pi_T^{k,0}$ in registration and so does π_C^n with $\pi_S^{i,j}$ and $\pi_T^{k,l}$ in authentication.

The above argument guarantees that *unique* server, token and client oracles are bound to the execution of PlA+PACA registration and authentication, as we claimed before. If this does not hold, then either the PlA protocol or the PACA protocol can be broken (reduction to the PACA protocol security can be done by considering the same corruption model as in PlA+PACA).

The details are in the full version of this paper [6].

Implications for FIDO2. The above result implies that FIDO2 components WebAuthn and CTAP2 securely compose to achieve the UA security guarantees under a weak corruption model UF-t: the protocol is broken if the adversary can corrupt *any* client that has access to the target token since the last power-up, or if the adversary can launch an *active* attack against an uncorrupted client (that the target user inputs its PIN into) via the CTAP2 API (i.e., the user thinks it is embedding the PIN into the token but it is actually giving it to the adversary). Such attacks are excluded by the trust model assumed for the client platform.

Security in the SUF model. The above result also implies that WebAuthn composes with our sPACA protocol from Section 8 to give UA security in the strongest corruption model we considered. Intuitively, no active attacks against the Bind subprotocol can help the attacker beyond simply guessing the user PIN. The corruption of clients (e.g., browsers) that have previously been bound to the token may be detected with the help of the user.

User gestures can upgrade security. UA gives strong guarantees to the server and client. However, it is not very clear what guarantees it gives to the human user. Apparently, there is a guarantee that an attacker that does not control the token cannot force an authentication, as it will be unable to provide a gesture. Furthermore, an attacker that steals the token must still guess the PIN in a small number of tries to succeed in impersonating the user.

One very important aspect of user awareness is to deal with malware attacks that may corrupt clients that have been bound to the token. Here, assuming SUF security has been established, the user can help prevent attackers from abusing the binding, provided that the token supports gestures that permit identifying the client-to-token access channel that is transmitting each command. In the weaker UF model there is no way to prevent this kind of abuse, as corrupting one access channel implies corrupting all access channels to the same token.

Gestures can also be used to give explicit guarantees to the user that the server identity used in a PlA session is the intended one. For example, there could be ambiguity with multiple (honest and malicious) client browser windows issuing concurrent commands from multiple servers. Suppose gesture G permits confirming which client session is issuing the registration and authentication commands.[22] In this case we get a strong guarantee that the token registered a credential or authenticated via an honest client in the server with identifier $\mathsf{id}_S^\star$, where $\mathsf{id}_S^\star$ was explicitly confirmed by the user on the client interface, provided that the honest client session issued only one command to the token. Alternatively, G can be defined to directly confirm the specific $\mathsf{id}_S^\star$ value that can be displayed by the authenticator itself and we get the same guarantee.

If the gesture cannot confirm consistency between client and token, then the user will not be able to distinguish which access channel is transmitting the PlA command and know for sure which id_S the command it is approving refers to. However, our composition result does show that trivial gestures are sufficient if the user establishes only one access channel with the token per power-up, as

[22] Confirming a client session means that the client browser and token somehow display a human-readable identifier that the user can crosscheck and confirm.

then there is no ambiguity as to which access channel is used and only a single client is provided with the intended server identity as input.

10 Conclusion

We performed the first provable security analysis of the new FIDO2 protocols for a standard of passwordless user authentication. We identified several shortcomings and proposed stronger protocols. We hope our results will help clarify the security guarantees of the FIDO2 protocols and help the design and deployment of more secure and efficient passwordless user authentication protocols.

Acknowledgments. We thank the anonymous reviewers for their valuable comments. We thank Alexei Czeskis for help with FIDO2 details. A. Boldyreva and S. Chen were partially supported by the National Science Foundation under Grant No. 1946919. M. Barbosa was funded by National Funds through the Portuguese Foundation for Science and Technology in project PTDC/CCI-INF/31698/2017.

References

1. FIDO Alliance. Client to authenticator protocol (CTAP) – proposed standard (January 2019), `https://fidoalliance.org/specs/fido-v2.0-ps-20190130/fido-client-to-authenticator-protocol-v2.0-ps-20190130.html`
2. Google 2-step verification (2020), `https://www.google.com/landing/2step/`
3. Abdalla, M., Barbosa, M.: Perfect forward security of SPAKE2. Cryptology ePrint Archive, Report 2019/1194 (2019), `https://eprint.iacr.org/2019/1194`
4. Abdalla, M., Barbosa, M., Bradley, T., Jarecki, S., Katz, J., Xu, J.: Universally composable relaxed password authenticated key exchange. Cryptology ePrint Archive, Report 2020/320 (2020), `https://eprint.iacr.org/2020/320`
5. Abdalla, M., Pointcheval, D.: Simple password-based encrypted key exchange protocols. In: CT-RSA 2005. pp. 191–208. Springer (2005)
6. Barbosa, M., Boldyreva, A., Chen, S., Warinschi, B.: Provable security analysis of FIDO2. Cryptology ePrint Archive, Report 2020/756 (2020), `https://eprint.iacr.org/2020/756`
7. Bellare, M.: New proofs for nmac and hmac: Security without collision-resistance. In: CRYPTO 2006. pp. 602–619. Springer (2006)
8. Bellare, M., Canetti, R., Krawczyk, H.: Keying hash functions for message authentication. In: CRYPTO 1996. pp. 1–15. Springer (1996)
9. Bellare, M., Pointcheval, D., Rogaway, P.: Authenticated key exchange secure against dictionary attacks. In: EUROCRYPT 2000. pp. 139–155. Springer (2000)
10. Bellare, M., Rogaway, P.: Entity authentication and key distribution. In: CRYPTO 1993. pp. 232–249. Springer (1993)
11. Bellare, M., Rogaway, P.: The exact security of digital signatures-how to sign with RSA and Rabin. In: EUROCRYPT 1996. pp. 399–416. Springer (1996)
12. Boldyreva, A., Chen, S., Dupont, P.A., Pointcheval, D.: Human computing for handling strong corruptions in authenticated key exchange. In: CSF 2017. pp. 159–175. IEEE (2017)
13. Chen, S., Jero, S., Jagielski, M., Boldyreva, A., Nita-Rotaru, C.: Secure communication channel establishment: TLS 1.3 (over TCP fast open) versus QUIC. Journal of Cryptology **34**(3), 1–41 (2021)

14. Cohn-Gordon, K., Cremers, C., Dowling, B., Garratt, L., Stebila, D.: A formal security analysis of the Signal messaging protocol. Journal of Cryptology **33**(4), 1914–1983 (2020)
15. Consortium, W.W.W., et al.: Web authentication: An API for accessing public key credentials level 1 – W3C recommendation (March 2019), `https://www.w3.org/TR/webauthn`
16. Czeskis, A., Dietz, M., Kohno, T., Wallach, D., Balfanz, D.: Strengthening user authentication through opportunistic cryptographic identity assertions. In: CCS 2012. pp. 404–414 (2012)
17. Davis, G.: The past, present, and future of password security (May 2018)
18. Dolev, D., Yao, A.: On the security of public key protocols. IEEE Transactions on Information Theory **29**(2), 198–208 (1983)
19. Dowling, B., Fischlin, M., Günther, F., Stebila, D.: A cryptographic analysis of the TLS 1.3 handshake protocol. Cryptology ePrint Archive, Report 2020/1044 (2020), `https://eprint.iacr.org/2020/1044`
20. Dworkin, M.: Recommendation for block cipher modes of operation. methods and techniques. Tech. rep., National Inst of Standards and Technology Gaithersburg MD Computer security Div (2001)
21. FIDO: Specifications overview. `https://fidoalliance.org/specifications/`
22. Gott, A.: LastPass reveals 8 truths about passwords in the new Password Exposé (November 2017)
23. Guirat, I.B., Halpin, H.: Formal verification of the W3C web authentication protocol. In: 5th Annual Symposium and Bootcamp on Hot Topics in the Science of Security. p. 6. ACM (2018)
24. Haase, B., Labrique, B.: AuCPace: Efficient verifier-based PAKE protocol tailored for the IIoT. IACR Transactions on Cryptographic Hardware and Embedded Systems pp. 1–48 (2019)
25. Hu, K., Zhang, Z.: Security analysis of an attractive online authentication standard: FIDO UAF protocol. China Communications **13**(12), 189–198 (2016)
26. Igoe, K., McGrew, D., Salter, M.: Fundamental elliptic-curve Cryptography Algorithms. RFC 6090 (Feb 2011). https://doi.org/10.17487/RFC6090
27. Jacomme, C., Kremer, S.: An extensive formal analysis of multi-factor authentication protocols. In: CSF 2018. pp. 1–15. IEEE (2018)
28. Jager, T., Kakvi, S.A., May, A.: On the security of the PKCS# 1 v1. 5 signature scheme. In: CCS 2018. pp. 1195–1208 (2018)
29. Jarecki, S., Krawczyk, H., Shirvanian, M., Saxena, N.: Two-factor authentication with end-to-end password security. In: PKC 2018. pp. 431–461. Springer (2018)
30. Moriarty, K., Kaliski, B., Jonsson, J., Rusch, A.: PKCS #1: RSA Cryptography Specifications Version 2.2. RFC 8017 (Nov 2016). https://doi.org/10.17487/RFC8017
31. Nahorney, B.: Email threats 2017. Symantec. Internet Security Threat Report (2017)
32. Panos, C., Malliaros, S., Ntantogian, C., Panou, A., Xenakis, C.: A security evaluation of FIDO's UAF protocol in mobile and embedded devices. In: International Tyrrhenian Workshop on Digital Communication. pp. 127–142. Springer (2017)
33. Pereira, O., Rochet, F., Wiedling, C.: Formal analysis of the FIDO 1. x protocol. In: International Symposium on Foundations and Practice of Security. pp. 68–82. Springer (2017)
34. Verizon: 2017 data breach investigations report. `https://enterprise.verizon.com/resources/reports/2017_dbir.pdf` (2017)