On the design of Survivable Distributed Passwordless Authentication and Single Sign-On

Luca Ferretti*[†][‡], Federico Magnanini*[‡], Mauro Andreolini*, Mattia Trabucco*, Michele Colajanni[§]
*University of Modena and Reggio Emilia, Italy
[§]University of Bologna, Italy

***Abstract*—Single Sign-On (SSO) protocols allow an identity provider to authenticate users and report the outcome by issuing identity attestations. Recent attacks show that breaching the identity provider infrastructure enables adversaries to issue arbitrary identity attestations and impersonate users. Survivable SSO protocols limit the risks of similar intrusions, but they have only been defined for password-based authentication, inheriting their limitations against powerful attacks such as credential phishing. While phishing-resistant passwordless authentication protocols have been standardized, they are not designed to guarantee intrusion tolerance. We initiate the research for Survivable Passwordless SSO (SPS) and propose a modular approach which includes the novel definition of Survivable Passwordless Challenge-response (SPC) protocols for authentication as a subroutine of SSO. We give the first frameworks and game-based security definitions both for SPC and SPS which capture both novel attack classes, such as session injection attacks in a decentralized setting, and existing but not yet formalized attack classes, such as detection of cloned authenticators. The design of the models includes novel strategies to capture proactive security in survivable protocols within security definitions and to compose authentication and SSO through a modular approach. Our strategies and models may also be applied with minor modifications to non-survivable protocols, possibly providing a novel approach to assess the security of existing SSO protocols.**

I. Introduction

Single Sign-On (SSO) denotes delegated authentication protocols where an identity provider authenticates users that need to prove their identity to service providers. To this aim, the identity provider adopts a logical identity server that authenticates users and that reports the authentication outcome by issuing an identity attestation. Recent serious incidents (e.g., [11], [2]) show the fragility of this centralized authentication design: attackers that compromise the identity server can access user credentials or, even worse, issue arbitrary attestations to impersonate users. A recent line of research [3], [6], [29] proposes *survivable* SSO to tolerate intrusions in the identity provider infrastructure: they adopt multiple logical identity servers that collectively issue identity attestations, and users must demonstrate their identity to a service provider by obtaining a number of attestations that is greater than the assumed maximum number of malicious identity servers.

To date, only password-based survivable SSO protocols have been proposed [3], [6], [29], which address the challenge of protecting weak credentials against offline dictionary attacks and identity attestation forgery in presence of intrusions. However, they inherit typical vulnerabilities related to password usage, such as phishing attacks targeting user passwords. Passwordless authentication, which is based on dedicated physical devices or electronic hardware components, is a very effective defense against a successful phishing attack, yet no existing survivable SSO protocol provides the strong security guarantees offered by passwordless authentication.

We initiate the research for survivable passwordless SSO protocols, where the identity provider deploys multiple failure-independent identity servers and manages their cryptographic material, and users authenticate at these servers by using a specialized hardware authenticator through a user agent. The user agent collects an identity attestation from each identity server, and processes them to produce a collective attestation which is sent to the service provider, and which is accepted only if the number of valid identity attestations exceeds a threshold. We identify four main non-functional requirements. *Security*: we consider realistic threat models which have been defined for well-known standards, in particular FIDO2 [1] and OpenID Connect (OIDC) [24], and consider additional details related to survivability without weakening their original assumptions and guarantees. *Usability*: the distributed nature of survivable authentication must be transparent to users, and must not hinder usability with regard to the original non-survivable protocols. *Compatibility*: hardware authenticators are hard or impossible to upgrade, and their re-design involves high costs, thus our protocols consider compatibility with unmodified FIDO2 authenticators as an important requirement to allow the use of existing hardware security tokens available on the market. *Performance*: protocols must obtain response times that are accepted by established performance metrics for usable authentication systems. In this paper we focus on *security* and *usability* by proposing an operations framework and a related formal security model that assume a single interaction between a user and an authenticator for each authentication procedure, regardless of the number of servers involved. We discuss the main design choices in terms of *compatibility* within the framework and we sketch a candidate specification which is compliant with unmodified FIDO2 authenticators. We broadly discuss *performance* within our proposal, and leave a proper experimental evaluation as future work.

As no prior work exists on survivable passwordless authentication, we first identify the novel attack classes that emerge in this context, then we design the proposed protocol through a

[†]Corresponding author: luca.ferretti@unimore.it
[‡]These authors contributed equally.

modular approach. We give a formal definition of a *Survivable Passwordless Challenge-response (SPC) protocol* which considers the concurrent execution of multiple challenge-response protocols between the client and each identity server. Then, we use SPC as a sub-routine for designing a *Survivable Passwordless SSO (SPS) protocol* with security assumptions that are similar to those of OIDC with OAuth2 Proof Key for Code Exchange (PKCE) [18]. We formalize security properties that capture both novel attack classes and existing attack classes that have never been formalized by the literature. This formalization can be of independent interest, and some of them may be applied to standard (non-survivable and non-distributed) challenge-response and SSO protocols. We propose strategies for modeling security games in the context of proactive security and for designing SSO protocols through a modular approach. We estimate that our models are general enough to be compatible with many types of specifications with different trade-offs in terms of security and performance.

Section II discusses related work. Section III presents the system model. Section IV outlines design and challenges. Sections V and VI describe the SPC and SPS protocols and related security models. Section VII describes design sketches for candidate specifications. Section VIII discusses final remarks and future work. Appendix A shows visual examples of novel attacks. Appendices B and C include SPC and SPS formal correctness definitions.

II. Related work

As no prior work exists in this area, we are the first to consider passwordless usability requirements in survivable authentication. FIDO2 requires a simple user mediation during each authentication, and is considered usable by end-users [20], [14], [19]. We argue that survivable passwordless authentication protocols should preserve the same usability level of their non-survivable counterparts. In particular, survivable passwordless authentication protocols should require a single user mediation [27] to confirm the user's willingness to complete an authentication procedure, even if authentication involves the execution of a distributed protocol with multiple servers. This requirement rules out strawman implementations that sequentially execute non-distributed authentication protocols with multiple servers, as they would require a user mediation for each server. We consider security and usability as the most important requirements and we design SPS accordingly. Recent parallel research efforts investigated extensions of FIDO2 authentication to native decentralized architectures [25], [17], also highlighting the need for a single user interaction within a distributed passwordless authentication, but they do not perform a comprehensive analysis of the involved security threats, and do not consider proactive security nor SSO for non-decentralized applications.

We also consider the deployability of SPS, in the sense of Bonneau et al. framework for evaluating Web authentication schemes [9]. Their framework does not consider a novel deployability benefit of *compatibility with authenticators* that is relevant to our effort, as we preserve compatibility with existing authenticator implementations. Moreover, our protocol is also *browser-compatible* in the sense of Bonneau et al. framework, assuming client-side computation which can be executed with standard browser technologies (e.g., JavaScript), thus not requiring any modification to existing browser software, and limiting the need for more critical assumptions, such as the need for high-quality entropy for randomized procedures.

This work is closely related to a recent line of research that proposes intrusion-tolerant SSO protocols based on password authentication [3], [6], [29]. PASTA [3] considers an adversary that can corrupt up to a threshold of identity servers during the whole system lifetime. However, this protocol does not define the due procedures to recover compromised identity servers to a safe state. The ability to recover after a successful intrusion is essential in proactively secure and survivable systems [10], [16]. SPS models the ability to recover compromised identity servers to provide proactive security guarantees, such as *mobile* adversaries [23], [28] that perpetually try to compromise a threshold of identity servers in a given time unit and ensure persistent presence by moving laterally among identity servers. Even PESTO [6] considers a mobile adversary under an adaptive corruption model, which gives stronger security guarantees than PASTA. However, PESTO authentication does not complete if one identity server is unavailable. PROTECT [29] also obtains good trade-offs in terms of proactive security and completion guarantees. Although the authors do not formally specify their adversarial model, their proposal seems to consider a mobile adversary. Our models are generic enough to capture different types of protocol specifications, including those which can guarantee completion by tolerating unavailability of a threshold of identity servers, and different classes of approaches which assume the existence of trusted third party (e.g., centralized components of the identity provider) to execute particular operations (e.g., system setup, refresh of cryptographic keys), or completely decentralized approaches. Although all these proposals do not explicitly state assumptions for the considered communication model, they seem to consider the same synchronous and reliable point-to-point communication channels, and different assumptions in terms of decentralization: PROTECT has a decentralized setup and decentralized key refresh that requires synchronous communications among servers, PESTO has centralized setup and decentralized key refresh which requires no communications among servers, and PASTA has trusted centralized setup (and does not support key refresh). Certain traits of our design are already specific for certain system and security assumptions, such as considering a secure and reliable system during registration procedures, while others are general enough to capture multiple classes of protocol specifications with different assumptions, such as allowing both centralized and decentralized setup and/or key refresh.

The proposed security games for passwordless authentication are related to the seminal work of Barbosa et al. [5] for FIDO2, which adopts the Bellare-Rogaway model [7] to formally analyze the security of the WebAuthn [26] and

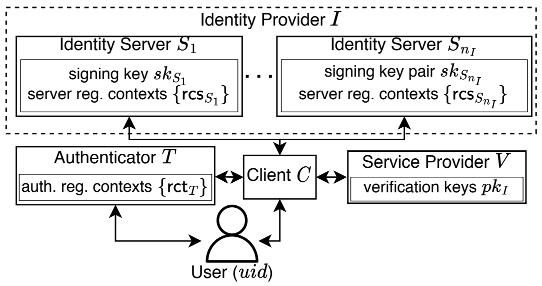

Identity Provider I
Identity Server S_1
signing key sk_{S_1}
server reg. contexts $\{\mathsf{rcs}_{S_1}\}$
Identity Server S_{n_I}
signing key pair $sk_{S_{n_I}}$
server reg. contexts $\{\mathsf{rcs}_{S_{n_I}}\}$
Authenticator T
auth. reg. contexts $\{\mathsf{rct}_T\}$
Client C
Service Provider V
verification keys pk_I
User (uid)

Fig. 1: System model

CTAP2 [4] protocols and their composition in FIDO2, and on the subsequent improvements by Bindel et al. [8], which considers more advanced threats but still considers non-survivable and non-decentralized settings. We adopt the formal model of [5] and use it to extend the security definitions of the original WebAuthn protocol to our distributed scenario. We also consider some of the modifications proposed by [8], such as assuming trusted registration without pre-distributed authenticator keys, which is more realistic for the most popular authentication scenarios. We do not capture other additional complexities considered by [8] and [5], such as downgrade attacks and a full composition with FIDO2-CTAP2 protocol, to focus on the novelties related to the distributed and survivable setting, and leave a comprehensive treatment as future work. We are the first to model authenticator cloning attacks within security games, partially building on analyses proposed by [22] in the context of techniques for detecting leakage of secret keys. While our models capture distributed and survivable settings, our results may also be of use to extend those proposed in [5] and [8].

Our formal security analyses for SSO relate to those for OIDC by [15], which are based on the symbolic Dolev-Yao adversarial model [13]. To the best of our knowledge, we are the first to define SSO security following the computational Bellare-Rogaway model [7], and a modular approach to combine authentication with SSO. While our analyses consider a decentralized setting, it is easy to adapt them for standard non-decentralized and/or non-survivable settings.

III. System model and notation

The protocols involve a set of parties $\mathcal{P}$ composed of the following finite, disjoint, non-empty subsets: users $\mathcal{U}$, authenticators $\mathcal{T}$, clients $\mathcal{C}$, identity servers $\mathcal{S}$, identity providers $\mathcal{I}$, service providers $\mathcal{V}$. Each identity provider $I \in \mathcal{I}$ exclusively *controls* a set of identity servers $\mathcal{S}_I \subseteq \mathcal{S}$ of cardinality $n_I = |\mathcal{S}_I|$. We identify servers and providers with globally unique labels denoted as id (that is, id_S, id_I, id_V). Each user is identified locally within an identity provider with label uid.

As shown in Figure 1, authenticator $T \in \mathcal{T}$ maintains a set of *authenticator registration contexts* $\{\mathsf{rct}_T\}$, and each rct_T includes secret and public key credentials denoted as $\mathsf{rct}_T.sk$ and $\mathsf{rct}_T.pk$. Each identity server S stores a set of *server registration contexts* $\{\mathsf{rcs}_S\}$, and each rcs_S includes the public key of an authenticator registration context $\mathsf{rct}_T.pk$ denoted as $\mathsf{rcs}_S.pk_T$, which is unique within $\{\mathsf{rcs}_S\}$. Authenticator and server registration contexts may also include other information defined by specifications, also including stateful data that changes throughout protocol execution. Each identity server also stores a *signing key* sk_S to authenticate user identity attestations. Service provider $V \in \mathcal{V}$ maintains a *verification key* pk_I for each identity provider I to verify authenticity of identity attestations released by any server $S \in \mathcal{S}_I$. Clients do not store persistent information.

We assume a synchronous system where honest parties do not crash, and we explicitly state when assuming reliable, authenticated and/or confidential channels. Identity provider I divides time into an ordered set of time periods $\Omega_I = \{\omega_I^\ell\}$, where each time period ω_I^ℓ is uniquely identified by a label $\ell \in \mathbb{N}_0$ denoting the ℓ-th time period. Each identity provider I assumes a maximum of $k_I \in \mathbb{N}_0$ malicious identity servers allowed within the same time period. While malicious identity servers may change throughout time periods, for clarity we assume that, for each I, k_I is constant among all time periods.

Further notation. We denote as $\langle \mathsf{rcs}_S \rangle_{S \in Q}$ the tuple of all server registration contexts associated with the same authenticator by some set of servers $Q \subseteq \mathcal{S}_I$. We denote as sid a session identifier and as Sid the set of all allowed values of sid. We use $\leftarrow$ and $\leftarrow\!\!\$$ to assign the output of deterministic and randomized algorithms, and $\perp$ is a special return value that denotes failure. We use $\{a|b\}$ to express mutually exclusive return values a or b. We denote as $\leftarrow\!\!\$\ \mathcal{D}$ discrete uniform sampling from some finite set $\mathcal{D}$. When $\mathcal{D} = \{0,1\}^\lambda$, we sample a bit string of length λ, where λ denotes the security level parameter (e.g, 128-bit). We denote as $[x, n]$ the set of integers $\{x, \dots, n\} : x \leq n$ and may use $[n]$ if $x = 1$, as $|Q|$ the cardinality of set Q, and as q_I the minimum cardinality of servers $Q \subseteq \mathcal{S}_I$ with which a client must execute registration or authentication procedures. We adopt the term *mapping* and denote it as $m : \mathcal{X} \rightarrow \mathcal{Y}$ in the sense of an abstract deterministic function which, given the same input $x \in \mathcal{X}$, always returns the same output $y \leftarrow m(x) : y \in \mathcal{Y}$. We may define a mapping as injective if input and outputs are bound to each other in a one-to-one relation. An injective mapping implies the existence of the inverse injective mapping m^{-1}.

IV. Overall design and challenges

We design Survivable Passwordless SSO (SPS) through a modular approach. First, we design a Survivable Passwordless Challenge-response (SPC) protocol that performs a distributed proof-of-possession among an authenticator and a set of identity servers. SPC differs from a typical challenge-response protocol because the prover (the pair client/authenticator) proves possession of a secret by authenticating a set of challenges, and the verifier (each identity server) checks that its own challenge belongs to the authenticated set. Second, we design SPS to perform SSO among a service provider and a set of identity servers by using SPC as a sub-protocol. With regard to a typical SSO protocol, SPS requires the service provider to accept only if a number of verifiers that accepted an SPC execution exceeds a threshold that depends on the maximum tolerable number of malicious identity servers. In the following, we overview the operations flow, we introduce the novel attack

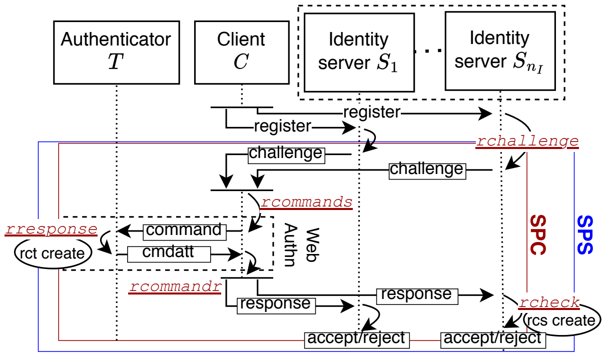


Fig. 2: SPS and SPC registration flow

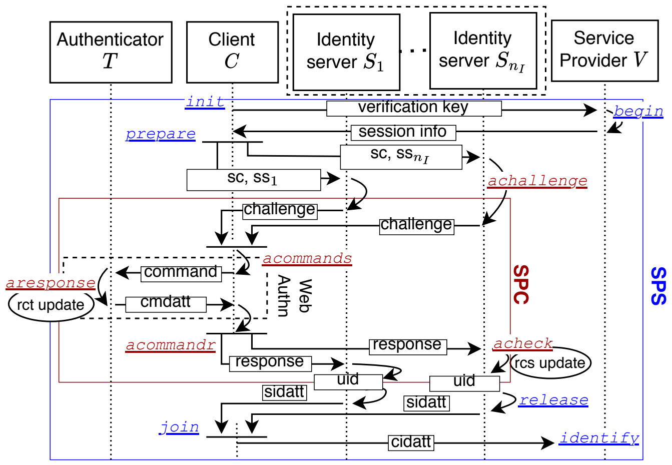


Fig. 3: SPS and SPC authentication flow

classes, and we overview the security experiments structure.

Overall flow. SPS consists of three protocols: *registration*, *authentication* and *refresh*. Registration lets a set of identity servers associate an authenticator to a known user identity. Authentication allows users to demonstrate their identity at a service provider by proving possession of their credentials via the authenticator to a set of identity servers. Refresh allows an identity provider to proactively rotate compromised signing keys and corrupted registration contexts on identity servers. Below we focus on registration and authentication to ease understanding of the protocol.

Figure 2 shows details of registration, where a client registers an authenticator at a set of identity servers. During registration there is no interaction with service providers, and SPS and SPC routines coincide. Each identity server executes rchallenge to compute a challenge. The client receives servers challenges and executes rcommands to aggregate them into a command data structure (*command*). The authenticator signs the aggregated challenges with rresponse to generate a command attestation (*cmdatt*) and creates a new authenticator registration context (rct). The client processes the result with rcommandr and sends a *client response* (*cr*) to all identity servers. Each server validates the aggregated challenges with rcheck and creates a new server registration context (rcs). As in [8], during registration we assume that client and servers have no pre-shared knowledge and communication channels are not under active attack, but the protocol can be adapted with trivial modifications to other settings involving pre-distributed public keys [5].

Figure 3 shows details of authentication, where a user identifies at a service provider by authenticating via SSO at a set of identity servers. Authentication starts with three SPS routines. The client executes init to generate and send a verification key to the service provider, that is information used by the end of the flow to prevent session injection attacks, *similarly* to OAuth2 PKCE [18]. The service provider executes begin to generate unique *session information* which characterizes the authentication flow. The client computes *shared context* information (*sc*) and distinct server-specific *subsession information* (*ss*) with the prepare routine, each forwarded to a different identity server. Client and servers use subsession information to initialize SPC to bind each execution of SPC to an execution SPS. Each identity server starts SPC by generating a challenge with achallenge. The client receives server challenges and executes acommands to aggregate them into *command*. The authenticator signs the aggregated challenges with aresponse to generate a command attestation (*cmdatt*) and updates the appropriate authenticator registration context. The client processes the result and sends a *client response* (*cr*) to the identity servers. Each server runs acheck to verify them and updates the registration context, finishing the execution of SPC and, in case of success, returning user identity uid and session identifier sid to SPS. In SPS, each server executes release to release a server identity attestation (*sidatt*) bound to uid and sid, thus uniquely binding the outcome of SPS to that of SPC. The client uses join to compute a client identity attestation (*cidatt*) from server identity attestations, and sends it to the service provider, which validates it with identify and accepts uid only if a quorum of identity servers released a valid identity attestation.

We observe that both *registration* and *authentication* are meant to be compatible with unmodified FIDO2-WebAuthn routines of authenticators (response routines within dashed boxes in Figures 2 and 3). The other routines may have been modified to support the decentralized setting (see Sections V-A and VI-A). In particular, note that FIDO2 only defines one command routine (that we name commands for *command send*) while responses from authenticators are forwarded. Instead, we allow clients to process responses with additional commandr routines.

Novel security threats. To define the security of SPS we extend existing *session authentication* and *session integrity* security properties of *non-survivable* SSO protocols [15] by considering a *mobile* adversary [23], [28] that can compromise a subset of identity servers within a time period. The *authentication* property refers to the inability of an adversary to impersonate a user at a service provider without compromising the identity provider (e.g., obtaining an identity attestation via a compromised communication channel). The *session integrity* property refers to the inability of an adversary to authenticate a user: i) at a service provider without the user's explicit consent (e.g., a replay attack), or ii) under an identity that is different from the identity proved to the identity provider (e.g., crosssite request forgery attacks).

We identify four novel security threats: *clone detection evasion*, *attestation mixing*, *shared session* and *forced authentication rejection*. We describe their impact on *authentication* and *session integrity* and hint at how to defend against them. Appendix A includes visual examples, and Sections V-B and VI-B describe security games to capture them.

Clone detection evasion consists in the undetectable creation of multiple instances of the same credential to perform stealth impersonation. In FIDO2, the identity server can detect authenticator cloning by relying on signature counters stored in registration contexts. We show that clone detection in a distributed environment is a harder challenge which cannot be exclusively enforced by identity servers alone. Since a subset of identity servers may be malicious, adversarial interleavings of authentication sessions may evade clone detection.

An *attestation mixing* attack refers to the ability of an adversary to mix and replay server attestations collected from sessions in different time periods, possibly issued by malicious identity servers, to exceed the allowed security threshold of malicious servers and break *authentication*.

A *shared session* attack refers to the ability of an adversary to inject an attestation in the victim session to authenticate the victim under the adversary's identity, thus breaking *session integrity*. This attack is possible if the adversary obtains any of two types of information: *subsession information* established between client and honest servers, or *command* data sent by the client to the authenticator. In both cases, the adversary may try to start concurrent SPC authentication subsessions to obtain honest server attestations related to the adversary identity which however are bound to the victim's session. The victim session can then be forced to use the adversary attestations via injection attacks. The attack is possible because we assume non-confidential communication channels with servers as in FIDO2. In this context, it is very important to observe that our authentication protocol does not require users to explicitly provide their identifier uid to clients, instead only requiring the user interaction with the authenticator. This approach is similar to FIDO2 *discoverable credentials* and is meant to improve authentication usability, however it introduces additional complexities in modeling and designing defenses to prevent session injection attacks, because the client does not have an easily-available trusted source for assessing the authenticity of the established uid. Since we consider SSO with non-authenticated communication channels and a distributed setting, modeling injection attacks and defending against them is quite more challenging than in FIDO2.

Forced authentication rejection is a denial-of-service attack where the adversary is able to trick honest servers into rejecting legitimate authentication requests. Intuitively, the adversary either tricks stateful logic of clone detection mechanisms to trigger false positives, or leverages vulnerabilities within the refresh procedure to let honest servers delete some registration contexts throughout time periods. The latter case, where the effect of the attack is permanent, could also be denoted as *forced account deletion*.

Security games and modeling novelties. We design five security games in total (three major security games and two variants). In Section V-B, we capture security of SPC authentication in Definition 1, which is based on Experiments 1 and 2 for capturing different classes of cloning adversaries attacks and related techniques. We also capture the capability of SPC authentication in protecting the confidentiality of session information for preventing *session injection attacks* in Definition 2 based on Experiment 3. In Section VI-B, we capture security of SPS in Definition 4, which is based on Experiment 4 and its variant Experiment 5, which on their turn extend SPC Experiments 1 and 2, respectively. We propose two novel modeling strategies: to design SPC security games in a survivable setting, where *refresh* operations cancel pending server sessions and may renew persistent storage, we extend the concept of *session oracles* first modeled in [7] and of partnering sessions modeled in [5] (see Section V-B); to adopt a modular approach and design SPC authentication as a subroutine of SPS, we express our survivable versions of *session authentication* and *integrity* first defined for standard OIDC in [15] by defining formal relations between SPC and SPS session identifiers which are very general and specification-agnostic, yet allow a rigorous definition of partnership within SSO (see Section VI-B).

V. SPC protocol

A. SPC operations framework

A Survivable Passwordless Challenge-response (SPC) protocol includes five sub-protocols: *authenticator setup*, *identity provider setup*, *register*, *authenticate* and *refresh*.

Authenticator setup $\mathsf{Spc.Tsetup}(T)$: executed once for each authenticator T, initializes its data structures, including initializing registration contexts to the empty set ($\{\mathsf{rct}_T\} = \emptyset$).

Identity provider setup $\{\perp \mid \langle \mathsf{id}_I, \{\mathsf{id}_S\}_{S\in\mathcal{S}_I}, \Omega_I, q_I\rangle\} \leftarrow \mathsf{Spc.Isetup}(I, \mathcal{S}_I, k_I)$: executed once for each identity provider $I \in \mathcal{I}$ and its servers $\mathcal{S}_I$ ($n_I = |\mathcal{S}_I|$), given security threshold k_I, returns identities $\mathsf{id}_I, \{\mathsf{id}_S\}_{S\in\mathcal{S}_I}$, time periods $\Omega_I = \{\omega_I^\ell\}_{\ell\in\mathbb{N}_0}$, and minimum cardinality q_I. We assume that outputs are public and may use them as implicit inputs to server routines if unambiguous. The protocol initializes data structures of each server $S \in \mathcal{S}_I$, including setting server registration contexts to the empty set $\{\mathsf{rcs}_S\} = \emptyset$. The protocol may reject with $\perp$ if k_I is invalid w.r.t. n_I.

Register ($\mathsf{Spc.Register}(C, T, Q, \mathsf{id}_I, uid)$): allows client C to register authenticator T for user identifier uid at a set of identity servers $Q \subseteq \mathcal{S}_I : |Q| \geq q_I$. It includes five routines:

- $rc_S \leftarrow\!\!\$\ \mathsf{Spc.rchallenge}(\{\mathsf{rcs}_S\})$: run by each $S \in Q$, generates challenge rc_S.
- $\langle M_r, pcr\rangle \leftarrow \mathsf{Spc.rcommands}(\mathsf{id}_I, uid, \langle rc_S\rangle_{S\in Q})$: run by C, creates registration command M_r and preliminary client response pcr for registering at I, given $\langle rc_S\rangle_{S\in Q}$.
- $\langle R_r, \{\mathsf{rct}_T\}'\rangle \leftarrow\!\!\$\ \mathsf{Spc.rresponse}(\{\mathsf{rct}_T\}, \mathsf{id}_I, M_r)$: run by T, registers a new credential for I given command M_r, returning registration response R_r and updated registration contexts $\{\mathsf{rct}_T\}'$.

- $\langle cr_S \rangle_{S \in Q} \leftarrow$ Spc.rcommandr(pcr, R_r, id$_{S \in Q}$): run by C, creates a client response cr_S for each $S \in Q$ by processing authenticator response R_r and preliminary response pcr.
- $\{\perp | \{\mathsf{rcs}_S\}'\} \leftarrow$ Spc.rcheck($\{\mathsf{rcs}_S\}, rc_S, uid, cr$): run by each $S \in Q$, validates cr w.r.t. rc_S, and returns updated registration contexts $\{\mathsf{rcs}_S\}'$ if accept, else $\perp$.

Authenticate ($\{\perp |uid\} \leftarrow$ Spc.Authenticate(C, T, Q, id$_I$, $sc, \langle ss_S \rangle_{S \in Q}$)): allows client C, bound to authenticator T, to authenticate at servers $Q \subseteq \mathcal{S}_I : |Q| \geq q_I$, given shared context sc and subsessions $\langle ss_S \rangle_{S \in Q}$. Return $\perp$ if reject, else identity uid[1]. It is composed of four routines:

- $ac_S \leftarrow\$$ Spc.achallenge($\{\mathsf{rcs}_S\}$): executed by each $S \in Q$, generates challenge ac_S.
- $\langle M_a, pcr \rangle \leftarrow$ Spc.acommands($\langle ac_S, ss_S \rangle_{S \in Q}, sc$): run by C, creates command M_a and preliminary client response pcr, given challenges $\langle ac_S \rangle_{S \in Q}$, shared context sc and subsessions $\langle ss_S \rangle_{S \in Q}$.
- $R_a, \{\mathsf{rct}_T\}' \leftarrow\$$ Spc.aresponse($\{\mathsf{rct}_T\}$, id$_I, M_a$): run by T, returns response R_a and updated registration contexts $\{\mathsf{rct}_T\}'$ given command M_a and identity id$_I$.
- $\langle cr_S \rangle_{S \in Q} \leftarrow$ Spc.acommandr($pcr, R_a, sc, \langle ss_S \rangle_{S \in Q}$): run by C, creates a client response cr_S for each $S \in Q$ by processing authenticator response R_a and preliminary response pcr.
- $\{\perp | \langle uid, \{\mathsf{rcs}_S\}' \rangle\} \leftarrow$ Spc.acheck($\{\mathsf{rcs}_S\}, ac_S, sc, ss_S$, cr): run by each $S \in Q$, validates responses cr of C w.r.t. ac_S, sc and ss_S. Returns uid and updated registration contexts $\{\mathsf{rcs}_S\}'$ if accept, else $\perp$.

Refresh (Spc.Refresh($I, \mathcal{S}_I, \omega_I^\ell$)): executed by identity provider I and its servers $\mathcal{S}_I$ at the end of time period ω_I^ℓ, terminates pending registration and authentication sessions, and generates new registration contexts as $\langle \{\mathsf{rcs}'_S\} \rangle_{S \in S_I} \leftarrow$ Spc.rcsrefresh(id$_I, k_I, \langle \{\mathsf{rcs}_S\} \rangle_{S \in S_I}$), where $\langle \{\mathsf{rcs}_S\} \rangle_{S \in S_I}$ and $\langle \{\mathsf{rcs}'_S\} \rangle_{S \in S_I}$ are registration contexts available during time periods ω_I^ℓ and $\omega_I^{\ell+1}$, respectively.

Correctness. Intuitively, correctness requires that each server $S \in \mathcal{S}_I$, even after multiple Spc.Refresh executions, always accepts an authentication that is consistent with a prior registration, that is, if the same server registered the user within the same time period, or a set of servers $Q \subseteq \mathcal{S}_I : |Q| \geq q_I$ registered the user in a previous time period. We leave the formal definition in Appendix B.

Compatibility with FIDO2. While apparently interfaces of Spc.rresponse and Spc.aresponse differ from those of FIDO2-CTAP2 due to the explicit adoption of identity provider identity id$_I$ among the inputs, in practice all FIDO2-CTAP2 specifications include the provider identity id$_I$ within the command data structure. More in general, we observe that any phishing-resistant authentication protocol would require sending such identity information to authenticators for scoping the protocol execution. We decide to let such information be explicit in the protocol interface for better modeling *identity-bound sessions* in security games, which becomes paramount when integrating the authentication with SSO for guaranteeing *session integrity* (Section VI-B). The outputs of the routines executed by authenticators (R_r and R_a) comply with those of FIDO2-CTAP2, thus not questioning compatibility issues. Obviously, for maintaining compliance, specifications have to assume that also the contents of these data structures are the same as those defined in FIDO2-CTAP2 (see Section VII). Instead, command routines modify the client operations with regard to FIDO2 clients, but can be implemented with standard browser technologies, such as JavaScript.

B. SPC security model

Session oracles. We extend session oracles modeled in [5] (which are based on [7]) to account for the decentralized and survivable setting. During execution there may be many instances of party $P \in \mathcal{T} \cup \mathcal{S}$. We denote each instance of a party P through a *session oracle* $\pi_P^{i,j}$, $(i,j) \in \mathbb{N}_0^2$, representing the j-th execution of the i-th parallel instance of party P. We say that an oracle *accepts* if it runs to completion without aborting or rejecting. $\pi_P^{i,j=0}$ is party P registration session for a new instance i, and $\pi_P^{i,j\geq 1}$ is party P j^{th} authentication session following the *accepted* registration session $\pi_P^{i,j=0}$. Thus, each party P is associated with a set of session oracles $\{\pi_P^{i,j}\}_{\langle i,j \rangle \in \mathbb{N}_0^2}$ that for ease of notation we simply denote as $\{\pi_P^{i,j}\}$ except when used otherwise. Persistent storage of a party is shared among all its associated oracles, such as authenticator registration contexts $\{\mathsf{rct}_T\}$ which are available to all oracles $\{\pi_T^{i,j}\}$, and may also be denoted as $\pi_T.\{\mathsf{rct}\}$. Due to survivability, the same assumption does not hold for server oracles, because at the end of each time period $\omega_I^\ell \in \Omega_I$ all server sessions are terminated and persistent storage is renewed. Thus, with regard to [5] we introduce novel notations, and denote as $\pi_{S,\ell}^{p,u} : S \in \mathcal{S}_I$ a server session that is active within time period $\omega_I^\ell \in \Omega_I$. During time period $\omega_I^\ell \in \Omega_I$, all and only oracles $\{\pi_{S,\ell}^{p,u}\}_{\langle p,u \rangle \in \mathbb{N}_0}$ access the same set of registration contexts $\pi_{S,\ell}.\{\mathsf{rcs}\}$. Each oracle also establishes a persistent state which is available by all and only oracles belonging to the same parallel instance. We denote as $\pi_{S,\ell}^p.\mathsf{rcs}$ ($\pi_T^i.\mathsf{rct}$) a registration context generated by $\pi_{S,\ell}^{p,u=0}$ ($\pi_T^{i,j=0}$) which can be accessed by any following instance $\pi_{S,\ell}^{p,u>0}$ ($\pi_T^{i,j>0}$). We let u be local within each time period, and any oracle of the form $\pi_{S,\ell}^{p,u=0}$ represents an initialization procedure, performed through a registration or a Refresh procedure. For ease of notation and without loss of generality, we use the same index to denote server oracles which registered the same users. More formally, for any pair of servers $S_1, S_2 \in \mathcal{S}_I$, time periods $\omega_I^{\ell_1}, \omega_I^{\ell_2} \in \Omega_I$, and indexes $p_1, p_2 \in \mathbb{N}_0$, if there exist $\pi_{S_1,\ell_1}^{p_1}.uid$ and $\pi_{S_2,\ell_2}^{p_2}.uid$, then $\pi_{S_1,\ell_1}^{p_1}.uid = \pi_{S_2,\ell_2}^{p_2}.uid \iff p_1 = p_2$.

Session identifiers. A protocol specification must define a *session identifier* sid$_{SPC}$ that allows to uniquely identify

[1]Authentication may be easily modified to accept an explicit user identifier uid' as input by including uid' within sc when calling Authenticate, and by validating $uid = uid'$ afterwards. As already hinted in Section IV, guaranteeing security in the proposed design, where the client does not access the user identifier, is strictly more difficult because the client cannot perform any validation on the user established by the servers.

Security experiment 1 **($\textsc{k-n}$-SPC with Unrestricted Cloning adversaries).** *The security experiment is run between a challenger and adversary $\mathcal{A}$.*
As setup, the challenger defines sets $\mathcal{I} = \mathcal{S} = \mathcal{T} = \emptyset$, identity provider I_t, and identity servers $\mathcal{S}_{I_t} : |\mathcal{S}_{I_t}| = \textsc{n}$; adds I_t to $\mathcal{I}$ and each $S \in \mathcal{S}_{I_t}$ to $\mathcal{S}$; initializes identity servers and generates public parameters as $\left\langle \mathsf{id}_{I_t}, \{\mathsf{id}_S\}_{S \in \mathcal{S}_{I_t}}, \Omega_{I_t}, q_{I_t} \right\rangle \leftarrow \mathsf{Spc.Isetup}(I_t, \mathcal{S}_{I_t}, k_{I_t})$, where $k_{I_t} = \textsc{k}$. The challenger also defines the set Unfresh $= \emptyset$ *of* unfresh *credentials.*
At any time, the adversary can define an authenticator T, or a corrupt *identity provider $I \notin \mathcal{I}$ and* corrupt *servers $\mathcal{S}_I \notin \mathcal{S}$, arbitrarily defining security threshold k_I and public parameters $\langle \mathsf{id}_I, \{\mathsf{id}_S\}_{S \in \mathcal{S}_I}, \Omega_I, q_I \rangle$. Each time, $\mathcal{A}$ sends everything to the challenger, which adds T to $\mathcal{T}$ (initializing it with $\mathsf{Spc.Tsetup}(T)$), I to $\mathcal{I}$, and all $S \in \mathcal{S}_I$ in $\mathcal{S}$.*
At any time, $\mathcal{A}$ may also modify storage and internal state of corrupt *entities to arbitrary values.*
Then, the security experiment proceeds in a series of rounds $\ell = 0, 1, \dots$ modeling time periods $\omega_{I_t}^{\ell} \in \Omega_{I_t}, \ell \in \mathbb{N}_0$. $\mathcal{A}$ chooses when to terminate a round and proceed to the next. At the beginning of each round, $\mathcal{A}$ chooses a set of servers $\Sigma_\ell \subset \mathcal{S}_{I_t} : |\Sigma_\ell| \leq \textsc{k}$ which are corrupt *during round ℓ, and gives Σ_ℓ to the challenger. Thus, during round ℓ, servers in $(\mathcal{S}_{I_t} \setminus \Sigma_\ell)$ are* honest, *while those in $(\mathcal{S} \setminus (\mathcal{S}_{I_t} \setminus \Sigma_\ell))$ are* corrupt. *At the end of each round, the challenger runs $\langle \{\mathsf{rcs}'_\mathsf{S}\} \rangle_{S \in \mathcal{S}_{I_t}} \leftarrow \mathsf{Spc.rcsrefresh}(\mathsf{id}_I, k_I, \langle \pi_{S,\ell}.\{\mathsf{rcs}\} \rangle_{S \in S_{I_t}})$, advances to the next round ($\ell = \ell + 1$), and sets $\pi_{S,\ell}.\{\mathsf{rcs}\} = \{\mathsf{rcs}'_\mathsf{S}\}, \forall S \in S_{I_t}$.*
$\mathcal{A}$ interacts with $\mathcal{T}$ and $\mathcal{S}$ via the following queries, constrained by the following running conditions: server oracles associated with round ℓ can be queried only in round ℓ, the game aborts *if an undefined oracle variable is accessed, each query can only run once for each session oracle, oracles that have* aborted, accepted, *or* rejected *can no longer be queried, and an authenticator oracle can either run* Response, CResponse, *or* Clone.

$\mathsf{Challenge}(\pi_{S,\ell}^{p,u}, sc, ss)$

```
1 : if $u = 0$ then abort
2 : $ac \leftarrow\!\!\$\ \mathsf{Spc.achallenge}(\pi_{S,\ell}.\{\mathsf{rcs}\})$
3 : $\pi_{S,\ell}^{p,u}.\langle ac, sc, ss \rangle := \langle ac, sc, ss \rangle$
4 : return $ac$
```

$\mathsf{Response}(\pi_T^{i,j}, M)$

```
1 : if $j = 0$ then abort
2 : $\langle R, \{\mathsf{rct}\}' \rangle \leftarrow \mathsf{Spc.aresponse}(\pi_T.\{\mathsf{rct}\}, \pi_T^i.\mathsf{id}_I, M)$
3 : $\pi_T.\{\mathsf{rct}\} := \{\mathsf{rct}\}'$
4 : return $R$
```

$\mathsf{CResponse}(\pi_T^{i,j}, \langle ac_S, ss_S \rangle_S, sc)$

```
1 : if $j = 0$ then abort
2 : $\langle M, pcr \rangle \leftarrow \mathsf{Spc.acommands}(\langle ac_S, ss_S \rangle_S, sc)$
3 : $R \leftarrow \mathsf{Response}(\pi_T^{i,j}, M)$
4 : $\pi_T^{i,j}.\langle ss_S \rangle_S := \langle ss_S \rangle_S$
5 : $\pi_T^{i,j}.sc := sc$
6 : return $M, pcr, R$
```

$\mathsf{Complete}(\pi_{S,\ell}^{p,u}, cr)$

```
1 : if $u = 0$ then abort
2 : $\{\perp \mid \langle uid, \{\mathsf{rcs}\}' \rangle\} \leftarrow \mathsf{Spc.acheck}(\pi_{S,\ell}.\{\mathsf{rcs}\}, \pi_{S,\ell}^{p,u}.ac, \pi_{S,\ell}^{p,u}.sc, \pi_{S,\ell}^{p,u}.ss, cr)$
3 : if $\perp$ then $\pi_{S,\ell}^{p,u}.\mathsf{res} := \mathsf{reject}$; return
4 : if $uid \neq \pi_{S,\ell}^{p}.uid$ then abort
5 : $\pi_{S,\ell}.\{\mathsf{rcs}\} := \{\mathsf{rcs}\}'$; $\pi_{S,\ell}^{p,u}.\mathsf{res} := \mathsf{accept}$
6 : return $\{\mathsf{rcs}\}'$
```

$\mathsf{Register}(\pi_T^{i,j}, \{\pi_{S,\ell}^{p,u_S}\}_{S \in Q}, uid, \mathsf{id}_I)$:

```
1 : if $Q \not\subseteq \mathcal{S}_I$ or $|Q| < q_I$ or $j > 0$ or $p > 0$ or
        $(\exists i' : \pi_T^{i'}.\mathsf{id}_I = \mathsf{id}_I)$ or $(\exists p' : \pi_{S,\ell}^{p'}.uid = uid)$ or
        $(\exists S \in Q : u_S > 0)$ then abort
2 : foreach $S \in Q$ do $rc_S \leftarrow \mathsf{Spc.rchallenge}(\pi_{S,\ell}.\{\mathsf{rcs}\})$
3 : $\langle M_r, pcr \rangle \leftarrow \mathsf{Spc.rcommands}(\mathsf{id}_I, uid, \langle rc_S \rangle_{S \in Q})$
4 : $\langle R, \{\mathsf{rct}\}' \rangle \leftarrow \mathsf{Spc.rresponse}(\pi_T.\{\mathsf{rct}\}, \mathsf{id}_I, M_r)$
5 : $\langle cr_S \rangle_{S \in Q} \leftarrow \mathsf{Spc.rcommandr}(pcr, R, \mathsf{id}_{S \in Q})$
6 : $\pi_T.\{\mathsf{rct}\} := \{\mathsf{rct}\}'$
7 : $\pi_T^i.\langle \mathsf{id}_I, uid \rangle := \langle \mathsf{id}_I, uid \rangle$
8 : foreach $S \in Q$ do :
9 :     $\{\perp \mid \{\mathsf{rcs}_S\}'\} \leftarrow \mathsf{Spc.rcheck}(\pi_{S,\ell}.\{\mathsf{rcs}\}, rc_S, uid, cr_S)$
10 :    / Result is never $\perp$ due to if conditions and SPC correctness.
11 :    $\pi_{S,\ell}.\{\mathsf{rcs}\} := \{\mathsf{rcs}_S\}'$
12 :    $\pi_{S,\ell}^{p}.uid := uid$
13 : endforeach
14 : return $\langle M_r, \langle cr_S \rangle_{S \in Q}, R, \langle \{\mathsf{rcs}_S\}' \rangle_{S \in Q} \rangle$
```

$\mathsf{Clone}(\pi_T^{i,j}, \{\pi_{S,\ell}^{p,u_S}\}_{S \in Q})$

```
1 : if $\exists I \in \mathcal{I} : (\mathsf{id}_I = \pi_T^i.\mathsf{id}_I$ and $(Q \not\subseteq \mathcal{S}_I$ or $|Q| < q_I))$ or
        $j = 0$ or $(\exists S \in Q : u_S = 0)$ or $\pi_{S,\ell}^{p}.uid \neq \pi_T^i.uid$ or
        $|\{\pi_{S,\ell}^{p,u_S}.sc\}_{S \in Q}| > 1$ then abort
2 : $\mathsf{Unfresh.add}(\langle \mathsf{id}_I, \pi_T^i.uid \rangle)$
3 : $sc := \pi_{S,\ell}^{p,u_S}.sc$, for any $S \in Q$
4 : $\langle M, pcr, R \rangle \leftarrow \mathsf{CResponse}(\pi_T^{i,j}, \langle \pi_{S,\ell}^{p,u_S}.ac, \pi_{S,\ell}^{p,u_S}.ss \rangle_{S \in Q}, sc)$
5 : $\langle cr_S \rangle_{S \in Q} \leftarrow \mathsf{Spc.acommandr}(pcr, R, sc, \pi_T^{i,j}.\langle ss_S \rangle_{S \in Q})$
6 : foreach $S \in Q$ do :
7 :     $\{\perp \mid \langle uid, \{\mathsf{rcs}_S\}' \rangle\} \leftarrow \mathsf{Spc.acheck}(\pi_{S,\ell}.\{\mathsf{rcs}\}, \pi_{S,\ell}^{p,u}.ac, \pi_{S,\ell}^{p,u}.sc, \pi_{S,\ell}^{p,u}.ss, cr_S)$
8 :     / Result is never $\perp$ and $uid = \pi_{S,\ell}^{p}.uid$ always holds due to if conditions.
9 :     $\pi_{S,\ell}.\{\mathsf{rcs}\} := \{\mathsf{rcs}_S\}'$
10 : endforeach
11 : return $\langle \pi_T^i.\mathsf{rct}, M_a, \langle cr_S \rangle_{S \in Q}, R_a, \langle \{\mathsf{rcs}_S\}' \rangle_{S \in Q} \rangle$
```

the session between an authenticator oracle and a set of identity server oracles. To uniquely identify a session, the sid_{SPC} can be defined as a function of the protocol transcript including messages of each participant that are unique among all possible protocol executions with an overwhelming probability. While we leave sid_{SPC} implicit in protocol framework because it does not impact correctness, we may access them from session oracles as $\pi_T^{i,j}.\mathsf{sid}_{SPC}$ and $\pi_{S,\ell}^{p,u}.\mathsf{sid}_{SPC}$ when formalizing security. Note that for authenticators, sid_{SPC} is defined when they run Spc.rresponse ($j = 0$) or Spc.aresponse ($j > 0$), and for servers when they run Spc.rcheck ($u = 0$) or Spc.acheck ($u > 0$). We denote session oracles without a session identifier as $\pi_T^{i,j}.\mathsf{sid}_{SPC} = \perp$ and $\pi_{S,\ell}^{p,u}.\mathsf{sid}_{SPC} = \perp$.

SPC Partnering. We extend the partnering notion of [5]. In our context, partnering captures the intuitive idea that an *honest* server session accepts for an authenticator session if it previously registered either at that same server within the same time period, or at a number of honest servers exceeding a security threshold during previous time periods. More formally, we say that server oracle $\pi_{S,\ell}^{p,u}$ and authenticator oracle $\pi_T^{i,j}$ are partnered if both oracles accept and: i) S is honest; ii) if $u = 0 \wedge j = 0$, they share the same sid_{SPC}; iii) if $u > 0 \wedge j > 0$, they share the same sid_{SPC}, and either iii.a) $\pi_{S,\ell}^{p,0}$ and $\pi_T^{i,0}$ are partnered, or iii.b) there exists $\omega_I^{\ell' < \ell} \in \Omega_I$ such that $\pi_T^{i,0}$ is partnered with a set of honest server oracles $\{\pi_{S,\ell'}^{p,0}\}_{S \in \mathcal{S}_I}$ of cardinality at least equal to $(k_I + 1)$. We say that a server oracle is *uniquely partnered* with an authenticator oracle to denote that the server oracle is partnered with exactly one authenticator oracle. In our distributed scenario, partnership is a surjective relation, as multiple server oracles may be

Security experiment 2 (K-N-SPC with Restricted Cloning adversaries). *The experiment is as Experiment 1, except that:*

- Response *and* CResponse *also* abort *if* $\langle \pi_T^i.\mathsf{id}_I, \pi_T^i.uid \rangle$ *is* unfresh *(i.e.,* $\langle \pi_T^i.\mathsf{id}_I, \pi_T^i.uid \rangle \in \mathsf{Unfresh}$*);*
- $\mathcal{A}$ *can also query the following* FullAuth *routine.*

```
FullAuth($\pi_T^{i,j}, \{\pi_{S,\ell}^{p,u_S}\}_{S \in Q}, sc, \langle ss_S \rangle_{S \in Q}$)
1:  if ($\exists I \in \mathcal{I} : (\mathsf{id}_I = \pi_T^i.\mathsf{id}_I$ and $(Q \not\subseteq \mathcal{S}_I$ or $|Q| < q_I))$) or
        $j = 0$ or ($\exists S \in Q : u_S = 0$) then abort
2:  $\langle M, pcr, R \rangle \leftarrow \mathsf{CResponse}(\pi_T^i, \langle \pi_{S,\ell}^{p,u_S}.ac, ss_S \rangle_{S \in Q}, sc)$
3:  $\langle cr_S \rangle_{S \in Q} \leftarrow \mathsf{Spc.acommandr}(pcr, R, sc, \langle ss_S \rangle_{S \in Q})$
4:  foreach $S \in Q$ do :
5:      $\{\perp \mid \langle uid, \{\mathsf{rcs}_S\}' \rangle\} \leftarrow \mathsf{Spc.acheck}(\pi_{S,\ell}.\{rcs\},$
            $\pi_{S,\ell}^{p,u_S}.ac, \pi_{S,\ell}^{p,u_S}.sc, \pi_{S,\ell}^{p,u_S}.ss, cr_S)$
6:      if $\perp$ then $\pi_{S,\ell}^{p,u_S}.\mathsf{res} := \mathsf{reject}$
7:      elseif $uid \neq \pi_{S,\ell}^{p,u_S}.uid$ then abort
8:      else $\pi_{S,\ell}^{p,u_S}.\mathsf{res} := \mathsf{accept}$; $\pi_{S,\ell}.\{\mathsf{rcs}\} := \{\mathsf{rcs}_S\}'$
9:  endforeach
10: $\pi_T.\{\mathsf{rct}\} := \{\mathsf{rct}\}'$
11: return $\langle M, pcr, R, \langle cr_S \rangle_{S \in Q} \rangle$
```

uniquely partnered with the same authenticator oracle.

Trust assumptions. We design two security experiments. Experiment 1 captures security of SPC authentication against adversaries which can freely interact with session oracles using unfresh credentials, namely *unrestricted cloning adversaries*. Experiment 2 slightly modifies Experiment 1 to detect unfresh credentials only against weaker adversaries, namely *restricted cloning adversaries*, which cannot act maliciously against session oracles using unfresh credentials to circumvent detection techniques. We comply with FIDO2 `None` attestation mode[2] and, during registration, we assume mutually authenticated, reliable, and non-private communication channels between authenticators and clients, and between clients and identity servers. Authenticators are tamper-proof against modifications: the adversary can only read the authenticator internal state and cannot modify it. More in detail, we consider fine-grained credentials extraction: the adversary could be able to clone any amount of credentials stored within the authenticator, but the extraction of a credential scoped for a certain identity provider does not affect detection logic associated with a different provider. We denote each cloned credential as *unfresh*, as typical for secret information that has been accessed by the adversary. During authentication, Experiment 1 assumes that channels between clients and identity servers, are not authenticated, not reliable, and not private, regardless of cloned or not cloned credentials. Adversaries may either directly access authenticators or clients connected to an authenticator, and in the latter case channels between them are mutually authenticated. Experiment 2 modifies Experiment 1 and assumes that, in case of cloned credentials, channels are server-side authenticated and reliable during authentication.

[2]`None` attestation mode is the most popular registration mode of FIDO2 which does not require distribution of authenticator public keys at servers. Threat modeling of this mode is based on WebAuthn official security statements [26, Section 13.4.4] and has already been considered in [8]. We could also comply with `Basic` mode, adopted in [5], with minor modifications.

Security of SPC authentication. Formally expressed in Definition 1, security holds if: i) *all* fresh and honest server oracles which accept must be partnered with exactly one authenticator oracle to prevent the possibility of impersonation of an authenticator session by an external adversary; ii) there exists a fresh server *partnered* with an authenticator, then it must be associated with the same user identifier ($\pi_{S,\ell_S}^{ps}.uid = \pi_T^i.uid$) to avoid impersonation from different authenticators or different credentials residing within the same authenticator (note that Complete aborts if $uid \neq \pi_{S,\ell}^p.uid$ to avoid trivial winning), and same session and subsession information ($\pi_{S,\ell_S}^{p,u}.ss = \pi_T^i.ss_S, \pi_{S,\ell_S}^{p,u}.sc = \pi_T^i.sc$) to ensure that they execute within the same authentication session; iii) there exist fresh servers which are *uniquely partnered* with an authenticator oracle (thus the server is also honest and accepts), then they must be associated with the same round ($|\{\ell_S\}_{S \in Q}| = 1$) to avoid *mixing of attestations*, and all of them must be associated either with a registration or with an authentication procedure (if $j = 0$ then all $u_S = 0$, if $j > 0$ then all $u_S > 0$); iv) $(k_{I_t} + 1)$ or more unfresh honest server oracles accept and are uniquely partnered, then the protocol did not detect a cloned credential and is vulnerable against *clone detection evasion* attacks; v) responses produced by a fresh authenticator oracle which correctly registered within the same or a previous round should always be accepted by an honest server oracle, otherwise the protocol is vulnerable against *forced authentication rejection* attacks.

Discussion and comparison. As in [8], we distinguish queries for authentication and registration due to different security assumptions. For authentication, the adversary can call Challenge and Complete at each server to allow adaptive behavior based on challenges and votes respectively, with no confidentiality or authenticity. Moreover, it can call Response, which models direct access to the authenticator as in [5] and [8], or CResponse, which is new and models access to a client connected to an authenticator and is meant to keep track of session information used to build the command data structure with acommands. For registration, Register forces registration at a threshold of servers to model that communication channels are reliable and not under active attack, as in [8], and that all honest servers are always available. Future work may drop this constraint to model weaker assumptions during registration, such as assuming honest servers and/or clients with *crash* faults. Intuitively, such a setting may require deploying some distributed agreement protocol (e.g., two-phase commit or state replication) to prevent that a set of servers of cardinality less than q_{I_t} installs a registration context, which would make the protocol behavior unpredictable. In particular, the outcome of a refresh execution would depend on how malicious nodes behave, possibly violating partnership throughout multiple time periods.

Clone is completely new and captures cloning of authenticators: its complexity is meant to model the concept of *observation of misuses* formally defined by [22], and prevents trivial winnings of the adversary by forcing authentication with cloned credentials at a threshold of servers. Note that authen-

Definition 1 (Secure K-N-SPC with Clone Detection). *Let* Π *be a Survivable Passwordless Challenge-response (SPC) protocol with security threshold* $\textsc{k}$ *and* $\textsc{n}$ *identity servers. We say that a server oracle* $\pi_{S,\ell}^{p,u}$ *is* fresh *if tuple* $\left\langle \pi_{S,\ell}^{p}.\mathsf{id}_I, \pi_{S,\ell}^{p}.uid \right\rangle$ *is not marked as* unfresh, *else it is* unfresh. *We define* $\mathsf{Adv}_{\Pi,\mathcal{A}}^{\mathrm{spc-du}}(\lambda, \textsc{k}, \textsc{n})$ *and* $\mathsf{Adv}_{\Pi,\mathcal{A}}^{\mathrm{spc-dr}}(\lambda, \textsc{k}, \textsc{n})$ *as the probabilities that* $\mathcal{A}$ *satisfies any of the following conditions respectively in Experiment 1 and 2:*

i. *an* honest *and* fresh *server oracle* accepts *and is not uniquely partnered with an authenticator oracle;*
ii. *a* fresh *server oracle* $\pi_{S,\ell}^{p,u}$ *is uniquely partnered with* $\pi_T^{i,j}$ *and ii.a)* $\pi_{S,\ell}^{p}.uid \neq \pi_T^{i}.uid$ *or ii.b)* $\pi_{S,\ell}^{p,u}.sc \neq \pi_T^{i,j}.sc \vee \pi_{S,\ell}^{p,u}.ss \neq \pi_T^{i,j}.ss_S$*;*
iii. *a set of* fresh *server oracles* $\{\pi_{S,\ell_S}^{p_S,u_S}\}_{S \in Q \subseteq \mathcal{S}_{I_t}} : |Q| \geq 1$ *uniquely partnered with authenticator oracle* $\pi_T^{i,j}$ *and iii.a)* $|\{\ell_S\}_{S \in Q}| \neq 1$ *or iii.b)* $(j = 0 \wedge \exists u_S > 0) \vee (j > 0 \wedge \exists u_S = 0)$*;*
iv. *a set of* honest *and* unfresh *server oracles* $\{\pi_{S,\ell}^{p,u_S}\}_{S \in Q \subseteq \mathcal{S}_{I_t} \setminus \Sigma_\ell} : |Q| \geq (k_{I_t} + 1)$ accept *and are uniquely partnered with an authenticator oracle;*
v. *an* honest *and* fresh *server oracle* $\pi_{S,\ell}^{p,u>0}, S \in \mathcal{S}_{I_t} \setminus \Sigma_\ell$ rejects *although there exists* $\pi_T^{i,j>0}$ *with the same* sid_{SPC}*, and either* $\pi_{S,\ell}^{p,u'=0}$ *is uniquely partnered with* $\pi_T^{i,j'=0}$*, or there exists a set* $\{\pi_{S,\ell'}^{p,u'=0}\}_{S \in Q' \subseteq \mathcal{S}_{I_t} \setminus \Sigma_{\ell'}} : |Q'| \geq (k_{I_t} + 1), \ell' < \ell$ *uniquely partnered with* $\pi_T^{i,j'=0}$.

Π *is a secure SPC and allows Detection of cloned credentials against Unrestricted cloning adversaries (DU), or against Restricted cloning adversaries (DR), respectively if* $\mathsf{Adv}_{\Pi,\mathcal{A}}^{\mathrm{spc-du}}(\lambda, \textsc{k}, \textsc{n})$ *or* $\mathsf{Adv}_{\Pi,\mathcal{A}}^{\mathrm{spc-dr}}(\lambda, \textsc{k}, \textsc{n})$ *are negligible in* λ*.*

tication is performed *without using the authenticator oracle itself*, so that afterwards the same oracle is not able to Complete if the protocol specification detects cloning. While [5] does not model violation of authenticator credentials, the security experiment proposed in [8] includes a Corrupt query which leaks all secret keys stored within an authenticator, marking the authenticator as unfresh, following a consolidated approach in the literature [7] where the adversary can read secrets to *learn* something about the protocol, but cannot use them to (trivially) win the game. We are the first to capture mechanisms to detect leaked secrets within a security game, that is, to allow adversaries to win with unfresh credentials[3].

We observe that clone detection techniques where states maintained by authenticators and servers are independent of the content of exchanged messages, namely those based on *observation of contradictions* [22], cannot guarantee security against Experiment 1, because the adversary could trivially *re-align* a cloned authenticator state after a Clone by executing Response as many times as needed. Nonetheless, this class of techniques is highly relevant due to advantages in terms of usability and/or reduced costs, as demonstrated by the adoption of counters within FIDO2 specifications. Experiment 2 limits interactions of adversaries with unfresh authenticator oracles by only allowing full authentication procedures at server-side authenticated servers, thus preventing an adversary from manipulating or dropping packets in transit, or from interacting with malicious identity providers to re-align states. The experiment still allows unrestricted interactions with fresh authenticator oracles, including those possibly associated with an authenticator for which cloned authenticator oracles exist.

Confidential subsession information. We propose Experiment 3 and Definition 2 to capture SPC protocols which keep subsession information confidential, which is inspired by the IND-DCPA game used for modeling confidentiality of deterministic encryption schemes. In our case, $\mathcal{A}$ is given access to Spc.acommands run by clients as an oracle to observe any arbitrary output for adaptively chosen inputs. Even after observing (a polynomial number of) commands $\langle M_t \rangle_t$ and preliminary client responses $\langle pcr_t \rangle_t$ for arbitrarily and adaptively chosen inputs, $\mathcal{A}$ must not be able to distinguish whether the challenger used its input $ss_{S'}$ or uniformly sampled ss' for running Spc.acommands, modeling that Spc.acommands protects confidentiality of ss_S, for any provider I, server S, and any other provided input. We need to constrain $\mathcal{A}$ in choosing a subsession for which it did not observe any outputs before, because we model acommands as deterministic[4]. Contrary to Experiment 1 and 2, we let $\mathcal{A}$ choose parameters $\textsc{k}$ and $\textsc{n}$ arbitrarily to capture that clients are stateless and may be unaware of the trusted public parameters associated with identity provider I, and thus an adversary may trick them in using fake parameters to its own advantage. Thus, if a specification is secure in the sense of Definition 2, confidentiality of subsession information is guaranteed for all values of $\textsc{k}$ and $\textsc{n}$, for any id_I, regardless of the authentic values $\textsc{k}_I$ and $\textsc{n}_I$.

Definition 2 (IND-DCSA: Indistinguishability of subsession information against Distinct Chosen Subsession Attacks). *For a SPC protocol* Π*, we define* $\mathsf{Adv}_{\Pi,\mathcal{A}}^{\mathrm{ind-dcsa}}(\lambda)$ *as the probability that* $\mathcal{A}$*, running Experiment 3, guesses* b*. We say that* Π *is IND-DCSA if* $\mathsf{Adv}_{\Pi,\mathcal{A}}^{\mathrm{ind-dcsa}}(\lambda) \leq 2^{-\lambda} + \mathsf{negl}(\lambda)$.

Authenticity of client responses. We propose Definition 3 to capture authenticity guarantees of a client response cr_S when subsession information ss_S can act as a cryptographic key. In this case, we do not propose an additional experiment because we can adopt the well-known EUF-CMA game typically associated with message authentication codes. Additional intuitions are in Section VII.

[3]We observe that Clone acts as Corrupt for authenticator credentials which are not associated with the honest identity provider, as running a full authentication procedure against a completely malicious identity provider would not limit nor add advantages to the adversary, thus we model strictly stronger adversaries against authenticators than [8].

[4]As for deterministic encryption, which cannot be secure in the well-known IND-CPA game, no deterministic acommands specification could prevent trivial winnings by $\mathcal{A}$ picking $\langle ss_S \rangle_{S \in Q'} \in \left\langle \langle ss_S^t \rangle_{S \in Q^t} \right\rangle_t$. Note that while acommands may be easily re-designed as a probabilistic algorithm in theory, such an approach would put additional real-world assumptions on the capabilities available to clients, and diverge from those assumed by FIDO2 (in particular, availability of a good source of entropy). Since uniqueness of subsession information is also a requirement of SPS, assuming distinct subsession information is realistic (see Section VI.).

Security experiment 3 (DCSA: Distinct Chosen Subsession Attacks). *The experiment is run between a challenger and adversary $\mathcal{A}$. Let $\mathcal{A}$ choose identity provider I, servers $\mathbb{S}_I$ (for any $\mathrm{N} = |\mathbb{S}_I|$), and security threshold k_I, and run* $\langle \mathsf{id}_I, \{\mathsf{id}_S\}_{S \in \mathbb{S}_I}, \Omega_I, q_I \rangle \leftarrow \mathsf{Spc.Isetup}(I, \mathbb{S}_I, k_I)$.
Then, for security level λ, $\mathcal{A}$ iteratively runs $\langle M^t, pcr^t \rangle \leftarrow \mathsf{acommands}(\langle ac_S^t, ss_S^t \rangle_{S \in Q^t}, sc^t)$ *for $t \in [1, \ldots]$, choosing inputs $\langle ac_S^t, ss_S^t \rangle_{S \in Q^t}$, sc^t arbitrarily and adaptively at each iteration t.*
To conclude, $\mathcal{A}$ arbitrarily chooses a set $Q' \subseteq \mathbb{S}_I$, a server $S' \in Q'$, shared context sc and subsession $\langle ss_S \rangle_{S \in Q'} \neq \langle ss_S^t \rangle_{S \in Q^t} \wedge$ $\mathsf{len}(ss_S) = \lambda, \forall t \leq poly(\lambda), \forall S \in Q'$*, and sends them all to the challenger, which samples $ss' \leftarrow\!\!\$ \{0,1\}^\lambda$, flips a coin $b \leftarrow\!\!\$ \{0,1\}$ and, only if $b = 1$, updates $ss_{S'} \leftarrow ss'$ within $\langle ss_S \rangle_{S \in Q'}$. The challenger executes* $\langle M', pcr' \rangle \leftarrow \mathsf{acommands}(\langle ac_S \rangle_{S \in Q'}, sc, \langle ss_S \rangle_{S \in Q'})$, *and gives M' and pcr' to $\mathcal{A}$, which wins if it guesses b.*

Definition 3 (EUF-CSA: Existential Unforgeability under Chosen Subsession Attacks). *For a SPC protocol Π, we say that Π is EUF-CSA if $\langle cr_S \rangle_S \leftarrow \Pi.\mathsf{acommandr}(pcr, R_a, sc, \langle ss_S \rangle_S)$ is a secure EUF-CMA scheme, where ss_S acts as secret key and $\langle pcr, R_a, sc \rangle$ acts as message.*

VI. Survivable Passwordless SSO (SPS) protocol

A. SPS operations framework

SPS consists of four subprotocols: *identity provider setup*, *register*, *authenticate*, *refresh*.

Identity provider setup. Run $\langle \mathsf{id}_I, \{\mathsf{id}_S\}_{S \in \mathbb{S}_I}, \Omega_I, q_I \rangle \leftarrow \mathsf{Spc.Isetup}(I, \mathbb{S}_I, k_I)$ and use returned parameters as implicit inputs as in SPC. Then, let I and its servers $\mathbb{S}_I$ run key generation $\langle sk_I^0 = \{sk_S^0\}_{S \in \mathbb{S}_I}, pk_I^0 \rangle \leftarrow\!\!\$ \mathsf{Sps.ikgen}(I, \mathbb{S}_I, k_I)$ to return secret and public key material $\langle sk_I^0, pk_I^0 \rangle$ associated with time period $\omega_I^{\ell=0} \in \Omega_I$. Secret key material sk_I^ℓ includes secret keys of servers $\{sk_S^\ell\}_{S \in \mathbb{S}_I}$. Depending on specifications, public key material pk_I^ℓ may be shared by all servers (e.g., in case of threshold cryptography [12]) or include server-specific keys ($pk_I^\ell = \{pk_S^\ell\}_{S \in \mathbb{S}_I}$).

Register ($\mathsf{Sps.Register}(C, T, Q, \mathsf{id}_I, uid)$): coincides with $\mathsf{Spc.Register}$.

Authenticate ($uid \leftarrow \mathsf{Sps.Authenticate}(V, C, T, Q, \mathsf{id}_I)$): executed among service provider V, identity servers $Q \subseteq \mathbb{S}_I : |Q| \geq q_I$ for identity provider identity id_I, authenticator T and client C, allows V to authenticate C for identity uid. Authenticate includes:

- $\langle cs, cv \rangle \leftarrow\!\!\$ \mathsf{Sps.init}(\mathsf{id}_V)$: run by C, generates client secret and verification data cs and cv for initiating the authentication flow with service provider id_V.
- $av \leftarrow\!\!\$ \mathsf{Sps.begin}(\mathsf{id}_I, \{\mathsf{id}_S\}_{S \in Q}, cv)$: run by V for starting an authentication flow toward $\{\mathsf{id}_S\}_{S \in Q}$ given client verification data cv. Generate session information av.
- $\left\langle sc, \langle ss_S \rangle_{S \in Q} \right\rangle \leftarrow \mathsf{Sps.prepare}(\mathsf{id}_V, av, \mathsf{id}_I, \{\mathsf{id}_S\}_{S \in Q}, cs)$: run by C, generates subsession information $\langle ss_S \rangle_{S \in Q}$ and shared context sc, given client secret data cs, session information av and identities id_V, id_I and $\{\mathsf{id}_S\}_{S \in Q}$.
- $\{\perp | uid\} \leftarrow \mathsf{Spc.Authenticate}(C, T, Q, \mathsf{id}_I, sc, \langle ss_S \rangle_{S \in Q})$: run SPC authentication among C, T and Q, given id_I, sc and $\langle ss_S \rangle_{S \in Q}$. Each $S \in Q$ obtains uid if it accepts.
- $vs_S \leftarrow \mathsf{Sps.release}(sk_S^\ell, uid, sc, ss_S)$: run by each $S \in Q$ for which $\mathsf{Spc.Authenticate}$ accepts, creates server attestation vs_S given uid returned by $\mathsf{Spc.Authenticate}$, and the same sc, ss_S used for $\mathsf{Spc.Authenticate}$.
- $\{\perp, vc\} \leftarrow \mathsf{Sps.join}(cs, av, sc, \langle ss_S, vs_S \rangle_{S \in Q})$: run by C, generates client attestation vc, given cs, sc and $\langle ss_S \rangle_{S \in Q}$ already known by C, and $\langle vs_S \rangle_{S \in Q}$ received by servers. May reject invalid input attestations with $\perp$.
- $\{\perp, uid\} \leftarrow \mathsf{Sps.identify}(pk_I^\ell, cv, av, vc)$: run by V, concludes the authentication flow started for identity provider id_I (associated with public key pk_I^ℓ known by V) and outputs identity uid if accepts, else $\perp$, given cv and av known by V, and vc given by C.

Refresh ($\mathsf{Sps.Refresh}(I, \mathbb{S}_I, \omega_I^\ell)$): consists of two routines: first, run $\mathsf{Spc.Refresh}$ to terminate pending sessions and renew server registration contexts; second, renew key material with $\langle sk_I^{\ell+1}, pk_I^{\ell+1} \rangle \leftarrow\!\!\$ \mathsf{Sps.ikrefresh}(\langle sk_I^\ell, pk_I^\ell \rangle, k_I)$, which is executed by I and its servers $\mathbb{S}_I$ at the end of time period $\omega_I^\ell \in \Omega_I$, returns key material $\langle sk_I^{\ell+1}, pk_I^{\ell+1} \rangle$ for the next time period $\omega_I^{\ell+1} \in \Omega_I$.

Correctness. Intuitively, correctness requires that a service provider accepts a user identity that has been voted by enough identity servers of an identity provider with regard to the same authentication flow previously initiated by the client and within the same time period. Formal definition is in Appendix C.

Variants. Although our notation binds each public key to a time period, specifications may allow re-use of the same key material throughout multiple (or all) time periods (i.e. $pk_I = pk_I^\ell, \forall \omega_I^\ell \in \Omega_I$). Although provider I may seem to be forcefully involved as a centralized party during setup and/or key refresh, specifications may still opt for decentralized approaches. As anticipated in Section IV, we decide to not provide uid as an input to $\mathsf{Sps.Authenticate}$ so that a user is authenticated by simply interacting with the authenticator, and without any additional interaction with the client (e.g., entering the username in a Web form). Our model offers strictly stronger security, and it is trivial to capture explicit provisioning of uid to C by introducing uid within the inputs of $\mathsf{Sps.prepare}$, such that sc can be bound to uid, and by letting each S verify the binding with regard to uid produced by $\mathsf{Spc.Authenticate}$ within $\mathsf{Sps.release}$ (and possibly also V during $\mathsf{Sps.identify}$).

B. SPS security model

Security notions. As anticipated in Section IV, we adapt notions of *authentication* and *session integrity for authentication* of [15] to our scenario, and we denote them as *SPS authentication* and *SPS session integrity*. Intuitively:

- *SPS authentication* captures that, at the end of an authentication flow, an honest service provider V accepts authentication of client C only if a set of honest identity servers $Q \subseteq \mathbb{S}_I : |Q| \geq (k_I + 1)$ accepted an execution of $\mathsf{Spc.Authenticate}$ for the same authentication flow (authentication *freshness*), and if C is the same client which started the authentication flow;

- *SPS session integrity* captures that, at the end of an authentication flow, an honest V accepts a user identity uid of identity provider I (declared by V as the expected *issuer* of the attestations at the beginning of the flow) for client C bound to authenticator T, if and only if V accepted I as *issuer* at the beginning of the flow and V has been set as the *audience* of the attestations, and uid has been previously bound to T at I during registration.

By definition, SPS authentication relies on SPC authentication, plus the capability for each identity server to communicate its vote to the identity provider, and of binding votes to authentication flows to guarantee their freshness and thus prevent their reuse. To approach the challenge, we formally relate SPS authentication to the concept of session identifiers, define novel notions of partnership, and adopt these concepts within the proposed security games.

Instead, SPS session integrity (for authentication) involves the capability of correctly communicating and/or validating the context of the authentication flow through metadata, which is quite simple to guarantee within the distributed setting by adopting strategies that are already known for centralized settings, such as explicitly storing them authenticated within attestations or by adopting specialized cryptographic keys[5]. A more challenging task is protecting against session injection attacks, especially because we consider passwordless authentication without the explicit input (or validation) of the username by the client (see $\mathsf{Sps.Authenticate}$ in the previous Section VI-A). To this aim, we propose a security game which captures both novel shared session attacks which emerge in the considered distributed setting (see Section IV) and attacks that are known in non-decentralized settings, but which have not been captured by the literature.

Session and subsession identifiers. SPS specifications must define a session identifier sid_{SPS} to uniquely identify an authentication flow among a service provider V and a client C. Moreover, to capture the execution of a subsession of SPS shared by C and by each server S participating in the flow, specifications must also define a subsession identifier ssid_{SPS} for each S (we may denote it as ssid^S_{SPS}).

To guarantee SPS authentication, sid_{SPS}, ssid_{SPS} and sid_{SPC} must be bound to each other and to the entities involved in the authentication flow. We model such bindings through the following *mappings* (see Section III for notation):

- injective mapping $m_A : \mathsf{Sid}_{SPS} \rightarrow \mathsf{Sid}_{SPC}$, modeling that SPS and SPC authentications must be uniquely bound to each other in a one-to-one relation;
- non-injective mapping $m_I : \mathsf{Sid}_{SPS} \rightarrow \mathcal{V} \times \mathcal{I} \times \mathcal{S}^q$, where $\mathcal{S}^q$ denotes some set of servers $Q \subseteq \mathcal{S}_I : I \in \mathcal{I}, |Q| = q \geq q_I$ chosen by $V \in \mathcal{V}$ to run the authentication flow, modeling that each sid_{SPS} is bound to specific identity and service providers, and to a subset of identity servers;
- injective mapping $m_F : \mathsf{Sid}_{SPS} \times \mathcal{S} \rightarrow \mathsf{Ssid}_{SPS}$, where allowed input servers $\mathcal{S}$ for some sid_{SPS} is the output of $m_I(\mathsf{sid}_{SPS})$, modeling *freshness* of a server SPS subsession with regard to only one authentication flow, and uniqueness of ssid_{SPS} among all servers and authentication flows.

If SPC is secure, then each server S contributes at most once in the same authentication process, as expressed by condition i of Definition 1, and thus there exists a *non-injective* mapping $m_F : \mathsf{Sid}_{SPC} \times \mathbb{S} \rightarrow \mathsf{Ssid}_{SPS}$ (but does not imply existence of m_F^{-1}). Thus, a SPS specification must show existence of mappings m_F^{-1} and m_I, and injective mapping m_A. Note that mappings could not be (efficiently) computable and/or verifiable by some or all parties, depending on design choices.

SPS session oracles and partnership. The protocol is executed by session oracles of party $P \in \mathcal{T} \cup \mathbb{S} \cup \mathcal{V} \cup \mathcal{C}$. Oracles $\pi^{i,j}_T$ and $\pi^{p,u}_{S,\ell}$ maintain the same semantics of SPC (Section V-B), while π^g_C and π^h_V denote the g-th and h-th ongoing or completed sessions of client $C \in \mathcal{C}$ and service provider $V \in \mathcal{V}$. We say that oracle π^h_V which accepts in $\mathsf{Sps.identify}$ is partnered with: a client oracle π^g_C, if they share the same sid_{SPS}, such that $m_I(\mathsf{sid}_{SPS}) = \langle V, I, Q\rangle : Q \subseteq \mathcal{S}_I$; a set of identity server oracles $\{\pi^{p,u_S>0}_{S,\ell}\}_{S \in Q}$, if each oracle $\pi^{p,u_S}_{S,\ell}$ accepts with $\pi^{p,u}_{S,\ell}.\mathsf{ssid}_{SPS} = m_F(\pi^g_V.\mathsf{sid}_{SPS}, S)$. Although authenticators $T \in \mathcal{T}$ are not directly involved within SPS, we may also say that π^g_V is partnered with some $\pi^{i,j>0}_T$ if $\pi^{i,j}_T.\mathsf{sid}_{SPC} = m_A(\pi^g_V.\mathsf{sid}_{SPS})$[6].

SPS trust assumptions and security guarantees. As we design SPS as an extension of SPC, we define SPS Experiment 4 and its (trivial) variant Experiment 5 which extend SPC Experiments 1 and 2, respectively for unrestricted and restricted cloning adversaries. We also propose Definition 4 for secure SPS authentication which extends SPC winning conditions of Definition 1. In both experiments, we consider the same trust assumptions adopted for SPC for channels between authenticators and identity servers, and for tamper-resistant authenticators (see Section V-B), with the exception of assuming that messages exchanged between legitimate clients and their bound authenticators are mutually authenticated. Moreover, we assume that messages sent from clients to service providers and identity servers are confidential[7]. Any other communication channel is not authenticated nor confidential. Client sessions may be corrupted and information

[5]Nowadays, SSO is often accompanied with delegated authorization mechanisms, where the additional *session integrity for authorization* should also be guaranteed [15]. We estimate that the decentralized setting does not pose additional challenges, and SPS can be extended for handling authorizations by including an additional *scope* attribute within identity server attestations, which should be consistent among all honest servers in Q for being accepted by the service provider. As authorization is not the focus of this paper, we leave in-depth analyses as future work.

[6]If π^g_V is partnered with $\pi^{p,u}_{S,\ell}$ and if SPC is secure, then $\pi^{p,u_S}_{S,\ell}$ is partnered with $\pi^{i,j}_T$ and thus for transitivity $m_A(\pi^g_V.\mathsf{sid}_{SPS}) = \pi^{i,j}_T.\mathsf{sid}_{SPC}$.

[7]Although assuming confidential channels only for outgoing messages sent by clients may seem unusual, it complies with assumptions of OAuth2 PKCE for native applications in mobile environments [18].

Security experiment 4 (K-N-SPS with Unrestricted Cloning adversaries). *Setup proceeds as in Experiment 1, but after the execution of* Spc.Isetup*, the challenger also runs* $\left\langle sk^0_{I_t} = \{sk^0_S\}_{S\in\mathcal{S}_{I_t}}, pk^0_{I_t}\right\rangle \leftarrow\!\!\$\ \mathsf{Sps.ikgen}(I_t, \mathcal{S}_{I_t}, k_{I_t})$*, and also defines the set* $\mathsf{Bnd} := \emptyset$ *of bindings between authenticator and client sessions.*
At any time, $\mathcal{A}$ can define new entities as in Experiment 1, and for each corrupt *I it can also arbitrarily define pk^ℓ_I and modify it at will throughout the protocol execution. $\mathcal{A}$ can also define clients $\mathcal{C}$ and service providers $\mathcal{V}$ (with arbitrary unique identities $\{\mathsf{id}_V\}_{V\in\mathcal{V}}$) and give them to the challenger, and also* corrupt *them for accessing or manipulating their internal state.*
The experiment proceeds in rounds as in Experiment 1, but at the end of each round $\omega^\ell_{I_t}$, after Spc.Refresh*, the challenger also runs* $\left\langle sk^{\ell+1}_{I_t}, pk^{\ell+1}_{I_t}\right\rangle \leftarrow\!\!\$\ \mathsf{Sps.ikrefresh}(\left\langle sk^\ell_{I_t}, pk^\ell_{I_t}\right\rangle, k_{I_t})$.
$\mathcal{A}$ can interact with parties in $\mathcal{T}\cup\mathcal{I}\cup\mathcal{C}\cup\mathcal{V}$ via the same queries of Experiment 1 and the following additional queries. Running conditions are the same of Experiment 1, except that the same π^g_C can run StartAuth *multiple times, the same $\pi^{i,j}_T$ can exclusively run* Prove, Response *or* CResponse*, the same $\pi^{p,u}_{S,\ell}$ can exclusively run* StartAuth *or* Challenge.

```
Init($\pi^g_C, \mathsf{id}_V$)
1: $\langle cs, cv\rangle \leftarrow\!\!\$\ \mathsf{Sps.init}(\mathsf{id}_V)$
2: $\pi^g_C.\langle cv, cs, \mathsf{id_V}\rangle := \langle cv, cs, \mathsf{id}_V\rangle$
3: return $cv$
```

```
Begin($\pi^h_V, \mathsf{id}_I, \{\mathsf{id}_S\}_S, cv$)
1: if $(\exists \mathsf{id}_S : S \notin \mathcal{S}_I)$ or $|\{\mathsf{id}_S\}_S| < q_I$ then
     abort
2: $av \leftarrow \mathsf{Sps.begin}(\mathsf{id}_I, \{\mathsf{id}_S\}_S, cv)$
3: $\pi^h_V.\langle \mathsf{id}_I, \{\mathsf{id}_S\}_S, av, cv\rangle := \langle \mathsf{id}_I, \{\mathsf{id}_S\}_S, av, cv\rangle$
4: return $av$
```

```
Prepare($\pi^g_C, av, \mathsf{id}_I, \{\mathsf{id}_S\}_S$)
1: $\{\perp, \langle sc, \langle ss_S\rangle_S\rangle\} \leftarrow \mathsf{Sps.prepare}(\pi^g_C.\mathsf{id}_V,$
        $av, \mathsf{id}_I, \{\mathsf{id}_S\}_S, \pi^g_C.cs)$
2: if $\perp$ then abort
3: $\pi^g_C.\langle av, \mathsf{id}_I, \{\mathsf{id}_S\}_S, sc, \langle ss_S\rangle_S\rangle :=$
        $\langle av, \mathsf{id}_I, \{\mathsf{id}_S\}_S, sc, \langle ss_S\rangle_S\rangle$
4: return $sc$
```

```
StartAuth($\pi^{p,u}_{S,\ell}, \pi^g_C, sc$)
1: if $u = 0$ or $\mathsf{id}_S \notin \pi^g_C.\{\mathsf{id}_{S'}\}_{S'}$ then
2:   abort
3: $ac \leftarrow \mathsf{Spc.achallenge}(\pi^{p,u}_{S,\ell}.\{\mathsf{rcs}\})$
4: $\pi^{p,u}_{S,\ell}.\langle ac, sc, ss\rangle := \langle ac, sc, \pi^g_C.ss_S\rangle$
5: return $ac$
```

```
Prove($\pi^{i,j}_T, \pi^g_C, \langle ac_S\rangle_S$)
1: if $|\langle ac_S\rangle_S| \neq |\langle \pi^g_C.ss_S\rangle_S|$ or $\pi^i_T.\mathsf{id}_I \neq \pi^g_C.\mathsf{id}_I$ then abort
2: $\langle M, pcr, R\rangle \leftarrow \mathsf{CResponse}(\pi^{i,j}_T, \langle ac_S, \pi^g_C.ss_S\rangle_S, \pi^g_C.sc)$
3: $\langle cr_S\rangle_S \leftarrow \mathsf{Spc.acommandr}(pcr, R, \pi^g_C.sc, \pi^g_C.\langle ss_S\rangle_S)$
4: $\mathsf{Bnd.add}(\langle \pi^{i,j}_T, \pi^g_C\rangle)$
5: return $\langle M, pcr, R, \langle cr_S\rangle_S\rangle$
```

```
Release($\pi^{p,u}_{S,\ell}$)
1: if $\pi^{p,u}_{S,\ell}.\mathsf{res} \neq \mathsf{accept}$ then
2:   abort
3: $vs \leftarrow \mathsf{Sps.release}(sk^\ell_S,$
        $\pi^p_{S,\ell}.uid, \pi^{p,u}_{S,\ell}.sc, \pi^{p,u}_{S,\ell}.ss)$
4: return $vs$
```

```
Join($\pi^g_C, \langle vs_S\rangle_S$)
1: if $|\langle vs_S\rangle_S| \neq |\langle \pi^g_C.ss_{S'}\rangle_{S'}|$ then abort
2: $\{\perp, vc\} \leftarrow \mathsf{Sps.join}(\pi^g_C.cs, \pi^g_C.av, \pi^g_C.sc, \langle \pi^g_C.ss_S, vs_S\rangle_S)$
3: if $\perp$ then $\pi^g_C.\mathsf{res} := \mathsf{reject}$
4: else $\pi^g_C.\langle \mathsf{res}, vc\rangle := \langle \mathsf{accept}, vc\rangle$
5: return
```

```
Identify($\pi^h_V, \pi^g_C$)
1: let $pk^\ell_I : \mathsf{id}_I = \pi^h_V.\mathsf{id}_I$
2: $\{\perp, uid\} \leftarrow \mathsf{Sps.identify}(pk^\ell_I, \pi^h_V.cv, \pi^h_V.av, \pi^g_C.vc)$
3: if $\perp$ then $\pi^h_V.\mathsf{res} := \mathsf{reject}$
4: else $\pi^h_V.\langle \mathsf{res}, uid\rangle := \langle \mathsf{accept}, uid\rangle$
5: return
```

```
Inject($\pi^g_C, vc$)
1: $\pi^g_C.vc := vc$
2: return
```

Security experiment 5 (K-N-SPS with Restricted Cloning adversaries). *The experiment runs as Experiment 4 and has the same winning conditions, except that it extends Experiment 2 instead of Experiment 1, and* Prove aborts *also if* $\langle \pi^i_T.\mathsf{id}_I, \pi^i_T.uid\rangle$ *is* unfresh.

maintained by clients and service providers may leak, but trivial wins with this information is not allowed.

Intuitions and remarks. Init, Begin, Prepare model calls to homonymous SPS routines, initializing client and service provider sessions for an authentication flow. Despite data sent by clients is assumed to be confidential, we let Init disclose vs to $\mathcal{A}$ to model potential compromise of service providers without introducing an additional method. Prove models confidential and authenticated interaction between a client and an authenticator, and the same authenticator session cannot execute both Prove and Response because both involve the execution of Spc.aresponse. In a correct execution, $\mathcal{A}$ calls them in the same order, then it calls routines of Experiment 1 (Experiment 2), which Experiment 4 (Experiment 5) extends, to perform a correct execution of SPC authentication. Finally, it can call Join and Identify to conclude SSO. Join allows adversaries to provide arbitrary attestations to clients, which however could validate them in Spc.join and possibly abort. Identify lets a service provider session receive client secret data cs confidentially, and identity attestation vc possibly based on one or multiple injected vs_S. To attack the protocol, since we are not considering authenticated channels, $\mathcal{A}$ can provide arbitrary data to all routines. In Identify, where client send confidential data, $\mathcal{A}$ can send arbitrary data to service providers by compromising a client with Compromise, thus allowing the possibility of manipulating its internal state. For Definition 4 to hold, we consider honest identity provider (I_t) and client sessions to avoid trivial winnings with corrupt entities, and require that: *vi*) each identity provider accepting must be uniquely partnered with a client session, otherwise the authentication flow could be hijacked or the client could be re-using the same identity attestation for multiple authentication flows (violation of *SPS authentication*); *vii*) each service provider accepting must be uniquely partnered with at least $k_{I_t} + 1$ (that is, $\kappa + 1$) honest identity server authentication sessions associated with the same credential and time period, otherwise the protocol could be vulnerable to different types of attestation mixing; *viii*) identity provider and authenticator partnered and bound to the same client are also uniquely partnered, otherwise the protocol can be affected by injection attacks. Note that such unique partnership implies that identity provider accepts identity uid stored within the bound authenticator session.

VII. Design sketch of candidate specifications

We outline candidate specifications for SPC and SPS authentication and refresh, and sketch security proofs. Due to

Definition 4 (**Secure K-N-SPS**). *Let Π be a Survivable Passwordless SSO (SPS) protocol with security threshold* K *and* N *identity servers. We define* $\mathsf{Adv}^{\mathrm{sps-du}}_{\Pi,\mathcal{A}}(\lambda,\mathrm{K},\mathrm{N})$ *and* $\mathsf{Adv}^{\mathrm{sps-dr}}_{\Pi,\mathcal{A}}(\lambda,\mathrm{K},\mathrm{N})$ *as the probabilities that $\mathcal{A}$ satisfies the conditions expressed in Definition 1 for Experiment 1 and 2, respectively, or if* $\mathsf{Identify}(\pi^h_V,\pi^g_C)$ accepts *in round ℓ in Experiments 4 and 5, respectively, if* $\pi^h_V.\mathsf{id}_I=\mathsf{id}_{I_t}$*,* π^g_C *is not* corrupt*, and any of the following conditions:*

vi. π^h_V *is not uniquely partnered with* π^g_C*;*
vii. π^h_V *is not uniquely partnered with some set* $\mathcal{S}_Q=\{\pi^{p,u_S>0}_{S,\ell}\}_{S\in Q\subseteq\mathcal{S}_{I_t}\setminus\Sigma_\ell}:|Q|\geq(k_{I_t}+1)$*;*
viii. π^g_C *is not* bound *to any* $\pi^{i,j}_T$ *or, let* π^g_C *be* bound *to* $\pi^{i,j}_T$*,* π^h_V *is not uniquely partnered with* $\pi^{i,j}_T$*.*

Π is a secure SPS and allows Detection of cloned credentials against Unrestricted cloning adversaries (DU), or against Restricted cloning adversaries (DR), respectively if $\mathsf{Adv}^{\mathrm{sps-du}}_{\Pi,\mathcal{A}}(\lambda,\mathrm{K},\mathrm{N})$ *or* $\mathsf{Adv}^{\mathrm{sps-dr}}_{\Pi,\mathcal{A}}(\lambda,\mathrm{K},\mathrm{N})$ *are negligible in λ.*

space limitations, full details are omitted in the current version, and many variants or alternatives may be designed as well.

Figure 4 shows a specification for SPC authentication for unmodified FIDO2 authenticators, that is, where the specification of aresponse is that of FIDO2, and acommands returns a FIDO2-compliant command data structure M_a[8]. The specification uses a commitment scheme which supports efficient membership verification as core primitive for performing the collective challenge-response, such as Merkle commitments as in [25] (see Section II). We denote as H a collision-resistant hash function, MAC an EUF-CMA message authentication code, $\mathsf{Sig}:=\langle\mathsf{Sig.sign},\mathsf{Sig.verify}\rangle$ an EUF-CMA digital signature scheme, $\mathsf{Com}:=\langle\mathsf{Com.commit},\mathsf{Com.verify}\rangle$ a commitment scheme[9]. We assume that T and S have previously established $\mathsf{rct}=\langle\mathsf{id}_I,cid,sk_T,cnt_T\rangle$ and $\mathsf{rcs}=\langle cid,pk_T,uid,cnt_S,\langle h_T,\sigma_T\rangle\rangle$, either during a registration or a refresh, where: $\langle sk_T,pk_T\rangle$ is a unique key pair; cid is the *credential identifier* generated by T for id_I at registration [1]; cnt_T is a counter incremented at each execution of aresponse and cnt_S is the last counter value seen by S; h and σ_T denote opaque data and a digital signature received from T used during SPC refresh for verifying the authenticity of cnt_S (details below). Note that id_I and cid act as indexing keys to access rct and rcs within {rct} and {rcs}.

We describe SPC Refresh informally. Let $\mathsf{rcs}^{cid,pk_T}_S$ denote a rcs_S including cid, pk_T, and a valid digital signature σ_T. Refresh includes four phases: (a) at the end of time period ω_ℓ, build a tuple $\left\langle\mathsf{rcs}^{cid,pk_T}_S\right\rangle_{S\in\mathcal{S}_I}$ for each couple $\langle cid,pk_T\rangle$ existing on any server $S\in\mathcal{S}_I$; (b) discard all tuples of cardinality equal or smaller than k_I; (c) within each remaining tuple, select the registration context with the maximum counter value cnt_S; (d) at the beginning of $\omega_{\ell+1}$, distribute all the selected registration contexts to all servers $S\in\mathcal{S}_I$, one for each couple cid, pk_T. Thus, at time period $\omega_{\ell+1}$, at each S there exist only registration contexts including $\langle cid,pk_T\rangle$ on which at least k_I+1 servers agreed at the end of ω_ℓ, and in presence of different counter values $\langle cnt_S\rangle_{S\in\mathcal{S}_I}$ for the same $\langle cid,pk_T\rangle$ only the maximum authentic value is kept.

```
      T ({rct})   C (I, Q, sc, ⟨ss⟩_S)   S ∈ Q ⊆ 𝒮_I ({rcs}, sc, ss_S)
                                         achallenge :
 1 :              <------ $ac_S$ ------     $ac_S \leftarrow_\$ \{0,1\}^\lambda$
                  acommands :
 2 :                foreach $S \in Q$ do
 3 :                  $m_S := \langle sc, \mathsf{id}_S, ss_S, ac_S\rangle$
 4 :                $\langle cm, \{w_S\}_{S\in Q}\rangle \leftarrow \mathsf{Com.commit}(\{m_S\}_{S\in Q})$
 5 :                $pcr_S := \langle cm, w_S\rangle$ ; $pcr := \{pcr_S\}_{S\in Q}$
 6 :                $h \leftarrow \mathsf{H}(sc, cm)$
 7 :    <------ $\mathsf{id}_I, M_a := h$ ------
      aresponse :
 8 :    $\langle cid, sk_T, cnt_T\rangle \leftarrow \{\mathsf{rct}\}.get(\mathsf{id}_I)$
 9 :    $cnt_T \leftarrow cnt_T + 1$;
10 :    $ad \leftarrow \langle \mathsf{H}(\mathsf{id}_I), cnt_T\rangle$
11 :    $\sigma_T \leftarrow \mathsf{Sig.sign}_{sk_T}(\langle ad, h\rangle)$
12 :    $\{\mathsf{rct}\}.\mathsf{update}(\{\mathsf{id}_I : cnt_T\})$
13 :    ------ $R := \langle cid, ad, \sigma_T\rangle$ ------>
                  acommandr :
14 :                foreach $S \in Q$ do $t_S := \mathsf{MAC}_{ss_S}(pcr_S, R)$
15 :                ------ $cr_S := \langle pcr_S, R, t_S\rangle$ ------>
                                         acheck :
16 :                                        if $\mathsf{MAC}_{ss_S}(pcr_S, R) \neq t_S$ then
17 :                                          return $\perp$
18 :                                        $\langle cid, ad, \sigma_T\rangle \leftarrow R$
19 :                                        $\langle pk_T, uid, cnt_S\rangle \leftarrow \{\mathsf{rcs}\}.\mathsf{get}(cid)$
20 :                                        $\langle h_{\mathsf{id}}, cnt_T\rangle \leftarrow ad$; $h \leftarrow \mathsf{H}(sc, cm)$
21 :                                        $m_S := \langle sc, \mathsf{id}_S, ss_S, ac_S\rangle$
22 :                                        if $h_{\mathsf{id}} \neq \mathsf{H}(\mathsf{id}_I)$ or $cnt_T \leq cnt_S$ or
23 :                                          $\mathsf{Sig.verify}_{pk_T}(\sigma_T, \langle ad, h\rangle) = 0$ or
24 :                                          $\mathsf{Com.verify}(cm, m_S, w_S) = 0$
25 :                                          then return $\perp$
26 :                                        $\{\mathsf{rcs}\}.\mathsf{update}(\{cid : \langle cnt_T, h, \sigma_T\rangle\})$
27 :                                        return $uid$
```

Fig. 4: Sketch of candidate SPC authentication.

Intuitively, the proposed SPC is IND-DCSA and EUF-CSA in the sense of Definitions 2 and 3 as long as Com is hiding (Line 4) and as long as MAC is EUF-CMA (Line 14). Let $\mathsf{sid}_{SPC}=\langle\mathsf{id}_I,\mathsf{H}(sc,cm),ad\rangle$ (trace seen both by T and any S), $q_I\geq 2k_I+1$, and $n_I\leq 3k_I+1$, then the protocol guarantees SPC-DR security (SPC with clone Detection against Restricted adversaries) in the sense of Definition 1: (i) during the same time period, violation of partnership implies duplication of ac_S, or violation of H collision resistance, Sig *unforgeability*, or of Com *binding* or *extractability*. During different time periods, refresh requires at least k_I+1 registration contexts agreeing on the same $\langle cid,pk_T\rangle$, thus k_I malicious servers cannot inject false contexts, and setting $q_I\geq 2k_I+1$ ensures that at least k_I+1 honest servers participate; (ii) pk_T is uniquely bound to a single uid and thus sk_T too, and the commit signed by T includes all the trace messages, thus authentication is cryptographically bound to session and subsession information; (iii) S rejects responses related to

[8]As anticipated in Section V-A, we move id_I outside of M_a in the formal syntax without any impact on actual implementations. Also note that we omit a server-side authenticator identifier denoted as uid in FIDO2 (which must not be confused with our notation of uid), which represents a technicality when channels are not secure because it is not authenticated by T, and thus can be manipulated by an adversary. As a confirmation, note that in [8, see Fig. 11, routine aVrfy] the server does not make use of it in any way.

[9]For brevity, we let Com.commit return both commit cm of set $\{m_S\}_{S\in Q}$ and witnesses $\{w_S\}_{S\in Q}$ (Line 4), where each w_S allows to verify that m_S has been included in cm through Com.verify (Line 24).

challenges released in previous time periods, and since servers are synchronous they do not release challenges related to different time periods simultaneously. Although registration is omitted here, mixing registration and authentication can be prevented through specialization of messages; (iv) to avoid clone detection, an adversary which executes FullAuth must authenticate with at least k_I+1 honest servers (that is, q_I-k_I) which were not included in the previous Clone, which also requested at least k_I+1 honest servers. Thus, the adversary needs at least $2k_I+2$ honest servers to win, and $3k_I+2$ in total (that is, also considering malicious servers). Since we set $n_I \leq 3k_I+1$, then the adversary cannot avoid detection; (v) clone detection based on counters cannot be tricked through message dropping, and refresh re-aligns the state to the maximum counter value seen by a server, which we know is an authentic value stored within the fresh authenticator because each rcs stores σ_T and related data h to verify it.

Figure 5 shows a specification for SPS authentication where V accepts an identity uid only if it receives at least k_I+1 attestations for the same identity uid, similarly to approaches described in [21]. Let $\mathsf{ssid}^S_{SPS} = \langle \mathsf{sid}_{SPC}, uid, ss_S \rangle$, then $m_F^{-1} : \mathsf{Ssid}_{SPS} \rightarrow \mathsf{Sid}_{SPC} \times \mathcal{S}$ exists because sid_{SPC} is cryptographically bound to ss_S, due to SPC winning condition (ii), and uid is the result of SPC authentication. Let $\mathsf{sid}_{SPS} = \langle cs, cv, av, \mathsf{id}_I, q_I, \{\mathsf{id}_S\}_{S\in Q}, \mathsf{id}_V \rangle$, then $m_I : \mathsf{Sid}_{SPS} \rightarrow \mathcal{V} \times \mathcal{I} \times \mathcal{S}^{q_I}$ exists because sid_{SPS} includes id_V, id_I and $\{\mathsf{id}_S\}_{S\in Q}$ of cardinality equal or greater than q_I. Also, $m_A : \mathsf{Sid}_{SPS} \rightarrow \mathsf{Sid}$ exists and is injective because values $\langle cv, av, \mathsf{id}_I, \{\mathsf{id}_S\}_{S\in Q}, \mathsf{id}_V \rangle$ are included or cryptographically bound to shared context sc, cv is included in sid_{SPC} via sc, and cs is cryptographically bound to cv. The specification is a secure K-N-SPS-DR in the sense of Definition 4: (vi) unique partnership between oracles of C and V is reduced to conditions *(i)* and *(ii)* of SPC and collision-resistance of H: id_I is directly included within sid_{SPC}; $cv, av, \{\mathsf{id}_S\}_{S\in Q}$ and id_V are included directly or after hashing within shared context sc by prepare; cs is also bound to cv by collision resistance of H; (vii) unique partnership between oracles of V and $Q \subseteq \mathcal{S}_I : |Q| \geq (k_{I_t}+1)$ is guaranteed by counting at least k_I+1 attestations with the same identity uid and information included in shared context within identify, plus EUF-CMA security of Sig for guaranteeing authenticity of information sent from servers to V; (viii) C must be bound to an authenticator T because each subsession is only known by C and by one (honest) server S, and thus only C can produce a command that is accepted by V. Since the proposed SPC is IND-DCSA, adversaries cannot obtain subsession information ss_S for a honest server S to authenticate at a parallel channel, thus a potentially injected attestation will fail validation of the identity provider, even with no validation in place within join, or fail partnership as for condition (ii) of SPC. An adversary may also try to re-use command M at a different authenticator to obtain a valid $R' \neq R$ without knowing subsession information, however it is not able to authenticate R' because the proposed SPC is also EUF-CSA.

```
$C\ (\mathsf{id}_V)$    $S(sk^\ell_I)$    $V(\mathsf{id}_I, q_I, pk^\ell_I = \{pk^\ell_S\}_{S\in Q})$
init :
  $cs \leftarrow\!\$\ \{0,1\}^\lambda$
  $cv \leftarrow \mathsf{H}(cs)$
                 $\dashrightarrow^{cv}$   begin :
                                        / Q selection policy is left generic.
                                        pick $Q \subseteq \mathcal{S}_I : |Q| \geq q_I$
                                        $av \leftarrow\!\$\ \{0,1\}^\lambda$
prepare :        $\xleftarrow{av, \mathsf{id}_I, \{\mathsf{id_S}\}_{S\in Q}}$
  $h := \mathsf{H}(cv, av)$
  $sc := \langle \mathsf{id}_V, \mathsf{id}_I, \{\mathsf{id}_S\}_{S\in Q}, h \rangle$
  foreach $S \in Q$ do
    $ss_S \leftarrow \mathsf{MAC}_{cs}(\langle sc, \mathsf{id}_S \rangle)$
  $\dashrightarrow^{sc, ss_S}$
........................ $uid \leftarrow$ Spc.Authenticate ........................
                 release :
                   / Check if our server is included in shared context.
                   / Typically also validate if audience $\mathsf{id}_V$ is allowed.
                   if $\mathsf{id}_S \notin sc$ then abort    / Slight abuse of notation.
                   $\sigma_S \leftarrow \mathsf{Sig.sign}_{sk^\ell_S}(\langle uid, sc, \mathsf{H}(ss_S) \rangle)$
  $\xleftarrow{vs_S}$       $vs_S := \langle uid, \sigma_S \rangle$
join :
  $vc := \left\langle cs, uid, \langle \sigma_S \rangle_{S\in Q} \right\rangle$
                 $\dashrightarrow^{vc}$   identify :
                                        if $\mathsf{H}(cs) \neq cv$ return $\perp$
                                        $\left\langle sc, \langle ss_S \rangle_{S\in Q} \right\rangle \leftarrow \mathsf{prepare}(\mathsf{id}_V, av, \mathsf{id}_I,$
                                                 $\{\mathsf{id}_S\}_{S\in Q}, cs)$
                                        $c := \{0\}$   / Dictionary with default value zero.
                                        foreach $S \in Q$ do :
                                          $t_S := \langle uid, sc, \mathsf{H}(ss_S) \rangle$
                                          if $\mathsf{Sig.verify}_{pk^\ell_S}(\sigma_S, t_S) = 0$ return $\perp$
                                          else $c[uid] := c[uid] + 1$
                                        endforeach
                                        foreach $uid \in c$ do
                                          if $c[uid] \geq (k_I + 1)$ return $uid$
                                        return $\perp$
```

Fig. 5: Sketch of candidate SPS authentication (dashed lines for confidential channels).

VIII. Conclusions and future work

We proposed the first protocol frameworks, security models and security games for survivable distributed passwordless authentication and Single Sign-On, and sketched candidate specifications based on established cryptographic assumptions. Major limitations may be related to the considered assumptions for distributed systems, related to synchronous servers, honest servers which do not crash and, during registration, reliable channels. Future work includes improving over these limitations, proposing more detailed specifications and related security proofs. Other improvements include investigating different classes of formal methods to verify security (e.g., computer-assisted symbolic analysis) and better strategies for composing security of authentication and SSO, and design alternative specifications with different trade-offs in terms of security, performance and usability.

Acknowledgments. This work has been supported by the project C4SI funded by the Emilia-Romagna region within the PR-FESR 2021-2027 (CUP E67G22000630003).

References

[1] FIDO Alliance Specifications. https://fidoalliance.org/specifications/.

[2] Detecting abuse of authentication mechanisms. Technical Report PP-20-1485, National Security Agency, Dec. 2020.

[3] Shashank Agrawal, Peihan Miao, Payman Mohassel, and Pratyay Mukherjee. PASTA: PASsword-based Threshold Authentication. In *ACM SIGSAC Conf. CCS*, 2018.

[4] FIDO Alliance. Client to Authenticator Protocol (CTAP) v2.1, Jun. 2021. https://fidoalliance.org/specs/fido-v2.1-ps-20210615/fido-client-to-authenticator-protocol-v2.1-ps-20210615.html.

[5] Manuel Barbosa, Alexandra Boldyreva, Shan Chen, and Bogdan Warinschi. Provable Security Analysis of FIDO2. In *CRYPTO*, 2021.

[6] Carsten Baum, Tore Kasper Frederiksen, Julia Hesse, Anja Lehmann, and Avishay Yanai. PESTO: Proactively Secure Distributed Single Sign-On, or How to Trust a Hacked Server. *IEEE EuroS&P*, 2019.

[7] Mihir Bellare and Phillip Rogaway. Entity authentication and key distribution. In *CRYPTO*, 1993.

[8] Nina Bindel, Cas Cremers, and Mang Zhao. FIDO2, CTAP 2.1, and WebAuthn 2: Provable security and post-quantum instantiation. In *IEEE Symp. S&P*, 2023.

[9] Joseph Bonneau, Cormac Herley, Paul C Van Oorschot, and Frank Stajano. The quest to replace passwords: A framework for comparative evaluation of web authentication schemes. In *IEEE Symp. S&P*, 2012.

[10] Ran Canetti, Rosario Gennaro, Amir Herzberg, and Dalit Naor. Proactive Security: Long-term protection against break-ins. *RSA Laboratories' CryptoBytes*, 1997.

[11] Damien Cash, Matthew Meltzer, Sean Koessel, Steven Adair, and Thomas Lancaster. Dark halo leverages solarwinds compromise to breach organizations, 2020. https://www.volexity.com/blog/2020/12/14/dark-halo-leverages-solarwinds-compromise-to-breach-organizations/.

[12] Yvo Desmedt. Society and group oriented cryptography: A new concept. In *Conf. Theory and Application of Cryptographic Techniques*, 1987.

[13] Danny Dolev and Andrew Yao. On the security of public key protocols. *IEEE Trans. Information Theory*, 29(2), 1983.

[14] Florian M. Farke, Lennart Lorenz, Theodor Schnitzler, Philipp Markert, and Markus Dürmuth. "You still use the password after all" – exploring FIDO2 security keys in a small company. In *USENIX 16th Symp. Usable Privacy and Security*, Aug. 2020.

[15] Daniel Fett, Ralf Küsters, and Guido Schmitz. The Web SSO Standard OpenID Connect: In-Depth Formal Security Analysis and Security Guidelines. In *IEEE Symp. CSF*, 2017.

[16] David Fisher, Richard Linger, Howard Lipson, Thomas Longstaff, Nancy Mead, and Robert Ellison. Survivable network systems: An emerging discipline. Technical Report CMU/SEI-97-TR-013, 1997.

[17] Qianwen Gao, Yuan Lu, Kunpeng Bai, Zhenfeng Zhang, and Yichi Tu. Thpla: Threshold passwordless authentication made usable and scalable. *IEEE Trans. Information Forensics and Security*, 2025.

[18] IETF. PKCE by OAuth Public Clients. RFC 7636, Sep. 2015.

[19] Leona Lassak, Annika Hildebrandt, Maximilian Golla, and Blase Ur. "It's Stored, Hopefully, on an Encrypted Server": Mitigating Users' Misconceptions About FIDO2 Biometric WebAuthn. In *USENIX Security*, 2021.

[20] Sanam Ghorbani Lyastani, Michael Schilling, Michaela Neumayr, Michael Backes, and Sven Bugiel. Is FIDO2 the Kingslayer of User Authentication? A Comparative Usability Study of FIDO2 Passwordless Authentication. In *IEEE Symp. S&P*, 2020.

[21] Federico Magnanini, Luca Ferretti, and Michele Colajanni. Flexible and survivable single sign-on. In *Cyberspace Safety and Security*, 2022.

[22] Kevin Milner, Cas Cremers, Jiangshan Yu, and Mark Ryan. Automatically detecting the misuse of secrets: Foundations, design principles, and applications. In *IEEE Symp. CSF*, 2017.

[23] Rafail Ostrovsky and Moti Yung. How To Withstand Mobile Virus Attacks. In *ACM Symp. PODC*, 1991.

[24] N. Sakimura, J. Bradley, M. Jones, B. de Medeiros, and C. Mortimore. OpenID Connect Core 1.0, Nov. 2014. https://openid.net/specs/openid-connect-core-1_0-final.html.

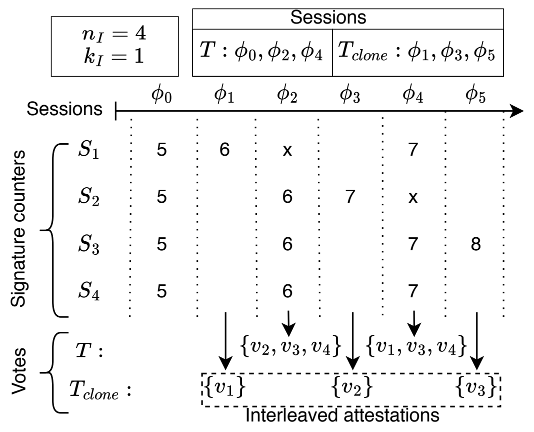
[25] Sijun Tan, Weikeng Chen, Ryan Deng, and Raluca Ada Popa. Mpcauth: Multi-factor authentication for distributed-trust systems. In *IEEE Symp. S&P*, 2023.

[26] W3C. Web Authentication: An API for accessing Public Key Credentials Level 2. https://www.w3.org/TR/webauthn-2/, Apr. 2021.

[27] W3C. Credential management level 1, 2022. https://w3c.github.io/webappsec-credential-management/#user-mediated.

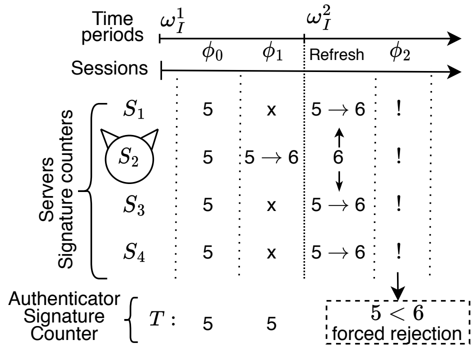
[28] Moti Yung. The "Mobile Adversary" Paradigm in Distributed Computation and Systems. In *ACM Symp. PODC*, 2015.

[29] Yuan Zhang, Chunxiang Xu, Hongwei Li, Kan Yang, Nan Cheng, and Xuemin Sherman Shen. PROTECT: Efficient Password-Based Threshold Single-Sign-On Authentication for Mobile Users against Perpetual Leakage. *IEEE Trans. Mobile Computing*, 2020.



$n_I = 4$
$k_I = 1$
Sessions
$T : \phi_0, \phi_2, \phi_4$
$T_{clone} : \phi_1, \phi_3, \phi_5$
Sessions
ϕ_0 ϕ_1 ϕ_2 ϕ_3 ϕ_4 ϕ_5
Signature counters
S_1 5 6 x 7
S_2 5 6 7 x
S_3 5 6 7 8
S_4 5 6 7
Votes
T : $\{v_2, v_3, v_4\}$ $\{v_1, v_3, v_4\}$
T_{clone} : $\{v_1\}$ $\{v_2\}$ $\{v_3\}$
Interleaved attestations

Fig. 6: Clone detection evasion via authentication interleaving



Time periods
ω_I^1
ω_I^2
Sessions
ϕ_0 ϕ_1 Refresh ϕ_2
Servers Signature counters
S_1 5 x $5 \rightarrow 6$!
S_2 5 $5 \rightarrow 6$ 6 !
S_3 5 x $5 \rightarrow 6$!
S_4 5 x $5 \rightarrow 6$!
Authenticator Signature Counter
T : 5 5
$5 < 6$
forced rejection

Fig. 7: Forced authentication rejection via false clone detection

Appendix A
Attacks examples

Clone detection evasion attack. Figure 6 shows an example of clone detection evasion where T and T_{clone} denote the legitimate authenticator and its clone. We denote as $\phi_i, i \in [0, 5]$ the challenge-response sessions participated by authenticators, and as v_j the identity attestation released by identity server j. Denied authentications are denoted as x. Authenticators T and T_{clone} execute the protocol in interleaved sessions: T_{clone} executes sessions ϕ_1, ϕ_3 and ϕ_5, while T executes sessions ϕ_2 and ϕ_4. In each session, T_{clone} only collects attestations from single distinct identity servers. All T sessions receive a denied authentication from the server that interacted with T_{clone} because, from the server's view, T presents a stale counter. However, T completes successfully because it receives a number of denied authentications that does not exceed the number of allowed malicious server $k_I = 1$. Thus, T does not detect the presence of T_{clone}. As a result, T_{clone} is able to authenticate after ϕ_5 by presenting interleaved identity attestations $\{v_1, v_2, v_3\}$.

Forced authentication rejection via false clone detection. Figure 7 shows an example of a forced authentication rejection attack. We consider authentication sessions throughout two time periods ω_I^1 and ω_I^2, separated by a Refresh of servers. Initially, in session ϕ_0, both authenticators and servers have signature counters equal to 5. In session ϕ_1, we consider a malicious server increasing its signature counter to 6, and we

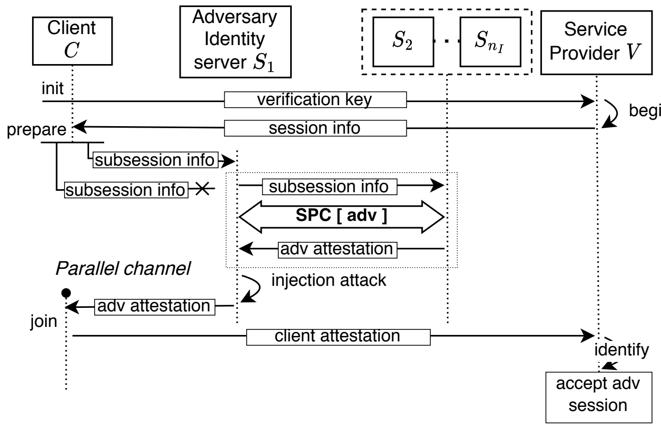


Fig. 8: Shared session attack

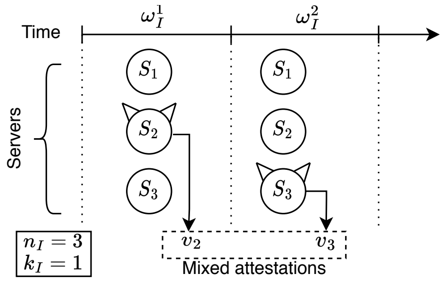


Fig. 9: Attestation mixing

assume that the Refresh procedure increases the counters of all servers to the maximum counter value, thus all servers set their counters to 6. Then, in session ϕ_2, authentication by the original authenticator is rejected by the server due to its counter value still set to 5. Note that this class of attack may change due to how clone detection and related authentication and refresh procedures are designed.

Shared session attack. Figure 8 shows an example of a shared session attack. The adversary controls identity server S_1 and is correctly registered at all remaining identity servers, which are assumed to be honest. The client starts SPS authentication with service provider V and forwards SPS session information to each identity server, as required by the protocol. The adversary obtains the victim session information and starts SPC authentication with honest identity servers by using the victim session information. As a result, each identity server issues an identity attestation of the adversary identity which is bound to the victim session information. Meanwhile, the client has opened a parallel communication channel with the adversary as a result of other attacks (e.g., CSRF, phishing). To conclude the attack the adversary builds an identity attestation collection and injects it into the parallel channel. The client sends the injected attestation collection to the service provider which authenticates the client under the adversary identity.

Attestation mixing. Figure 9 shows an example of identity attestation mixing in which malicious identity servers S_2 and S_3 release rogue attestations v_2 and v_3 in distinct time periods ω_I^1 and ω_I^2. The attack is successful if the service provider counts identity attestations v_2 and v_3 together and does not detect that they belong to distinct time periods.

Appendix B
SPC correctness definition

Correctness (SPC). For any identity provider I, servers $\mathcal{S}_I$, authenticator T, context sc and subsessions $\langle ss_S \rangle_{S \in \mathcal{S}_I}$, user identifier uid, and security parameter and threshold λ and k_I, such that T is initialized with $\mathsf{Spc.Tsetup(T)}$ (thus, $\{\mathsf{rct}_T\} = \emptyset$), and I and S_I are initialized as $\langle \mathsf{id}_I, \{\mathsf{id}_S\}_{S \in \mathcal{S}_I}, \Omega_I, q_I \rangle \leftarrow \mathsf{Spc.Isetup}(I, \mathcal{S}_I, k_I)$ (thus, $\{\mathsf{rcs}_S\} = \emptyset, \forall S \in \mathcal{S}_I$), for any $Q, Q'_{\ell,i} \subseteq \mathcal{S}_I : |Q| \geq q_I \wedge |Q'_{\ell,i}| \geq q_I$, where $i \in [x_\ell]$ for any $x_\ell \leq \mathsf{poly}(\lambda), \forall \ell \in [|\Omega_I|]$, if:

```
$\mathsf{Spc.Register}(C, T, Q, \mathsf{id}_I, uid)$ :
    $rc_S \leftarrow\!\!\$\ \mathsf{Spc.rchallenge}(\{\mathsf{rcs}_S\}), \forall S \in Q$
    $\langle M_r, pcr \rangle \leftarrow\!\!\$\ \mathsf{Spc.rcommands}(\mathsf{id}_I, uid, \langle rc_S \rangle_{S \in Q})$
    $\langle R_r, \{\mathsf{rct}_T\}' \rangle \leftarrow \mathsf{Spc.rresponse}(\{\mathsf{rct}_T\}, \mathsf{id}_I, M_r)$
    $\langle cr_S \rangle_{S \in Q} \leftarrow \mathsf{Spc.rcommandr}(pcr, R_r, \mathsf{id}_{S \in Q})$
    $\langle b_S, \{\mathsf{rcs}_S\}' \rangle \leftarrow \mathsf{Spc.rcheck}(\{\mathsf{rcs}_S\}, rc_S, uid, cr_S), \forall S \in Q$
    $\{\mathsf{rcs}_S\} := \{\mathsf{rcs}_S\}', \forall S \in Q$
**foreach** $\ell \in [|\Omega_I|]$ **do** :
    $\mathsf{Spc.Refresh}(I, \mathcal{S}_I, \omega_I^\ell)$ :
        $\langle \{\mathsf{rcs}_S\}' \rangle_{S \in \mathcal{S}_I} \leftarrow \mathsf{Spc.rcsrefresh}(\mathsf{id}_I, k_I, \langle \{\mathsf{rcs}_S\} \rangle_{S \in \mathcal{S}_I})$
        $\{\mathsf{rcs}_S\} := \{\mathsf{rcs}_S\}', \forall S \in \mathcal{S}_I$
    **foreach** $i \in [x_\ell]$ **do** $\mathsf{Spc.Authenticate(C, T, Q'_{\ell,i}, id_I, sc, \langle ss_S \rangle_{S \in Q'_{\ell,i}})}$ :
        $ac_S \leftarrow\!\!\$\ \mathsf{Spc.achallenge}(\{\mathsf{rcs}_S\}), \forall S \in Q'_{\ell,i}$
        $\langle M_a, pcr \rangle \leftarrow \mathsf{Spc.acommands}(\langle ac_S, ss_S \rangle_{S \in Q'_{\ell,i}}, sc)$
        $\langle R_a, \{\mathsf{rct}\}' \rangle \leftarrow\!\!\$\ \mathsf{Spc.aresponse}(\{\mathsf{rct}_T\}, \mathsf{id}_I, M_a)$
        $\langle cr_S \rangle_{S \in Q'_{\ell,i}} \leftarrow \mathsf{Spc.acommandr}(pcr, R_a, sc, \langle ss_S \rangle_{S \in Q'_{\ell,i}})$
        $\left\langle b'_{S,\ell,i}, \left\langle uid'_{S,\ell,i}, \{\mathsf{rcs}_S\}' \right\rangle \right\rangle \leftarrow \mathsf{Spc.acheck}($
                        $\{\mathsf{rcs}_S\}, ac_S, sc, ss_S, cr_S), \forall S \in Q'_{\ell,i}$
        $\{\mathsf{rcs}_S\} := \{\mathsf{rcs}_S\}', \forall S \in Q'_{\ell,i}$
```

then $b_S = 1, \forall S$ and $b'_{S,\ell,i} = 1 \wedge uid'_{S,\ell,i} = uid, \forall S, \ell, i$ holds with probability 1.

Appendix C
SPS correctness definition

Correctness (SPS). For any identity and service providers I and V, servers $\mathcal{S}_I$, authenticator T, client C, user identifier uid, and security parameter and threshold λ and k_I, such that T is initialized with $\mathsf{Spc.Tsetup(T)}$ (thus, $\{\mathsf{rct}_T\} = \emptyset$), if I and all S in S_I are successfully initialized with $\langle \mathsf{id}_I, \{\mathsf{id}_S, \Omega_I, q_I\}_{S \in \mathcal{S}_I} \rangle \leftarrow \mathsf{Spc.Isetup}(I, \mathcal{S}_I, k_I)$ (thus, $\{\mathsf{rcs}_S\} = \emptyset, \forall S \in \mathcal{S}_I$) and $\langle sk_I^0 = \{sk_S^0\}_{S \in \mathcal{S}_I}, pk_I^0 \rangle \leftarrow \mathsf{Sps.ikgen}(I, \mathcal{S}_I, k_I)$, for any $Q, Q'_{\ell,i} \subseteq \mathcal{S}_I : |Q| \geq q_I, |Q'_{\ell,i}| \geq q_I$ and $x_\ell \leq \mathsf{poly}(\lambda), \forall \ell \in [|\Omega_I|]$, if:

```
$\mathsf{Sps.Register}(C, T, Q, \mathsf{id}_I, uid)$ :
    $\mathsf{Spc.Register}(C, T, Q, \mathsf{id}_I, uid)$
**foreach** $\ell \in [|\Omega_I|]$ **do** :
    $\mathsf{Sps.Refresh}(I, \mathcal{S}_I, \omega_I^\ell)$ :
        $\mathsf{Spc.Refresh}(I, \mathcal{S}_I, \omega_I^\ell)$
        $\left\langle sk_I^\ell, pk_I^\ell \right\rangle \leftarrow \mathsf{Sps.ikrefresh}(\left\langle sk_I^{\ell-1}, pk_I^{\ell-1} \right\rangle, k_I)$
    **foreach** $i \in [x_\ell]$ **do** $\mathsf{Sps.Authenticate}(V, C, T, Q'_{\ell,i}, \mathsf{id}_I)$ :
        $\langle cs, cv \rangle \leftarrow\!\!\$\ \mathsf{Sps.init}(\mathsf{id}_V)$
        $av \leftarrow\!\!\$\ \mathsf{Sps.begin}(\mathsf{id}_I, \{\mathsf{id}_S\}_{S \in Q'_{\ell,i}}, cv)$
        $\left\langle sc, \langle ss_S \rangle_{S \in Q'_{\ell,i}} \right\rangle \leftarrow \mathsf{Sps.prepare}(\mathsf{id}_V, av, \mathsf{id}_I, \{\mathsf{id}_S\}_{S \in Q'_{\ell,i}}, cs)$
        $\left\langle uid'_{\ell,i} \right\rangle \leftarrow \mathsf{Spc.Authenticate}(C, T, Q'_{\ell,i}, \mathsf{id}_I, sc, \langle ss_S \rangle_{S \in Q'_{\ell,i}})$
        $vs_S \leftarrow \mathsf{Sps.release}(sk_S^\ell, uid, sc, ss_S), \forall S \in Q'_{\ell,i}$
        $vc \leftarrow \mathsf{Sps.join}(cs, av, sc, \langle ss_S, vs_S \rangle_{S \in Q'_{\ell,i}})$
        $uid''_{\ell,i} \leftarrow \mathsf{Sps.identify}(pk_I^\ell, cv, av, vc)$
```

then all protocols and routines do not abort and $uid = uid'_{\ell,i} = uid''_{\ell,i}, \forall \ell, i$ holds with probability 1.