Attribute-based Single Sign-On: Secure, Private, and Efficient

Tore Kasper Frederiksen
Zama Inc.
Paris, France
tore.frederiksen@zama.ai

Julia Hesse
IBM Research Europe – Zurich
Rüschlikon, Switzerland
JHS@zurich.ibm.com

Bertram Poettering
IBM Research Europe – Zurich
Rüschlikon, Switzerland
POE@zurich.ibm.com

Patrick Towa
Aztec Network
London, United Kingdom
patrick.towa@gmail.com

ABSTRACT

A Single Sign-On (SSO) system allows users to access different remote services while authenticating only once. SSO can greatly improve the usability and security of online activities by dispensing with the need to securely remember or store tens or hundreds of authentication secrets. On the downside, today's SSO providers can track users' online behavior, and collect personal data that service providers want to see asserted before letting a user access their resources.

In this work, we propose a new policy-based Single Sign-On service, i.e., a system that produces access tokens that are conditioned on the user's attributes fulfilling a specified policy. Our solution is based on multi-party computation and threshold cryptography, and generates access tokens of standardized format. The central idea is to distribute the role of the SSO provider among several entities, in order to shield user attributes and access patterns from each individual entity. We provide a formal security model and analysis in the Universal Composability framework, against proactive adversaries. Our implementation and benchmarking show the practicality of our system for many real-world use cases.

KEYWORDS

SSO, MPC, threshold cryptography, identity management

ACKNOWLEDGMENTS

This work has received funding from the EU Horizon 2020 research and innovation programme under grant agreement No 786725 OLYMPUS. Julia Hesse was supported by the Swiss National Science Foundation (SNSF) under the AMBIZIONE grant "Cryptographic Protocols for Human Authentication and the IoT". The work of Tore Frederiksen was done while at the Alexandra Institute, and part of Patrick Towa's work was done while at ETH Zurich.

1 INTRODUCTION

In 2022, the average number of online accounts maintained by a single person in the US or in Europe was above 70, most of which secured by passwords. The two main tools to tame the complexity of memorizing or storing authentication data such as passwords or cryptographic keys are (1) password managers or wallets secured with one main secret, and (2) Single Sign-On (SSO) systems. Password managers or cryptographic key wallets store all authentication data of a user, and reveal it if the user provides the main secret (e.g., a password, biometrics, or a hardware token). An SSO system dispenses with the need to set up credentials for all different accounts in the first place, and instead relies on a single "main" account at an identity provider (IdP). A user who can log in to that account gets redirected to its account at the service provider (SP), who trusts the IdP with having authenticated the user correctly. Hence, the main secret used to authenticate at the IdP takes a similar role to the main secret in the first solution, and a secure choice (e.g., two-factor authentication) of the main secret provides excellent protection of the user against adversaries who try to steal a user's identity.

Both the above approaches provide great relief to users when it comes to managing their accounts. On the other hand, there exist obvious drawbacks even if the main secret is chosen carefully, and potentially consists of multiple factors. Namely, wallet applications and the IdP all constitute single points of failure, and they must be fully trusted not to impersonate the user. In SSO, IdPs are even involved in the whole login procedure and can hence potentially track users' online behavior. The situation worsens if the IdP additionally vouches for users' attributes, such as being of age or not being from an embargoed country, as it then must even be trusted to handle users' private data.

In this work we investigate the design of a policy-based SSO system where access can be granted based on the user fulfilling a certain policy (e.g., being over age). The central design goal is to avoid the above mentioned main drawback of SSO, namely that a single IdP can impersonate the user or learn their private attributes (e.g., their birthday).

We recall the concept of SSO in a bit more detail. In SSO (e.g. the "Sign in with Google" option on a gambling website), a user registers with only one identity provider (**IdP**, here Google) and can later authenticate to that identity provider to obtain an access token for a service provider (**SP**, here the gambling website). The access token is a "bearer" token, i.e., an access request signed by the identity provider that has time-constrained validity (realized through the inclusion of timestamps in the signed message). A bearer token could be a JSON Web Token (JWT) and hence used for compatibility with authentication standards such as OAuth [53] and OIDC [83]. Alternatively the token could be XML and used with SAML [78]. These standards only support few signature schemes, generally only (EC)DSA or RSA. For compatibility, the bearer tokens of practical SSO schemes are hence restricted in the same way.

We put forward the notion of *attribute-based distributed SSO* (ab-dSSO), which has three features on top of SSO. First, in ab-dSSO, the role of the IdP can be distributed among arbitrarily many servers,

thereby obsoleting trust in a single entity or piece of hardware. Second, users can equip their accounts with arbitrarily many certified attributes (e.g., the user's birthday) obtained from attribute authorities. And third, access tokens are policy-based, and IdPs only issue a token if a user's attributes fulfill the token's policy (e.g., "age over 18 and resident of an appropriate state" to show they are legally allowed to do online gambling). Another example of the use of ab-dSSO could be to show to a loan provider that a user has a certain monthly income (e.g. through an attribute on a bank statement) and that they are a "resident of the EU, US, or Canada". In ab-dSSO, IdP servers are required to be oblivious of attributes and, depending on the message signed, the tokens can be untraceable and unlinkable. Unlinkable means that colluding SPs cannot find their common set of users, from the tokens they have received. Untraceable means that even colluding SPs and the IdP cannot discover which user a token was issued towards. Thus unlinkability would prevent the loan and gambling providers colluding and learning that a user trying to get a loan also does online gambling. On the other hand untraceability would ensure to the user that even a subpoenaed IdP and service providers would not be able to learn that the user does both gambling and has a loan. We discuss these features more in App. C.3. An ab-dSSO scheme further allows for efficient attribute revocation, works with any type of certified attribute that an attribute authority has vouched for,[1] supports selective disclosure and demonstration of predicates over attributes, and produces tokens of a standardized format (like ECDSA). The last property ensures that the loan and gambling websites would not need to add new code-dependencies on their backend in order to support more exotic cryptographic signature schemes.

1.1 Our Approach

We build an ab-dSSO scheme by combining (outsourced [39, 58]) multi-party computation (MPC) techniques with threshold cryptography. MPC can be used to compute any function on inputs provided by a set of parties, with participants being oblivious of each other's inputs. Thus, a first (generic) approach to building an ab-dSSO scheme would let IdP servers and a user engage in an MPC instance to obliviously verify the validity of a user's attribute certificates and to verify that the attributes fulfill a given policy. If these checks pass, the servers generate the token using a distributed signature scheme producing signatures of a standardized format. However, for a number of reasons the resulting ab-dSSO scheme does not scale well in the number of tokens generated, mainly because attribute certificate validation will be tedious. We hence improve upon the above generic proposal by *preprocessing* attribute certificates during registration, turning them into efficiently verifiable *attribute tokens* which have the form of message authentication codes (MACs) on attributes, generated from a key that is distributed among the IdP servers. This yields an ab-dSSO scheme with fast token generation but potentially slow attribute registration, which is efficient enough for many of the use cases we consider in this work. Using threshold cryptography, we can specify a version of this protocol that requires only a subset of IdP servers to be available during token generation,[2] and which allows recovery from server corruption or breach (*proactive security*).

We benchmark token generation of our scheme using the FRESCO framework [3], for five different policies such as a constraint on the user's age, or that they appear on a pre-approved list. We demonstrate an end-to-end latency of less than a second, for many policies in a realistic setting with the distributed IdP spread over multiple data centers in distinct countries. We furthermore show server throughput of two-digit policy validations per second and core for most of our test policies. Overall, our contributions are:

- We provide a strong formal model for attribute-based distributed Single Sign-On (ab-dSSO), compatible with existing standards.
- We work out the relation of ab-dSSO to competing cryptographic identity management primitives.
- We combine non-proactively secure maliciously secure, dishonest majority MPC with Shamir secret sharing to construct a scheme where data for MPC computations can be proactively refreshed and the MPC computation can be run by a t-out-of-n set of MPC servers. We thus reach a compromise between the security of dishonest majority MPC and the reliability on t-out-of-n secret sharing, while gaining proactive security.
- We provide a strong model for proactively secure threshold signing and a construction thereof, without relying on trusted hardware such as [11, 21]. We also highlight shortcomings of previous proofs of security in this setting [82] and discuss how they can conservatively be handled at low cost in our setting.
- Finally we combine these contributions to obtain an efficient ab-dSSO construction using MPC and threshold cryptography, which has a fast token generation phase that requires only a subset of the IdP servers to be responsive, at the cost of a slower but still feasible registration phase. We formally prove the security of our construction, provide an implementation, and benchmark several real-world use cases.

1.2 Related Work

Recently, several SSO services for demonstrating user attributes have emerged. Verimi [92] is a smartphone app that allows users to present digitized images of official documents such as ID, driving license, or vaccination pass. Users control which part of the stored information is shared with a service provider. Predicate proving is not possible, and the presentation in form of digitized images prevents compatibility with authentication standards such as OAuth. ID4me [57] supports OpenID/OAuth based identity management with "weak identities": The service verifies that claimed user IDs indeed belong to specified users; this connection is made on first login and is independent of real-world user identities. The servers can fully trace the user login behavior. UPPRESSO [51] achieves untraceability and unlinkability through the use of homomorphically computed single-use IDs. The protocol supports the OIDC flow, although it requires the service providers to run custom code.

The recent PASTA [2] and PESTO [11] protocols represent a major privacy-wise improvement. They implement provably secure

[1] Clarifying terminology: an *attribute authority* vouches for the validity of user attributes (e.g., a bank or federal office) by issuing *attribute certificates*. An *identity provider* (IdP) assists users with generating presentations of their identities. We often user the term *server* to refer to one part of our distributed IdP.

[2] Since we still rely on all servers to be available during registration, we cast our notion as "distributed" instead of "threshold".

distributed SSO protocols that allow users to bootstrap access tokens for arbitrary service providers from one password only. The systems support an arbitrary number of servers and prevent any proper subset of servers from impersonating users. Both PASTA and PESTO could trivially become attribute-based by sharing the attributes in plain with *each* server, a solution that however does not meet the strong privacy goals that we aim for in this work. In relation to our examples from the introduction it would mean that *each* IdP server stores *all* users' birthday and address information, making *each* of them a prime target for attackers looking to extract as much personal information as possible, on a large set of people. Another difference is that PASTA and PESTO are in the password setting, while our protocol relies on secure storage on the user side. A UC-secure password-based threshold key manager [20] could be used on top of our ab-dSSO scheme to allow users to store the required key material under a password in a similar security model.

Self-sovereign identity (SSI) is the concept of individuals or enterprises having the sole ownership of their digital identity, and fully controlling how and when personal data attached to their identity is shared and used. This implies that certified identity information or attributes, issued by different trusted third parties towards a specific user, will never be stored or managed in a centralized location. When attributes (e.g., age or liquidity) are needed to prove certain policies towards a third party (e.g., a vendor or a bank), a proof is constructed which can be presented towards the third party *without* disclosing any other piece of personal information. This is known as *minimal disclosure*. Until today, a practically efficient SSI system fulfilling both sole ownership and full control with minimal disclosure has remained elusive. Popular approaches based on a blockchain [65, 74] or attribute-based credentials (ABCs) [25, 68, 79] each suffer from one or another drawback, such as inefficient attribute revocation [68], expensive computations on the user side [25, 79], or unclear trade-offs between user-side resource requirements and trust assumptions [65, 74]. Many digital identity management solutions that are deployed today [25, 57, 77, 79, 92] rely on a fully trusted identity provider (**IdP**), which then "owns" its users' identities and can impersonate them on the Internet.

In this work, we provide a trade-off system where the user is not fully dependent on a single server but also not fully self-sovereign, as they rely on the honesty and availability of some fraction of servers. Similar to existing SSI schemes, users in our system can build up their digital identity by obtaining *attribute certificates* from external attribute authorities, such as federal offices or banks.

An overview of how the discussed systems and cryptographic techniques fare with respect to desirable privacy properties can be found in Tab. 1. We refer the reader to App. A for a formal comparison of ab-dSSO to related concepts such as attribute-based signatures or attribute-based credentials.

2 PRELIMINARIES

We write s for statistical and λ for computational security parameters. In Tab. 2 we maintain an overview of these and other symbols used throughout the article. We write $[n] = \{1, \ldots, n\}$. When a variable depends on another variable, we may use sub-script notation for emphasis: If the value of variable v is defined using another variable a, notation v_a associates v with a.

Writing conventions for ideal functionalities. We use the Universal Composability (UC) framework [26] and use the following conventions. The adversary gets notified about the contents of inputs and outputs and involved parties, excluding contents that are marked as *private*. If an input or output is completely private, the adversary did not even get notified that it happened. We further let the adversary acknowledge all inputs it is notified about. For outputs, we use *delayed* output to a party when the adversary gets to schedule its delivery. *Immediate* outputs are delivered without any delay. For brevity we assume the functionality to reject malformed inputs, or those re-using a subsession identifier *ssid*.

2.1 Adversarial Model

We detail the adversarial model that we assume throughout this paper. We in particular define *proactive* adversaries, taking departure in existing definitions [5, 56] but using slightly different terminology to better fit our setting. All signature and SSO schemes in this paper proceed in different phases, as illustrated in Fig. 1: a period during which servers sign under a certain key is called a *signing epoch*. At some point during such an epoch, servers *refresh* their key material, but meanwhile continue to sign under their old keys. This is important as otherwise delays in the interactive refreshing would cause interruptions to the signing service. As soon as the refresh is concluded, servers erase their old key material and switch to signing under the new key, at which point the next signing epoch starts. We define a *corruption span* to start right before a refresh is initiated and to end before the next refresh. While arbitrary corruption spans could be defined, our choice is motivated by the fact that refreshes are likely initiated by a server when detecting traces of a corruption, and/or at regular time intervals. Hence we note that a refresh can be initiated by either a corrupt or honest server. We call an adversary *static* if it announces, right before each corruption span, who is going to be corrupted during the next span, and servers remain corrupted throughout the whole span. We call an adversary *malicious* if it can fully control a corrupted party's actions.

It can be observed from the figure that, since a refresh always overlaps with one signing epoch, a server that is corrupt during the refresh might leak key material from two consecutive signing epochs to the adversary. To guarantee unforgeability, in our threshold setting, we cannot allow leakage of more than $t-1$ keys to the adversary. Hence, security of our schemes will rely on the adversary corrupting at most $t-1$ servers in two consecutive corruption spans. We call such an adversary *proactive*. In particular and as an example, if in the first corruption span in Fig. 1 already $t-1$ servers are corrupted, the proactive adversary cannot corrupt anybody in the second corruption span. Moreover, our refresh phase requires an honest majority among the servers, and hence we restrict the proactive adversary further to corrupt at most $\min\{t-1, n/2\}$ servers in two consecutive corruption spans.

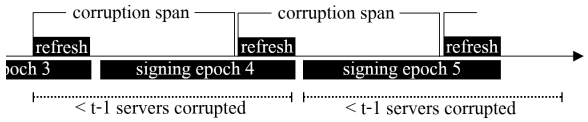

Figure 1: Illustration of proactive security.

Service	Trust model	Adaptive security	Proactive security	Standardized format	Lightweight on user	Lightweight on IdP(s)	Offline token verification	Revocation	Selective disclosure	Predicate-enabled	Unlinkability
Verimi [92]	Wallet/Cloud	-	-	○	●	-	●	●	●	○	○
ID4me [57]	Cloud	-	-	●	●	●	●	●	-	-	○
uPort [74]	Permissionless ledger	-	-	○	●	-	○	○	●	●	○
Sovrin [65]	Permissioned ledger, trusted agents	-	-	○	○	-	●	●	●	●	●
BBS+ [68]	Trusted IdP (Credential Issuer)	-	-	◕	◕	-	●	○	●	○	●
CL-ABC [22]	Trusted IdP (Credential Issuer)	-	-	○	◕	-	●	●	●	●	●
UPPRESSO [51]	Trusted IdP (Credential Issuer)	-	-	◕	●	●	●	-	-	-	●
CanDID [72]	Distributed IdP	○	○	○	○	◕	●	◕	●	●	●
PASTA [2]	Distributed IdP	○	○	●	●	●	●	-	-	-	●
PESTO [11]	Distributed IdP	●	●	●	●	●	●	-	-	-	◕
This work	Distributed IdP	○	●	●	●	○	●	●	●	●	●

Table 1: Identity Management/SSI systems, by the properties of Sect. 3. ● = satisfied, ○ = not satisfied, ◕ = partially satisfied/ inefficient, - = not applicable. Systems that support attributes additionally rely on trusted attribute authorities.

Table 2: Parameters and variables of our ab-dSSO scheme.

λ	Computational security parameter.
s	Statistical security parameter.
p	Modulus representing a field where attributes live.
c	A λ and p dependent variable defining $\mathbb{F}_{p^c}$.
N	The modulus of an RSA key used for signing.
t	The threshold for secret sharing.
n	The number of servers.
M	The message to be included in the token.
σ	A signature or token.
vk	A public verification key for a signature.
sk	A private signing key.
P	A policy.
A	A set of user attributes; $a_i \in A$ for $i \in [\lvert A\rvert]$ with $a_i \in \mathbb{Z}_p$.
π	A certificate/non-interactive proof.
S_i	A server, being part of the IdP realization.
U	A user of the IdP.
V	A verifier/service provider validating a user's request.

2.2 Threshold Signatures

A threshold signature scheme features n (signing) servers that initially run a distributed key generation that establishes individual shares of a global signing key pair. Servers can later compute partial signatures using their key shares, and a threshold t of partial signatures on the same message are necessary and sufficient to construct a full signature that verifies against the global public key. An adversary corrupting at most $t-1$ servers cannot construct signatures. Proactive threshold signatures additionally involve a refresh protocol during which all servers jointly compute a fresh set of shares of the (unmodified) global secret key, and start a new signing epoch. Refreshing heals from corruptions and leads to forward security: As long as at most $t-1$ shares in the *same* signing epoch are compromised, signatures remain unforgeable.

We introduce a UC definition of proactively secure threshold signatures, and demonstrate how to realize it by adapting Rabin's signature scheme [82] to our setting where only t servers participate in signing. For details see App. E.

3 ATTRIBUTE-BASED DISTRIBUTED SSO

Attribute-Based Distributed SSO (ab-dSSO) refers to Single Sign-On with additional features: Token generation happens in a distributed fashion, and the decision of whether a token is issued or not is based on whether the requesting user's attributes fulfill some policy. In principle, ab-dSSO involves four entities: users, attribute authorities (AA), service providers (SP), and identity providers (IdP, "servers"). Users hold attributes for which they receive attribute certificates from the AA. Users further, from the SP, receive token requests in association with policies; to satisfy such token requests, users contact the IdPs who, if the policies verify with respect to the user's attribute certificates, issue the tokens to the requesting user who relays them back to the SP.

In a formal model for ab-dSSO, the active entities are effectively just two, users and IdP, while the role of AA and SP is minimal: Regarding the AA, note their only task is to cryptographically vouch for the validity of users' attributes via attribute certificates π. These certificates depend on (static) attributes of the user that can be independently verified in a trusted process. In particular, they do not depend on prior protocol interactions or similar. That is, they can w.l.o.g. be assumed to have been issued 'a priori' and without interference of an adversary. It is thus conceptually sufficient to model AA as abstract sources of certificates π that can be (non-interactively) verified by evaluating a predicate $\mathsf{CAVfy}(vk_{\mathsf{CA}}, A, \pi)$, where vk_{CA} is the AA's public key and A is a user's claimed attribute (set). Regarding the SP, observing their task is merely to specify the message-policy pair that a requested token should satisfy, that they verify such tokens via a public function, and given they never communicate with IdPs directly but only by relaying messages via users, from a modeling perspective, SPs and users can naturally be joined up to a single entity.

Our work presents a secure and privacy-friendly *cryptographic core* of a policy-based SSO system. Building on a trustworthy certification service by an AA infrastructure as a necessary prerequisite, it shall guarantee that all interactions between SP, user, and IdP preserve privacy yet offer strong authentication. As, in resemblance with digital signatures, generated tokens shall be offline-verifiable, in line with established SSO solutions such as OAuth [53],

SAML [78], and OIDC [83], our formalizations of the above use a terminology close to that common for signature schemes.

Definition 3.1. An attribute-based distributed Single Sign-On (ab-dSSO) system is a set of interactive procedures run between a user U and servers $\mathcal{S} := \{S_1, \ldots, S_n\}$. The scheme is parameterized by a distributed signature scheme $\mathsf{TSIG} = (\mathsf{KGen}, \mathsf{Sign}, \mathsf{Vfy})$.

- **Setup phase.** The servers run TSIG.KGen to generate (sk, vk), where sk is shared among all servers and vk is output. Setup is run only once.
- **Registration phase.** On input (uid, A, π), where A is a set of attributes $A := (\mathtt{type}_j : a_j)_{j \in [m]}$, where $\mathtt{type}_j$ is a public label (e.g., "birthday") and a_j the actual value of the attribute (e.g., "1980-02-24"), a certificate π and a username uid, U engages in an interactive protocol with servers in $\mathcal{S}$. As a result, each server S_i either stores record $(\mathsf{uid}, \mathtt{type}_j, \mathsf{st}_i)$ for each $j \in [m]$ and the user outputs success, or both output failure. Registration can be called multiple times for the same uid, to subsequently store more type records for uid.
- **Signing phase.** On input (uid, M, P, A'), where uid is a username, M is a message, P is a policy and A' is an attribute set, U engages in an interactive protocol with servers in $\mathcal{S}$ and outputs either a token σ or failure, and the servers output a bit indicating whether failure happened or not. Signing can be run arbitrarily many times.

Correctness. Let vk denote the output of the setup phase, and let T denote the set of type records stored by the servers for uid at a given point in time. If a user calls signing with inputs (uid, M, P, A), if $T_A \subseteq T$ for T_A the set of all types in A, and $P(A) = 1$, then σ output by the user is a signature on M, P verifying under vk. This correctness property must hold for all ab-dSSO schemes as long as the user and all servers behave honestly.

3.1 Desired properties of ab-dSSO

The set of properties that we demand of an ab-dSSO scheme is essentially the combination of properties of SSO and attribute-based credential systems [2, 11, 22, 51, 72], with added protection against a potentially corrupted central IdP. We start with notions guaranteeing the functionality and practicality of the system and relate them to the example of the loan and gambling providers from the introduction.

Standardized token format. An ab-dSSO scheme shall produce signatures of a standard format, e.g., ECDSA or RSA. This ensures that service providers do not need to update *any* code to become compatible with our ab-dSSO scheme (unless they use attribute-based policies), and only minor updates, without the need of new libraries, if they use attribute-based policies.

Compatibility. An ab-dSSO scheme shall work with attribute certificates of arbitrary formats. This allows the IdP to integrate with *any* AA, without requiring them to do *any* code or policy update. Consider for example a governmental AA certifying a birthday and citizenship, a municipal/state AA certifying residence, and a bank certifying a regular monthly income.

Combined presentations. An ab-dSSO scheme shall allow users to combine attributes when requesting tokens. The gambling and loan providers both require information about the user from different authorities (governmental and municipal/state for gambling and governmental and bank for the loan).

Revocation. The IdP servers shall learn the *types* of attributes a user has registered so that they can revoke them individually and without relying on a collaboration with the user. Thus the SP can trust the reliability of the tokens issued by the IdP, while remaining agnostic to the AA.

Flexible policies. An ab-dSSO scheme shall generate tokens for any efficiently verifiable policy. The user sees the policy and can hence decide to refuse policies that would reveal unnecessary information about them. E.g., the user will know and approve that the gambling website learn their age is above a certain threshold, and that their residency is part of a publicly known set, allowing them to keep a reasonable level of privacy and anonymity, while providing the gambling provider with the data they legally require.

We next list the *security and privacy* properties that we demand from an ab-dSSO scheme.

Unforgeability. The adversary cannot obtain a token verifying under vk for any pair (M, P) unless (a) a corrupt user registered successfully with a set of attributes A with $P(A) = 1$, and (b) the remaining honest servers agree to participate in the token generation. A few corrupt IdP servers cannot impersonate the user, e.g. by signing arbitrary tokens, even if also the user is corrupt and colluding. This ensures both that the user can trust outsourcing SSO to the IdP and that the SP can put a high degree of trust in signed tokens even if the SP doesn't trust the user.

Attribute privacy. The adversary shall not be able to infer anything about the attributes used in registration and signing. The user can trust that their attributes remain private and hidden towards the IdPs and SP, even if some of the servers are corrupt.

Proactive security. A corruption of an IdP server followed by a "refresh" shall reset the adversary to be oblivious of the corrupted server's state. This ensures longevity of the system. In this work we are not interested in "trivial refreshing", i.e., resetting all entities and starting from scratch. (Such solutions would require all users to re-register and hence are impractical.)

Detection of online attacks. The IdP servers shall learn which uid a signing request is issued for, so that rate-limiting policies can be applied.

The above functionality requirements are complemented by:

Efficiency. An ab-dSSO scheme shall support multiple attribute authorities, a large amount of users, and have a throughput that allows handling many requests per second. Token verification by the SP must be fast and offline (without a need for the SP to interact with an AA or IdP). The handling of token requests shall be parallelizable on the IdP side. The user and SP latency shall be correspondingly small.

We next describe tracing protection of ab-dSSO. In standard SSO systems, IdPs and service providers can often trace which services a user is accessing, because the message and policy included in the token might contain the identity of the service provider and also information uniquely identifying a user, e.g. in $M =$ `"CasinoRoyal-20230219-johndoe@gmail.com"`. Protecting against curious IdPs collecting the data from the token generation for such messages on the level of cryptography seems to require IdPs to sign blindly, but unfortunately there are currently no blind signature

schemes that produce tokens of standardized format.[3] Therefore, the best protection that an ab-dSSO scheme can currently offer is protection against service providers who try to get a more complete picture of a user's online behavior, beyond the usage of the services offered by itself. For example, gambling and loan providers could aim at linking their users, allowing them to infer whether a user gambles on its loans.

Unlinkability against colluding SPs. Tokens generated by an ab-dSSO scheme shall not reveal whether they were created for the same uid. More details in App. C.3.

The above list does not explicitly encompass **selective disclosure** of attributes, which is a sought-after goal in identity management systems. The reason is that it is implied by the policy-based nature of the system, plus attribute privacy: an ab-dSSO scheme leaves it to the user to reject policies that would reveal too much information about the user's attributes, and it ensures via attribute privacy that the token does not reveal anything about the attributes used to generate the token beyond what the policy reveals.

We implicitly assume that the servers $\mathcal{S}$ "know" and trust the attribute authorities. This assumption is, first of all, needed to make the proof in the registration phase sensible, through boot-strapped trust in the AA. However, it is also necessary in order to boot-strap revocation, since $\mathcal{S}$ must know where to look to figure out if certain user attributes have been revoked.

3.2 Comparison ab-dSSO vs. classic SSO

We compare our approach and solution with what is achieved by current SSO systems like OIDC through Google. We do this by evaluating the SSO properties introduced above in the context of our loan and gambling website examples. While our solution meets all indicated properties, as we will see this is not the case for OIDC. Recall that for age verification, loan and gambling applications would typically require uploading a copy of a governmental ID, potentially with liveliness verification. The loan website might additionally require information on the user's salary, e.g., in form of a bank statement.

(1) The properties *standard token format*, *compatibility*, *combined presentations* are not met by OIDC as it handles attributes in a custom way at each SP depending on the AA, and results are not efficiently consolidated into a single token at the IdP side.
(2) Threshold, *proactive security* and *unforgeability* aren't met either as the single, centralized IdP could impersonate users.
(3) *Attribute privacy* is not met since the *full* attributes are shared directly with service providers, which both hurts privacy and allows for linkability.

3.3 Security Model

We introduce a UC functionality $\mathcal{F}_{\text{ab-dSSO}}$ to model an ab-dSSO scheme with the above guarantees. We use the writing conventions introduced in Sect. 2. At its core, $\mathcal{F}_{\text{ab-dSSO}}$ is a signature functionality. However, opposed to the standard signature functionality due to Canetti *et al.* [27], $\mathcal{F}_{\text{ab-dSSO}}$ cannot allow the adversary to determine how signatures look like. The reason is that signatures in the context of SSO are *objects of value*, while normal use cases of signatures only require their unforgeability, but not their *secrecy*. We hence follow the related literature [11, 18] and let the functionality itself create the signatures. Aspects of threshold signing are modeled along the lines of the distributed SSO scheme PESTO [11], where "Proceed" interfaces were introduced to allow IdPs to refuse services. We omit the password authentication parts of PESTO, but newly introduce the attribute aspect to the functionality, which is however relatively simple: $\mathcal{F}_{\text{ab-dSSO}}$ stores registered attributes in uid-specific accounts. If user uid wants to obtain a token for message-policy pair (M, P) under attribute set A later, $\mathcal{F}_{\text{ab-dSSO}}$ only hands out the token if $P(A) = 1$ *and* A is a subset of the attributes registered for uid. Another modification from [11] is that we allow the token generation phase to work with only a threshold t of all n servers, while registration is still allowed to involve all n servers. Security in both phases holds as long as at most $t - 1$ servers are corrupted. This allows for analyzing protocols that aim particularly at speeding up the token generation phase. We now explain the functionality in detail, using the [X.Y] markings in the functionality code, and indicating in boldface the properties from above. We first do an honest walkthrough, and explain handling of corrupt users and servers separately afterwards.

Key generation. $\mathcal{F}_{\text{ab-dSSO}}$ is parameterized with a digital signature scheme (KGen, Sign, Vfy) and, if [K.1] all servers assist in the setup procedure, $\mathcal{F}_{\text{ab-dSSO}}$ [K.2] generates its own key pair (vk, sk). From that point on, the other interfaces of $\mathcal{F}_{\text{ab-dSSO}}$ can be called and $\mathcal{F}_{\text{ab-dSSO}}$ will produce and verify tokens w.r.t. this one key pair.

Attribute registration. Any party can call $\mathcal{F}_{\text{ab-dSSO}}$ to register attributes by providing a user name uid, attributes A, and attribute certificate π obtained from an AA. A parses as $(\texttt{type}_j : a_j)_{j\in[m]}$ for some m, i.e., can contain m attributes of different types. $\mathcal{F}_{\text{ab-dSSO}}$ is parameterized with an arbitrary (**compatibility**) verification algorithm $\mathsf{CAVfy}(vk_{\mathsf{CA}}, \cdot, \cdot)$ that is used to verify a proof (third argument) on the attributes (second argument) against the public key vk_{CA} (first argument) of an AA. If the certificate is not valid, or A does not contain a pair $\texttt{uID}:\texttt{uid}$, then [R.2,PR.2] registration fails, indicated by flag b. Servers [R.4] learn about the registration request and which attribute types should be added to uid (enabling efficient **revocation** on a per-type basis), but crucially [R.1] do not learn A nor π (**attribute privacy**). Servers can then signal their willingness to participate by sending ProceedReg for the corresponding subsession. Registration can only be completed if [PR.1] all servers participate. To complete the registration, $\mathcal{F}_{\text{ab-dSSO}}$ creates [PR.3] one record $(\texttt{account}, U, \texttt{uid}, \texttt{type}_j, a_j)$ for each attribute in A, where U is the party identifier of the user who ran the registration. As soon as such records exist, the functionality is ready to produce tokens for user U.

Signing. A successfully registered user U can now request token generation by providing the message to be signed M, username uid, policy P and attribute set A' that should be used to satisfy the policy. The functionality first [S.3] informs the adversary whether this is a valid request, i.e., the policy is fulfilled, the types match, and the correct uid is contained. $\mathcal{F}_{\text{ab-dSSO}}$ proceeds with signing [S.4] only if all these checks pass. $\mathcal{F}_{\text{ab-dSSO}}$ does not care whether all attributes were registered in the same registration query, enabling **combined presentations** of attributes from different authorities. If all the

[3] Our ab-dSSO construction *does* achieve some form of protection against IdPs: It *hides tokens* from IdPs as long as only a subset of them is corrupt. See discussion in App. C.3.

above checks pass, the servers get [S.5] notified about the signing request, but without learning A' (**attribute privacy**). To signal their participation in a token generation session, servers can input `ProceedSign` with the corresponding subsession identifier. This allows to implement arbitrary rate-limiting policies on a per-uid basis, where policies to counter online attacks can be decided by the application (**detection of online attacks**). As soon as the threshold of t participating servers (who all need to be in the same signing epoch) is reached [PS.1], $\mathcal{F}_{\text{ab-dSSO}}$ [PS.3] generates a signature σ for message (M, P) using the Sign algorithm and sk, installs a [PS.4] signature record that will allow successful verification of σ, and [PS.5] outputs the signature to U. Neither the signature nor the record includes uid. This, together with the fact that no subset of the t servers actually learns the signature, enables the issuance of **untraceable and unlinkable tokens**. We note however that the policy might de-anonymize a user, but it is then up to the user to not accept such policies from a relying party.

Verification and revocation. Everyone can check validity of signatures from P, M, σ under a verification key vk'. If $vk' = vk$, i.e., verification is requested for the verification key associated with $\mathcal{F}_{\text{ab-dSSO}}$, then the signature records are used to determine whether σ verifies or not: the output is set to true only if a record $(\texttt{sigrec}, P, M, \sigma, vk, \texttt{true})$ exists. Since such records only get created through successful signing requests, **unforgeability** is enforced. For $vk' \neq vk$, $\mathcal{F}_{\text{ab-dSSO}}$ uses the Vfy algorithm of the signature scheme to determine the result. Allowing verification for "incorrect" public keys is necessary to avoid that $\mathcal{F}_{\text{ab-dSSO}}$ implies a trusted certification authority. Finally, $\mathcal{F}_{\text{ab-dSSO}}$ enables revocation of attributes on a per-type-per-uid basis: if one server wishes to expire an attribute entry, $\mathcal{F}_{\text{ab-dSSO}}$ destroys the corresponding record and hence it cannot be used in signing requests anymore.

Adversarial influence and leakage. The adversary [PS.2] learns whether the policy is fulfilled and can still prevent successful token generation by not sending `sign-ok` for the corresponding subsession. The adversary also [R.3] learns whether a proof verifies upon registration, the correct uid is contained in the attributes, and the types of the attributes match the policy. $\mathcal{F}_{\text{ab-dSSO}}$ implies that a user has access to cryptographic material that uniquely identifies them as the one having registered the attributes. This is reflected in $\mathcal{F}_{\text{ab-dSSO}}$ by binding usage of an account to the party identifier who registered said account [S.3]. This restriction is lifted [S.2] in case the cryptographic material is leaked to the adversary, i.e., if the user gets corrupted. However, $\mathcal{F}_{\text{ab-dSSO}}$ prevents "mix-and-match" attacks of attributes issued for different uid's or attributes of different types, by enforcing presentations to use attribute records with all same uid and the correct public type [S.4]. $\mathcal{F}_{\text{ab-dSSO}}$ does not leak any attributes or attribute certificates to the adversary ([R.1],[S.1]), and it keeps the adversary from learning tokens generated by honest users ([PS.6]).

Refresh. Any server can indicate that it wants to move to the next epoch. The functionality acts only [Rf.1] if the server does not skip any epoch, and if it has not already started refreshing in the current epoch. As soon as all servers [Rf.2-3] joined the refreshing procedure in the current epoch, [Rf.5] the next epoch is entered and the servers are notified about it. The adversary can [K.3,Rf.4] delay completion of individual servers, and it can cause all servers to [Rf.6] abort an ongoing refresh procedure. $\mathcal{F}_{\text{ab-dSSO}}$ keeps [K.3,Rf.5] individual epoch counters epoch_i for every server S_i to allow for asynchronous completion of the refresh phase, i.e., one server signs under the new keys already while another one is still waiting for the final message in a refresh invocation. Avoiding such situations would require costly methods to enforce synchrony, and we hence design $\mathcal{F}_{\text{ab-dSSO}}$ with individual counters, to enable the analysis of more efficient SSO schemes where servers might be "off" by 1 regarding their epochs.

4 CONSTRUCTION

We are now ready to present our ab-dSSO scheme. The central idea is to turn any valid proof π of attribute possession held by a user into an efficiently verifiable *attribute token* v. Both validity check and conversion are performed as a secure outsourced multi-party computation, such that the individual IdP servers neither learn the actual attributes nor the attribute tokens. The latter consist essentially of affine-linear (information theoretic) message authentication codes (MACs) in a field $\mathbb{F}_p$ representing the domain of an MPC computation. That is, the MAC is

$$v_a = a \cdot \Delta + \beta_a \mod p$$

on the message a which is the attribute, generated with MAC key (Δ, β_a). This key is distributed among all IdPs. Δ is a long-term component of the IdPs and β_a is a (distributed) random attribute-specific component that IdPs generate for each attribute upon registration, and specific to each user uid. As part of the registration the user gets v_a as private output of the outsourced MPC protocol, whereas the servers each get their share of β_a. It is then the (distributed) storage of the tuple $(\text{uid}, \texttt{type}, \beta_a)$ which allows the user with uid to convince IdPs that they have successfully registered attribute a of type `type` (e.g., `type=age`) before. Concretely this is done by the user giving a and v_a as private input to the outsourced MPC computation. The servers then validate the MAC on a using their shares of Δ and β_a, in MPC. The idea of using an information theoretic MAC is seen in many non-outsourced MPC schemes, for example in the SPDZ family [40, 41]. While most recent SPDZ schemes use only Δ as MAC key, i.e., $\beta_a = 0$, this does not securely work in our ab-dSSO construction, where a malicious user could linearly combine such MAC tags into tags for new attributes. Hence, we introduce β_a as contribution to the MAC key which is unique per attribute-uid pair. Since the user obtains at most one MAC constructed with β_a, the above attack is prevented. At the same time, we conveniently use β_a as a privacy-preserving (i.e., hiding) and binding attribute identifier that can be stored by the IdPs. Furthermore, different from previous MPC schemes, our usage of the MAC is completely independent of how the underlying MPC scheme is realized and allows us to keep a reactive state of user-data for users that can come and go and who need not participate in the *underlying* MPC computation. We allow the values Δ and β_a to be Shamir secret-shared [86] between n servers, so that a subset ("threshold") of t servers is sufficient for token generation. While this increases reliability, it also means that we can tolerate $t - 1$ corrupted servers without losing on unforgeability of tokens. Figure 4 shows the flows of the protocol, slightly simplified with only one attribute instead of many and without threshold signing. We provide additional explanations of our SSO scheme in App. B.

$\mathcal{F}_{\text{ab-dSSO}}$ is parametrized by security parameter λ, threshold $t \le n$, algorithms (KGen, Sign, Vfy) and CAVfy$(vk_{\text{CA}}, \cdot, \cdot)$, and initially undefined integer counters $\text{epoch}_1, \ldots, \text{epoch}_n$. $\mathcal{F}_{\text{ab-dSSO}}$ interacts with servers $\Sigma := \{S_1, ..., S_n\}$ (specified in the *sid*), arbitrary other parties and an adversary $\mathcal{A}$.

Key Generation and Attribute Registration

On receiving (KeyGen, *sid*) from server S_i
- Ignore if a record (key, *sk*, *vk*) exists
- If (KeyGen, *sid*) was received [K.1] from all S_i
  - [K.2] Record (key, *sk*, *vk*), $(vk, sk) \leftarrow_R \text{KGen}(1^\lambda)$
  - For $i = 1, \ldots, n$ do:
    - On [K.3] (KeyConf, *sid*, S_i) from $\mathcal{A}$, set $\text{epoch}_i \leftarrow 0$ and immediately output (KeyConf, *sid*, *vk*) to S_i.

On (Register, *ssid*, uid, $A := (\text{type}_j : a_j)_{j\in[m]}, \pi$) **with** [R.1] **private** $a_j, j \in [m], \pi$ **from party** U:
- Set $b^{\text{vf}} \leftarrow (\text{CAVfy}(vk_{\text{CA}}, A, \pi) \wedge \text{uID:uid} \in A)$
- For $i \in [m]$ do:
  - Find index j s.t. $\text{type}_j = \text{uID}$ and if $a_j \neq$ uid then set $b^{\text{uid}} = 0$, else set $b^{\text{uid}} = 1$
  - Receive (Register, *ssid*, uid′, $(\text{type}'_j)_{j\in[m]}, S_i$) from $\mathcal{A}$ and if there is any j such that $\text{type}'_j \neq \text{type}_j$ or if uid ≠ uid′ then set $b^{\text{type}} = 0$
  - Set [R.2] $b \leftarrow 0$ if any of $b^{\text{vf}}, b^{\text{type}}, b^{\text{uid}}$ is 0
  - Send [R.3] (Register, *ssid*, $b^{\text{vf}}, b^{\text{uid}}$) to $\mathcal{A}$
  - Record (register, *ssid*, U, uid, A, b)
  - Send an [R.4] immediate output (Register, *ssid*, uid′, $(\text{type}'_j)_{j\in[m]}$) to $S_i \in \Sigma$

On (ProceedReg, *ssid*) **from server** S_i:
- Retrieve record (register, *ssid*, U, uid, A, b) and parse $A = (\text{type}_j : a_j)_{j\in m}$ for some m
- If there exists epoch $\in \mathbb{N}$ such that (ProceedReg, *sid*, *ssid*) was received from [PR.1] all servers S_i while $\text{epoch}_i = \text{epoch}$, and [PR.2] $b = 1$, then [PR.3] record (account, U, uid, type_j, a_j) for $j \in [m]$ and send [PR.4] delayed output (Registered, *ssid*, uid) to U and all $S_i \in \Sigma$.

Signing

On (Sign, *ssid*′, uid, $M, P, A' := (\text{type}'_j : a'_j)_{j\in[m']}$) **with** [S.1] **private** $a'_j, j \in [m']$ **from party** U:
- Find index j s.t. $\text{type}_j = \text{uID}$ and if $a_j \neq$ uid then set $b^{\text{uid}} = 0$, else set $b^{\text{uid}} = 1$. Set $b^{\text{pol}} = P(A')$. If records (account, $\bar{U}$, uid, type_j, a_j) exists for each $j \in [m']$, with either $\bar{U} = U$ or [S.2] $\bar{U}$ corrupt, then set $b^{\text{type}} = 1$, else set $b^{\text{type}} = 0$
- Send [S.3] (Sign, *ssid*′, $b^{\text{pol}}, b^{\text{uid}}, b^{\text{type}}$) to $\mathcal{A}$
- Proceed only if [S.4] $b^{\text{pol}} = b^{\text{uid}} = b^{\text{type}} = 1$
- Record (sign, *ssid*′, U, uid, M, P, A') and send a [S.5] delayed output (Sign, *ssid*′, uid, $M, P, (\text{type}_j)_{j\in[m']}$) to all $S_i \in \Sigma$.

On (ProceedSign, *ssid*′) **from server** S_i:
- Retrieve record (sign, *ssid*′, U, uid, M, P, A').
- If there exists epoch $\in \mathbb{N}$ such that (ProceedSign, *sid*, *ssid*) was received from [PS.1] t servers S_i while $\text{epoch}_i = \text{epoch}$:
  - Retrieve (key, *sk*, *vk*)
  - Send [PS.2] (sign-ok, *ssid*′) to $\mathcal{A}$ and receive back (sign-ok, *ssid*′)
  - Set [PS.3] $\sigma \leftarrow \text{Sign}(sk, (M, P))$, abort if (sigrec, $M, P, \sigma, vk, 0$) exists.
  - Record [PS.4] (sigrec, $M, P, \sigma, vk, 1$) and [PS.5] send a delayed output (Signature, *ssid*′, σ) with private σ to U.

Figure 2: $\mathcal{F}_{\text{ab-dSSO}}$ modeling distributed attribute-based token generation.

Verification

On (Verify, M, P, σ, vk') **from party** V **privately:**
- If [V.1] (sigrec, M, P, σ, vk', f') exists, set $f \leftarrow f'$
- Else, [V.2] record (sigrec, M, P, σ, vk', f) where f is determined as follows:
  - If [V.3] $vk = vk'$, set $f \leftarrow 0$, else set [V.4] $f \leftarrow \text{Vfy}(vk', (M, P), \sigma)$.
- Send an immediate private output (Verified, M, P, σ, vk', f) to V.

Revocation

On (Revoke, uid, type) **from server** S_i:
- Delete record (account, $*$, uid, type, $*$).

Refresh

On receiving (Refresh, *sid*, e) from server S_i
- Ignore the input if $\text{epoch}_i \neq e - 1$ or if $\text{refreshing}_i = 1$
- Set $\text{refreshing}_i \leftarrow 1$
- If $\text{refreshing}_i = 1$ for all $i \in [n]$:
  - On (Refresh, *sid*, e, S_i) from $\mathcal{A}$, set $\text{epoch}_i = e$ and send an immediate output (Epoch, *sid*, e) to S_i

On receiving (Refresh, *sid*, abort) from $\mathcal{A}$, set [Rf.5] $\text{refreshing}_i = 0$ for all $i \in [n]$.

Figure 3: $\mathcal{F}_{\text{ab-dSSO}}$, cont'd.

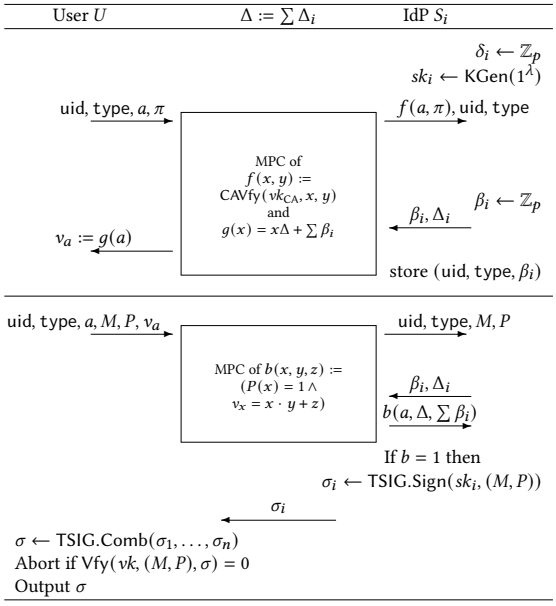

Figure 4: Core of our ab-dSSO scheme. We show a simplified setting of a user registering (top) and requesting a token (bottom) for only one attribute a, omitting the check of certificate π containing uid **during registration.**

Adding proactive security. Recall that proactive security means that a corrupt server can become honest again after a corruption, in the sense that the data an adversary has learned from that server can be rendered useless. This is achieved through rerandomizing shares. In our ab-dSSO scheme this is done in the *Refresh* phase, described in Fig. 7. In Fig. 10 we show the methods needed to provide proactive security on the MAC shares, in the manner shown in Fig. 1. In a similar manner, the methods in $\mathcal{F}_{\text{psThrSig}}$ show how to achieve proactive security of the underlying signing keys the servers need to store, with Fig. 13 and Fig. 14 (App. E) giving a realization based on RSA.

For (re)sharing the MAC shares we use Pedersen's verifiable secret sharing (VSS) scheme [80]. Unlike Feldman's [43] verifiable

Setup phase. Executed by each server S_i for $i \in [n]$ on input (KeyGen):

- If vk not yet defined, S_i sends (KeyGen, sid) to $\mathcal{F}_{\text{psThrSig}}$ and waits to receive back (KeyConf, sid, vk).
  //Threshold generation of long-term MAC key:
- If $[\Delta^{(j)}]_{j\in 0,\dots,t-1}$ not yet defined, do $[\Delta^{(j)}] \xleftarrow{\text{rnd}} \mathcal{F}^{ssid}_{\text{reqABB}}$ for $j = 0,\dots,t-1$. Let $f_{[\Delta]}(X) := \sum_{j=0,\dots,t-1}[\Delta^{(j)}]x^j \bmod p$.
- Compute $[\Delta_j] = [f_{[\Delta]}(j)]$ and send $([\Delta_j], S_j) \xrightarrow{\text{open}} \mathcal{F}^{ssid}_{\text{reqABB}}$ for $j \in [n]$.
- Execute Pedersen$^+$.MPC-Share of Fig. 10 on $[\Delta_j]$ for each $j \in [n]$ s.t. server S_i receives back $\{\text{priv} := \Delta_{j,i}, t_{j,i}\}, \{\text{pub} := A_{j,0},\dots,A_{j,t-1}\})$ for $j \in [n]$.
- Server S_i stores $(\Delta_i, \{\Delta_{j,i}, t_{j,i}, A_{j,0},\dots,A_{j,t-1}\}_{j\in[n]})$ and outputs (KeyConf, sid, vk).

Registration phase. Run between U and all S_i for $i \in [n]$.
User U, on input (Register, $ssid$, uid, A, π), does:
//Input attributes and proof into the MPC functionality:

- Send $\pi \xrightarrow{\text{priv}} \mathcal{F}^{ssid}_{\text{reqABB}}$.
- Parse each element in A as $\text{type}:a$ and send $\text{type} \xrightarrow{\text{pub}} \mathcal{F}^{ssid}_{\text{reqABB}}$ and $a \xrightarrow{\text{priv}} \mathcal{F}^{ssid}_{\text{reqABB}}$.
- Find index j s.t. $\text{type}_j = \text{uID}$, and abort if $a_j \neq \text{uid}$.
- Let m denote the number of elements in A. Send message $(ssid, \text{uid}, (\text{type}_j)_{j\in m})$ to each server S_i.

Upon receiving message $(ssid, \text{uid}, (\text{type}_j)_{j\in m})$ from user U, S_i outputs (Register, $ssid$, uid, $(\text{type}_j)_{j\in m}$).

Upon input (ProceedReg, $ssid$), each S_i does:
//Obliviously evaluate validity of attributes and belonging to uid*:*

- Send $\text{uid} \xrightarrow{\text{pub}} \mathcal{F}^{ssid}_{\text{reqABB}}$ and $vk_{\text{CA}} \xrightarrow{\text{pub}} \mathcal{F}^{ssid}_{\text{reqABB}}$.
- Find index j s.t. $\text{type}_j = \text{uID}$, send $([a_j], \{S_1,\dots,S_n\}) \xrightarrow{\text{open}} \mathcal{F}^{ssid}_{\text{ABB}}$, receive back a_j and abort if $a_j \neq \text{uid}$.
- Query (compute, $ssid$, ...) of $\mathcal{F}_{\text{reqABB}}$ to compute
$$\text{CAVfy}(vk_{\text{CA}}, \text{type}_1:[a_1],\dots,\text{type}_m:[a_m],[\pi]),$$
and only continues below if the result is 1.

//Obliviously compute a MAC v on fresh random value β for each attribute:

- Send $\Delta_i \xrightarrow{\text{priv}} \mathcal{F}^{ssid}_{\text{reqABB}}$.
- Send $[\beta^{(0)}_{\text{uid},j}], [\beta^{(1)}_{\text{uid},j}],\dots,[\beta^{(t-1)}_{\text{uid},j}] \xleftarrow{\text{rnd}} \mathcal{F}^{ssid}_{\text{ABB}}$. Let $f_{[\beta_{\text{uid},j}]}(X) := \sum_{k=0,\dots t-1}[\beta^{(k)}_{\text{uid},j}]x^k \bmod p$ the j-th polynomial for $j = 1,\dots,m$. *//Threshold sampling of attribute-specific MAC keys*
- Compute $[\beta_{l,\text{uid},j}] \leftarrow f_{[\beta_{\text{uid},j}]}(l)$ for $l \in [n]$.
- Compute $[\Delta] \leftarrow f_{[\Delta]}(0) = \sum_{j\in[t]}[\Delta_j] \cdot \prod_{l=1,l\neq j}^{t} \frac{l}{l-j} \bmod p$.
- For $j = 1,\dots,m$ compute the attribute-specific MAC
$$[v_{\text{uid},j}] \leftarrow [a_j]\cdot[\Delta] + [\beta_{\text{uid},j}] \bmod p$$
and send $([v_{\text{uid},j}], U) \xrightarrow{\text{open}} \mathcal{F}^{ssid}_{\text{ABB}}$. *//MACs are given to user.*
- Send $([\beta_{l,\text{uid},j}], S_l) \xrightarrow{\text{open}} \mathcal{F}^{ssid}_{\text{ABB}}$ for $l \in [n]$.
- After having received (output, $ssid$, $\beta_{i,\text{uid},j}$) from $\mathcal{F}_{\text{reqABB}}$ for $j = 1,\dots,m$, execute Pedersen$^+$.MPC-Share of Fig. 10 on $[\beta_{l,\text{uid},j}]$ for each $j \in [m]$, $l \in [n]$ s.t. server S_i receives back $\{\text{priv} := \beta_{l,\text{uid},j,i}, t_{l,\text{uid},j,i}\}, \{\text{pub} := A_{l,\text{uid},j,0},\dots,A_{j,t-1}\})$ for $l \in [n]$.
- Server S_i stores $(\text{uid}, \text{type}_j, \{\beta_{i,\text{uid},j}\}_{j\in[m]}, \{\{\beta_{l,\text{uid},j,i}\}_{j\in[m]}, t_{l,i}, A_{l,0},\dots,A_{l,t-1}\}_{j\in[n]})$ and outputs (Registered, $ssid$, uid).*//Store attribute-specific MAC keys.*
  Ignore all subsequent messages with identifier $ssid$.

After having received (output, $ssid$, $v_{\text{uid},j}$) from $\mathcal{F}_{\text{reqABB}}$ for $j = 1,...,m$, the user U stores $(\text{uid}, \text{type}_j, a_j, v_{\text{uid},j})_{j\in[m]}$ and outputs (Registered, $ssid$, uid).

Figure 5: Our ab-dSSO scheme, setup, registration, calling macro Pedersen$^+$ of Fig. 10 and functionalities $\mathcal{F}_{\text{reqABB}}$ (Fig. 12) and $\mathcal{F}_{\text{psThrSig}}$ (Fig. 18).

Signing phase.
Executed by each server S_i for $i \in [T]$ where $T \subset [n]$ with $|T| = t$.

Upon input (Sign, $ssid'$, uid, M, P, A'), user U does:

- Ignore the query if $P(A') \neq 1$, or if this is not the first one for $ssid'$. Parse A' as $(\text{type}_j : a_j)_{j\in[m]}$ for some m
- Find index j s.t. $\text{type}_j = \text{uID}$, abort if $a_j \neq \text{uid}$
- Send $\text{type}_j \xrightarrow{\text{pub}} \mathcal{F}^{ssid'}_{\text{reqABB}}$ and $v_{\text{uid},j}, a_j \xrightarrow{\text{priv}} \mathcal{F}^{ssid'}_{\text{reqABB}}$ for $j \in [m]$; drop the query if some type_j is not stored
- Input (Sign, $ssid'$, (M,P)) to $\mathcal{F}_{\text{psThrSig}}$
- Send message $(ssid', \text{uid}, M, P, \{\text{type}_j\}_{j\in[m]})$ to all S_i, $i \in [n]$.

Upon receiving message $(ssid', \text{uid}, M, P, (\text{type}_j)_{j\in[m]})$ from user U' and output (Sign, $ssid'$, (M,P)) from $\mathcal{F}_{\text{psThrSig}}$, server S_i does the following:

- Send $\Delta_i \xrightarrow{\text{priv}} \mathcal{F}^{ssid'}_{\text{reqABB}}$.
- For each type_j, $j \in [m]$ do:
  - Retrieve record $(\text{uid}, \text{type}_j, \beta_{i,\text{uid},j})$
  - Find index j s.t. $\text{type}_j = \text{uID}$, send $([a_j], \{S_1,\dots,S_n\}) \xrightarrow{\text{open}} \mathcal{F}^{ssid}_{\text{ABB}}$, receive back a_j and abort if $a_j \neq \text{uid}$.
  - Send $\beta_{i,\text{uid},j} \xrightarrow{\text{priv}} \mathcal{F}^{ssid'}_{\text{reqABB}}$
    //Check if the MAC provided by user is correct:
  - Compute
$$[\Delta] \leftarrow f_{[\Delta]}(0) = \sum_{i\in T}[\Delta_i] \cdot \prod_{l\in T, l\neq i} \frac{l}{l-i} \bmod p$$
    and, for $j \in [m]$,
$$[\beta_{\text{uid},j}] \leftarrow f_{[\beta_{\text{uid},j}]}(0) =$$
$$\sum_{i\in T}[\beta_{i,\text{uid},j}] \cdot \prod_{l\in T, l\neq i} \frac{l}{l-i} \bmod p.$$
  - Abort if
$$[v_{\text{uid},j}] \neq [a_j]\cdot[\Delta] + [\beta_{\text{uid},j}] \bmod p.$$
    //Check if the policy is fulfilled:
  - Query (compute, $ssid'$, ...) of $\mathcal{F}_{\text{reqABB}}$ to verify that
$$P(\{\text{type}_j : [a_j]\}_{j\in[m]}) = 1,$$
- Output (Sign, $ssid'$, uid, M, P, $(\text{type}_j)_{j\in[m]}$).

On input (ProceedSign, $ssid'$), each server S_i does:

- Input (ProceedSign, $ssid'$) to $\mathcal{F}_{\text{psThrSig}}$

On receiving output (Signature, $ssid'$, (M,P), σ) from $\mathcal{F}_{\text{psThrSig}}$, user U outputs (Signature, $ssid'$, σ).

Revocation.
Upon input (Revoke, uid, type), broadcast this message to all S_i and delete record (uid, type, $*$).

Figure 6: Our ab-dSSO scheme, signing and revocation, calling functionalities $\mathcal{F}_{\text{reqABB}}$ (Fig. 12) and $\mathcal{F}_{\text{psThrSig}}$ (Fig. 18).

secret sharing scheme, Pedersen's scheme does not require the message shared to be of high entropy, and hence is not brute-forceable. The messages we need to share are in $\mathbb{F}_p$, where attributes and the information theoretic MACs live. Thus we imagine p to be of 32-128 bits, hence Feldman would not be secure here. Methods Share, Verify and Reconstruct of Fig. 10 describe Pedersen's scheme. The method Recover is an application of Pedersen's VSS, showing how to validate, and potentially reconstruct, shares during *Refresh*, after a *corruption span* (assuming an honest majority). I.e., it is Pedersen's verifiable secret sharing used to achieve proactive secret sharing. Method MPC-Share constructs a verifiable secret sharing of share s_i within an MPC functionality, in a way that ensures that s_i can be trusted even if server S_i contributing it acts maliciously.

Verification.
On input $(\texttt{Verify}, M, P, \sigma, vk)$, a party sends $(\texttt{Verify}, sid, vk, (M, P), \sigma)$ to $\mathcal{F}_{\text{psThrSig}}$, receives back $(\texttt{Verified}, vk, (M, P), \sigma, f)$ and outputs $(\texttt{Verified}, M, P, \sigma, vk, f)$.

Refresh.
On receiving $(\text{Refresh}, sid, S_i)$, the servers in Σ proceed as follows:

- Send $(\text{Refresh}, sid, S_i)$ to $\mathcal{F}_{\text{psThrSig}}$ and receive back $(\text{Epoch}, sid, \text{epoch}_i, S_i)$
- Define the set $G_i = \{\Delta_i, \left\{\left\{\beta_{i,\text{uid},j}\right\}_{j\in[|A_{\text{uid}}|]}\right\}_{\text{uid}\in Y}\}$ where A_{uid} is the set of registered attributes for user with uid and Y is the set of all uid used for registration and for each $s_i \in G_i$ and let $t_i, \{s_{j,i}, t_{j,i}, A_{j,0}, \ldots, A_{j,t-1}\}_{j\in[n]}$ be the associated values to s_i received from Pedersen$^+$, *//All parties will have G_i of equal size, but with their individual shares*
- For each $s_i \in G_i$ execute Pedersen$^+$.Recover on S_i's values $(s_i, t_i, \{s_{j,i}, t_{j,i}, A_{j,0}, \ldots, A_{j,t-1}\}_{j\in[n]})$ and let the result be $(s'_i, t'_i, \{s'_{j,i}, t'_{j,i}, A'_{j,0}, \ldots, A'_{j,t-1}\}_{j\in[n]})$. Replace the current values with the result. I.e. $s_i := s'_i$, $t_i := t'_i$, $s_{j,i} := s'_{j,i}$, $A_{j,0} := A'_{j,0}, \ldots, A_{j,t-1} := A'_{j,t-1}$.
- For each $s_i \in G_i$ and its associated t_i, execute Pedersen$^+$.Share on 0, 0 such that each server S_j gets $\{\texttt{priv} := (s'_{i,j}, t'_{i,j})\}$, $\{\texttt{pub} := A'_{i,0}, \ldots, A'_{i,t-1}\}$,
  - abort if Pedersen$^+$.Verify on $(s'_{j,i}, t'_{j,i}), A'_{j,0}, \ldots, A'_{j,t-1})$ rejects for any $j \in [n]$.
  - Abort if it is not a 0-sharing, i.e. if $A'_{i,0} \neq 1$.
  - Update backup shares, s.t. $s_i := s_i + \sum_{j\in[n]} s'_{j,i} \mod p$ and associated $t_i := t_i + \sum_{j\in[n]} t'_{j,i}$ along with $A_{i,l} := A_{i,l} \prod_{j=0}^{t-1} A'_{j,l}$ for $j \in [n]$ and $l \in [t-1] \cup \{0\}$.
- For each $s_i \in G_i$ each party executes Pedersen$^+$.Share on values (s_i, t_i) for associated value t_i, thus each party S_j for $j \in [n]$ learn $\{\texttt{priv} := (s'_{i,j}, t'_{i,j})\}$, $\{\texttt{pub} := A'_{i,0}, \ldots, A'_{i,t-1}\}$ and does the following:*//Note that $s_i, t_i, A_{i,0}, \ldots, A_{i,t-1}$ have already been refreshed in the previous step*
  - Execute Pedersen$^+$.verify on $(s'_{j,i}, t'_{j,i}, A'_{j,0}, \ldots, A'_{j,t-1})$ and abort if it rejects.
  - Abort if the newly shared values are inconsistent with the ones from the previous refresh, i.e. if $A_{j,0} \neq A'_{j,0}$.
  - Update the back-up shares by setting $s_{j,i} := s'_{j,i}$, $t_{j,i} := t'_{j,i}$ and $A_{j,l} := A'_{j,l}$ for $l = 0, \ldots, t-1$.
- Output $(\text{Epoch}, sid, \text{epoch}_i, S_i)$.

Figure 7: Our ab-dSSO scheme, verification and refresh, calling macro Pedersen$^+$ of Fig. 10.

Note that later Share ensures consistency using commitments to shares, which MPC-Share, that is used to produce the initial sharings during Setup and Registration of our ab-dSSO scheme, cannot rely on.

Efficient integration of VSS and MPC. One final subtle difference between the methods described in Fig. 10 compared to the proactive secret sharing protocol of Pedersen's verifiable secret sharing scheme [80] is that we do sharing over an extension field $\mathbb{F}_{p^c}$, instead of a prime field $\mathbb{F}_q$ with a large subgroup. While the protocol could easily work over $\mathbb{F}_q$, it would then also be required that our MPC scheme works over a similar field, as otherwise reconstructing the Shamir secret sharing computed in MPC-Share would yield a value $s_i + \gamma \cdot p$ for some γ. Doing MPC over a field $\mathbb{F}_q$ with a prime large enough to ensure that the discrete logarithm problem is hard, would be very slow, since it would require $\log_2(q) \geq 2048$. Note we only use MPC to do computation of smaller numbers, in a domain of size similar to the domain of a regular CPU. By having $s_i \in \mathbb{F}_p$ we can represent it as an element in $\mathbb{F}_{p^c}$, by interpreting it as the constant term of the polynomial representation of an element in $\mathbb{F}_{p^c}$, with all non-constant terms being 0. This allows the Pedersen commitments to work in the expected manner. During *Refresh* we can reconstruct by doing Lagrange interpolation just over $\mathbb{F}_p$ instead of $\mathbb{F}_{p^c}$, for the message part of the commitment. Still, the randomness for the Pedersen commitments do need to be in $\mathbb{F}_{p^c}$, as otherwise s_i would not be statistically hidden. The bindingness of the commitments now depends on the hardness of the discrete log problem in $\mathbb{F}_{p^c}$ which is easier than in a field $\mathbb{F}_q$ where the discrete logarithm problem is hard, and thus c must be picked such that $p^c > q$. See Sect. 4.1.1 for a discussion of concrete values.

To summarize, we have described how Pedersen's verifiable secret sharing can integrate with an MPC functionality to afford proactive security of our ab-dSSO scheme through the methods in Fig. 10. Concretely these are used as part of the *Setup*, *Registration* and *Refresh* phases of our ab-dSSO scheme (Fig. 5 and Fig. 7) to allow proactive secret sharing of the MAC shares and keys. Specifically each server t-out-of-n shares each of their shares with all other servers in a verifiable manner, using MPC-Share during the *Setup* and *Registration* phases. We call these "shares-of-shares" *back-up shares*. This allows a quorum of t servers to "help" another server, recovering from a corruption, to relearn their shares using Recover, which is then executed during the *Refresh* phase. After using Recover it is then possible for each server to re-share their shares using Share, which renders the old shares an adversary might have learned useless.

4.1 Formal Protocol Description

Our ab-dSSO scheme is an interactive protocol executed between arbitrary users and n servers $S_1, \ldots, S_n$. It is parameterized with an algorithm $\mathsf{CAVfy}(vk_{\mathsf{CA}}, \cdot, \cdot)$ and a distributed signature scheme $\mathsf{TSIG} = (\mathsf{KGen}, \mathsf{Sign}, \mathsf{Comb}, \mathsf{Vfy})$. All parties have access to an ideal outsourced multi-party computation functionality $\mathcal{F}_{\mathsf{reqABB}}$ (see App. D.2) where computation on some user U's private input is carried out by $S_1, \ldots, S_n$ by an instance of functionality $\mathcal{F}_{\mathsf{reqABB}}(n, p)$ (or, more shortly, $\mathcal{F}_{\mathsf{reqABB}}$). For brevity we omit session identifier *sid* from all inputs, outputs and messages of the protocol, and use the abbreviated notation introduced in App. D.1. We make the following assumptions about formats:

- We assume each user is conceptually a UC party and they provide attribute sets A as input, the elements of which can be parsed as $(\texttt{type}_i : a_i)_{i\in[n]}$, where $\texttt{type}_i$ is a public attribute type (e.g., birth date, social security number, or country of residence). We assume policies to be compatible with this format.
- We assume certificates π to be issued for attribute sets that contain the attribute $\texttt{uID:uid}$, which is unique system-wide. This is not a restriction, since attribute authorities bind certificates to unique identities anyway. Just as these authorities, we use the unique identifiers uid to ensure that Alice cannot fill her account with attributes from stolen certificates.

Theorem 4.1. *Our ab-dSSO scheme (see Figs. 5,6,7) securely realizes $\mathcal{F}_{ab\text{-}dSSO}$ parameterized with t, n, algorithms $\mathsf{SIG} = (\mathsf{KGen}, \mathsf{Sign}, \mathsf{Vfy})$ and $\mathsf{CAVfy}(vk_{\mathsf{CA}}, \cdot, \cdot)$ in the $(\mathcal{F}_{\text{psThrSig}}, \mathcal{F}_{\mathsf{reqABB}})$-hybrid model, where $\mathcal{F}_{\text{psThrSig}}$ is parameterized with (the same) SIG, and there exist secure and server-side authenticated channels between users and each S_i, a broadcast channel among all S_i, and participants face a static proactive and malicious adversary (as defined in Sect. 2.1) corrupting at any given time only a minority of the servers. The distinguishing*

advantage of any PPT environment is upper bounded by

$$\frac{Q_r \cdot m}{2 \cdot p} + \frac{Q_{ps} \cdot m}{2 \cdot p} + \frac{5 \cdot (t-1) \cdot (Q_r \cdot m + 1) \cdot (1 + Q_{ref})}{2 \cdot p^c} + \mathsf{Adv}^{\mathsf{eDL}}(p^c, g, h) + \mathsf{Adv}^{\mathsf{EUF-CMA}}_{\mathsf{SIG}}(1^\lambda),$$

where Q_r is an upper bound on the number of registration queries, Q_{ps} is an upper bound on the number of ProceedSign *queries, Q_{ref} is an upper bound on* Refresh *queries, m is the number of attribute types, p is the field size and $\mathsf{Adv}^{\mathsf{eDL}}(p^c, g, h)$ is the advantage of the adversary in solving the discrete logarithm problem of h with base g in the extension field $\mathbb{F}_{p^c}$.*

The full proof is deferred to App. F.

Proof sketch. The simulator is in full control of the hybrid functionalities $\mathcal{F}_{\mathsf{reqABB}}$ and $\mathcal{F}_{\mathsf{psThrSig}}$ and hence can choose the MAC keys Δ and $\beta_{\mathsf{uid},j}$ for each attribute, without necessarily knowing the attributes itself. The simulator leaves it to $\mathcal{F}_{\mathsf{ab\text{-}dSSO}}$ to decide whether π verifies during registration, and whether the policy is fulfilled during signing. Corrupt users can be checked for malicious behavior by checking whether they provide the correct MACs for the attributes they want to use, as they need to input them into the simulator-controlled $\mathcal{F}_{\mathsf{reqABB}}$. Due to usage of a universally composable signature scheme through $\mathcal{F}_{\mathsf{psThrSig}}$, the simulator does not need to worry about simulating signature shares, as no such objects exist in a world with $\mathcal{F}_{\mathsf{psThrSig}}$. Hence the complexity of, e.g., share simulatability is outsourced to the realization of $\mathcal{F}_{\mathsf{psThrSig}}$. The only values that the simulator is missing for a perfect simulation of a real execution are attribute values and proofs. However, as described in Sect. 3.3, the simulator learns from $\mathcal{F}_{\mathsf{ab\text{-}dSSO}}$ whether a user inputs attributes that fulfill the policy, and whether a proof verifies when a user registers. Since actual signatures are otherwise independent of the attributes (i.e., only message and policy are signed), this information is enough to simulate either successful or unsuccessful registration and signing transcripts. □

4.1.1 Security parameters. Operating our ab-dSSO solution in practice requires instantiating all its building blocks with reasonable security parameters. As the employed primitives are of diverse types (signatures schemes, MPC, secret sharing, information-theoretic MACs), we provide insight into how we selected parameters for our implementation (see Sect. 5). Our guideline for all computational primitives was to achieve what is commonly known as the 128-bit security level. This arguably represents a robust choice, independently of whether proof artifacts like tightness factors are taken into account or not (like in our case).

Some of the building blocks are defined in the UC framework and instantiated with solutions presented in independent work. As is standard in the UC domain, for these primitives we assume a common asymptotic security parameter λ and leave it to articles considering their construction to translate asymptotic to concrete values. For our RSA-based building blocks we follow the proposals of standardization bodies [10] and instantiate moduli N with composite numbers of bit-length 3072 (or more). For statistical security definitions (in particular in the MPC setting), we accept bounds in the order of $< 2^{-80}$. (Bounds of the order $< 2^{-128}, \ldots, < 2^{-256}$ seem overkill for adversaries that are not computational.) For collision resistant hashing we employ SHA256.

An interesting case where our solution seemingly accepts too lose bounds is for MAC unforgeability: Recall that, to enable efficient MPC computations, our MAC tags are only 64 bits in size (we benchmark also with just 32 bits), which, on first sight, might invite for forgery attacks via MAC tag guessing. As generally recognized in cryptography, prominently for instance in [75], MAC tags may be short in practice as invalid tags immediately indicate active attacks to which servers can react with a variety of non-cryptographic measures including throttling the number of connection attempts or even banning users. While, in this sense, we feel that 64 bit tags offer sufficient security, it is straightforward to increase this to, say, 80 bits, at the expense of slightly less efficient MPC.

As discussed above, our instantiation of Pedersen's verifiable secret sharing is over the multiplicative group of $\mathbb{F}_{p^c}$ instead of over that of some $\mathbb{F}_q$. As the binding property of Pedersen's scheme is DLP-based, the pair p, c has to be chosen carefully to resist DLP attacks in extension fields [9, 54]. An overview on the state of the art of the latter is provided in [49, Tab. 1]. For fields of medium characteristic, $2 \ll p \ll 2^{1024}$, the current DLP-breaking record is reported for $p \approx 2^{25}$ and $p^c \approx 2^{1425}$, with no cryptanalysis advances since 10 years. In this light, our choice of $p \approx 2^{64}$ and $p^c \approx 2^{4096}$ seems reasonable (if not conservative). (Again, at the expense of less efficient MPC, switching to $p \approx 2^{80}$ for a better security margin is always possible.)

5 IMPLEMENTATION

While MPC allows us to validate any polynomial-time computable policy, for concreteness we only consider policies over the binary relations of *equality*, *less-than*, and *set-membership* and assume the operands are private values from $\mathbb{F}_p$ and the output is 0 or 1. We consider equality $([a], [b]) \to [c]$, less-than $([a], [b]) \to [c]$ and set membership $([a], \{[b_i]\}_{i \in [m]}) \to [c]$ for $a, b, b_1, \ldots, b_m \in \mathbb{F}_p$ and $c \in \{0, 1\}$. We also allow negation (NOT), conjunction (AND) and disjunction (OR) on each of these relations. Protocols for equality and less-than which only use calls to $\mathcal{F}_{\mathsf{ABB}}$ are Protocols 3.6 and 3.7 of [29]. Set membership can be realized by computing $[z] = \prod_{i \in [m]}([a] - [b_i])$ followed by equality $([z], 0)$. Logical operators on indicator bits are also efficient to realize. E.g., AND as $[a] \wedge [b] = [a] \cdot [b]$, OR as $[a] \vee [b] = [a] + [b] - [a] \cdot [b]$ and NOT as $\neg[a] = 1 - [a]$. Validating a MAC requires a single multiplication gate and one equality check.

We illustrate the complexity of all the operations in Tab. 3. The base policies (equality, less-than, set-membership) can be applied by themselves or they can be combined to allow for more rich policies. An example of a base policy for "less-than" could be "born before 1955", which would prove that the user is a senior citizen. Another example could be the usage of "set-membership" to prove that the user is a citizen of a certain set of countries. Concretely we note that these base policies are sufficient to implement both our loan and gambling examples since "less-than" can be made to "greater-than" by subtracting the input, and value to compare with, from p. Furthermore, state residency and country citizenship are just special cases of set-membership of sets of at most 50, respectively 195. A salary bound can again be realized using "greater-than" or even set-membership if a few possible brackets are used. The gate complexity for arbitrary policies can the easily be computed by adding together

Table 3: Multiplication gates complexity in MPC over $\mathbb{F}_p$. Based on the work of Catrina *et al.* [29]. Promised operands are in $[0; 2^{p'}]$. m is the cardinality of the set.

	# gates	depth
AND / OR / NOT	1 / 1 / 0	1 / 1 / 0
MAC check	$2 \cdot (\lvert A\rvert + 1)$	2
equality	$p' + 4\lceil \log_2(p') \rceil$	4
less-than	$3p' - 4$	$\log(p') + 2$
set memb.	$m + p' + 4\lceil \log_2(p') \rceil - 1$	$4 + \lceil \log_2(m) \rceil$

the relevant operations from Tab. 3. In Tab. 4 we sketch concrete policies and evaluate their *multiplicative* gate complexity along with concrete benchmarks. These numbers can then also be used to approximate the execution time of more advanced policies, by adding the execution time of each base policy together. The depth of the combined policy will then be the maximum of the circuit depth of each base policy added to $\log(\beta)$, where β is the amount of base policies. This overhead comes in order to account for the logical operations for combining the policies. Since the overhead of each benchmark includes setting up a network between servers, along with handshaking, we expect that for a β in the single digit, adding the execution time of the base policies together will upper-bound the total time of the advanced policy. We benchmark the following base policies:

Same value Proving that two attributes from $\mathbb{F}_p$ are the same. This consists of a single equality check.

Same object Proving that an attribute is the same as a publicly known, potentially big, object. This consists of $\lceil 256/\log_2(p) \rceil$ equality checks and $\lceil 256/\log_2(p) \rceil - 1$ AND gates, by validating equality checks on each $\lceil 256/\log_2(p) \rceil$ component of a SHA-256 hash digest of an object.

Range Proving an integer attribute is within *any* range in $\mathbb{F}_p$, e.g., proving one's age is in a certain range. This consists of doing two comparisons of $\lceil \log_2(p) \rceil$ integers through an AND gate.

Country Proving citizenship among a set of countries. Countries are represented as integers. This consists of a set membership test of a set of at most $m = 200$ elements.

Pre-approved Proving to have an attribute from a list of 1.000 pre-approved values. This consists of set membership for a set of $m = 1.000$ elements.

We chose to benchmark our protocols using $p = 32$ and $p = 64$, to reflect the domain sizes of common CPUs. We note that the MACs will also be in $\mathbb{F}_p$, and while the MACs themselves are information theoretic and cannot be brute-forced *offline* by a client, our ab-dSSO scheme allows a malicious client to authenticate using only the MAC, and thus could allow an *online* brute-force attack. We find it reasonable that the servers would simply throttle brute-force attempts on MACs since they necessarily knows the uid of the user trying to authenticate, and thus lock down the user's account after some failed attempts. This is similar to passwords or OTP, the latter of which have significantly less entropy. Furthermore to ensure no adversary can amount a denial of service for a legitimate user, the servers can require the user to authenticate before executing the signing phase using a method which is not brute-forceable. This could for example be realized via U2F or a passkey [46]. Alternatively p could be increased to 128.

Table 4: Server-side policy evaluation benchmark. Gates and depth express the *minimal* possible amount of multiplication gates and multiplicative depth of the circuit computing the specified policy. Computation is on 32 or 64 bits numbers when operands are promised to be in $[0; 2^{p'}]$ and using $s = 40$. Online/offline refers to the SPDZ online/offline parts. All numbers include computation and validation of MACs, uid**, share interpolation, reconstruction of user-input, but exclude threshold signatures and user communication.**

	bits	# gates	depth	t	Time (ms) Offline	Time (ms) Online	Online throughput
Same value	32	56	6	2	370	190 ± 6.7	118 ± 34
				3	952	444 ± 20	20.5 ± 5.1
	64	88	6	2	1,060	199 ± 7.1	80 ± 51
				3	2,560	466 ± 26	21 ± 7.1
Same object	32	441	13	2	2,920	253 ± 8.6	69 ± 20
				3	7,500	589 ± 32	21 ± 7.8
	64	365	9	2	4,380	262 ± 5.8	59 ± 12
				3	10,600	593 ± 30	20 ± 13
Range	32	189	10	2	1,250	201 ± 8.0	78 ± 23
				3	3,210	449 ± 26	40 ± 11
	64	381	11	2	4,570	200 ± 7.2	66 ± 23
				3	11,100	460 ± 25	36 ± 7.2
Country	32	259	14	2	1,710	3,670 ± 70	21 ± 3.6
				3	4,400	8,450 ± 380	8.2 ± 0.82
	64	295	14	2	3,540	3,700 ± 75	20 ± 3.6
				3	8,580	8,560 ± 340	8.6 ± 0.83
Pre-approved	32	1,055	16	2	6,970	17,700 ± 340	5.2 ± 0.81
				3	17,900	40,200 ± 2,200	2.1 ± 0.06
	64	1,091	16	2	13,100	17,410 ± 340	5.2 ± 0.41
				3	31,700	39,300 ± 2,000	2 ± 0.048

We implemented the signing phase of our protocol,[4] and note that the expected execution time of the registration phase is highly dependent on how the user's attributes are issued as discussed in App. C. We did not benchmark the refresh phase, as it is only supposed to be run rarely, e.g. once every few months, hence the efficiency of this is not crucial for our ab-dSSO scheme. Even so, the refresh phase is very lightweight as it does not require any expensive multiplications in MPC. However, the complexity is linearly bounded by the total amount of attributes stored in our ab-dSSO scheme, for each of which must be reshared. Such a resharing requires $O(t)$ exponentiations, in order to compute the Pedersen commitments. Furthermore, each server must also validate $O(n)$ of such sets of commitments (received from the other servers). Thus the total complexity per server is bounded by $O(|\mathcal{A}| \cdot t \cdot n)$ large group exponentiations, where $|\mathcal{A}|$ is the total amount of attributes. The goal of our implementation and benchmarking is to show what speed to expect from a realistic deployment when constructing tokens *in practice*. For this reason we based our implementation on the MPC framework FRESCO [3], which uses the SPDZ [41] protocol to realize $\mathcal{F}_{\mathsf{ABB}}$. To allow for outsourced computation, and thus to realize $\mathcal{F}_{\mathsf{reqABB}}$, we implemented the protocol of Jakobsen *et al.* [58]. Concretely we implemented our policy benchmarks off the FRESCO-Outsourcing project [4], which already implements the infrastructure for enabling outsourced MPC using FRESCO.

[4] Code available at https://github.com/aicis/fresco-outsourcing/tree/macro-bench.

	bits	Memory $t=2$	Memory $t=3$	CPU
Server (Offline)	32	747	809	51%
	64	711	729	58%
Servers (Online)	32	565	625	26%
	64	602	607	28%
Client	32	24	44	28%
	64	23	42	21%

Table 5: Maximum system utilization during signing phase. Memory is in megabytes, CPU usage is maximum of test of $t = 2, 3$.

	bits	$t = 2$	$t = 3$
Same value	32	401 ± 12	778 ± 25
	64	407 ± 8.6	800 ± 34
Same object	32	465 ± 12	921 ± 37
	64	467 ± 7.8	927 ± 32
Range	32	409 ± 9.3	775 ± 32
	64	407 ± 7.7	787 ± 29
Country	32	3, 870 ± 69	8, 780 ± 380
	64	3, 900 ± 74	8, 880 ± 338
Pre-approved	32	17, 900 ± 340	40, 500 ± 2200
	64	17, 600 ± 340	39, 500 ± 2, 000

Table 6: Client latency for MPC over field of 32 or 64 bits. Promised operands are in $[0; 2^{p'}]$. Times are in milliseconds.

For threshold signing we used an open source implementation [93] of Rabin's signature scheme [82] without the zero knowledge proofs, which has essentially the same complexity as our signature scheme (presented in Fig. 13).

Setup. We used standard price-friendly T3.xLarge AWS EC2 instances running Amazon Linux 2 and executed all our benchmarks with servers and a client residing in different datacenter locations. We observed roundtrip latencies between 8 and 30 ms. Our benchmark numbers are average time with errors being the standard deviation, based on 20 iterations, after 10 uncounted "warm-up" iterations to limit errors due to the JIT compilation and VM optimizations happening when running Java.

System footprint. We ran our experiments with the command line arguments `-Xms1024m -Xmx1024m -Xnoclassgc` to prevent overhead in dynamic heap memory allocation and garbage collection. To benchmark the maximum memory (and CPU) usage we ran tests without `-Xnoclassgc`. The result of these tests are in Tab. 5. We only include the maximum regardless of whether the test was run for $t = 2$ or 3 servers, as there is minimal difference. While FRESCO uses multi-threading to manage network and scheduling, the heavy cryptographic operations are single-threaded. Thus we expect the CPU usage to stay the same regardless of whether the code is executed on a single-core or multi-core system.

Performance. In Tab. 4 and Tab. 6 we show the performance of the signing phase of our ab-dSSO scheme running on servers, respectively a client. While Tab. 4 only considers the MPC-related benchmarks, Tab. 6 shows the full end-to-end latency experienced by a client. The online throughput expresses how much throughput a server can handle during the SPDZ online phase of our ab-dSSO scheme, in the optimal case where it processes multiple client queries concurrently, while waiting for response from the other servers. Most of the execution time is wasted on network latency, in particular if there are only two servers. We chose to run the experiments for $t = 2, 3$ as this fits the typical benchmark settings in the MPC/distributed computing space [11, 13, 21, 40] since it represents a good compromise between speed (as overhead is bound in amount of servers) and security. While we concretely used $n = t + 1$ in our benchmarks, the complexity and speed of the signing phase is completely independent of n.

While the raw throughput might not meet the requirements of providers like Google or Facebook, the throughput can be scaled by letting the server be a cluster of multiple machines, sharing the same database. Both *same value*, *same object* and *range* are already below the 1 second limit of user's flow of thought.

While we do not benchmark the network communication, for the client, it is in the order of a few kilobytes and bound by $O(t \cdot (|A| \cdot (p+s)+\lambda))$ plus the size of the message to be signed. Crucially it is *independent* of the computation required to validate the policy. For the servers the communication is bound by $O(|C| \cdot (p + s))$ where $|C|$ is the amount of multiplication gates needed to validate the policy. In general, the servers communication complexity will be bound by the realization of $\mathcal{F}_{\mathsf{ABB}}$.

Full deployment performance. While we have not implemented a full deployment of our protocol we observe that PESTO [11] uses a similar setup for distributed single-sign on. Specifically they also use Amazon AWS servers in different countries (for $t = 2$), and have implemented their protocol in Java. Their benchmarks [11, Tab. 1] show a latency of 124 ms for authentication, including client-server communication with a TLS connection. Thus using PESTO for password-based authentication (where the service provider of the PESTO SSO execution is actually the servers themselves), or purely relying or their distributed partially blind OPRF to facilitate authentication, allows us to estimate the latency for distributed client server authentication for $t = 2$.[5]. Adding distributed password authentication on top of our ab-dSSO scheme brings our solution closer to a real-world scheme, since this provides a second factor (knowledge) on top of the MACs, which *could* be considered a possessive factor if they are stored in secure hardware. See App. C.2 for more details. Thus we expect the additional overhead of a real deployment of our ab-dSSO scheme to be bounded by the policies benchmarks in Tab. 4 plus 124 ms.

REFERENCES

[1] Azure Active Directory Verifiable Credentials documentation. https://docs.microsoft.com/en-gb/azure/active-directory/verifiable-credentials/, 2021. Accessed: 2021-12-16.

[2] S. Agrawal, P. Miao, P. Mohassel, and P. Mukherjee. PASTA: PASsword-based threshold authentication. In D. Lie, M. Mannan, M. Backes, and X. Wang, editors, *ACM CCS 2018*, pages 2042–2059. ACM Press, Oct. 2018.

[3] Alexandra Institute. FRESCO - a FRamework for Efficient Secure COmputation. https://github.com/aicis/fresco.

[4] Alexandra Institute. FRESCO outsourcing. https://github.com/aicis/fresco-outsourcing.

[5] J. F. Almansa, I. Damgård, and J. B. Nielsen. Simplified threshold RSA with adaptive and proactive security. In S. Vaudenay, editor, *EUROCRYPT 2006*, volume 4004 of *LNCS*, pages 593–611. Springer, Heidelberg, May / June 2006.

[6] G. Alpár and B. Jacobs. Credential design in attribute-based identity management. 2013.

[7] E. Bach. Discrete logarithms and factoring. 1984.

[8] B. Barak, Y. Lindell, and T. Rabin. Protocol initialization for the framework of universal composability. Cryptology ePrint Archive, Report 2004/006, 2004. https://eprint.iacr.org/2004/006.

[9] R. Barbulescu, P. Gaudry, A. Joux, and E. Thomé. A heuristic quasi-polynomial algorithm for discrete logarithm in finite fields of small characteristic. In P. Q. Nguyen and E. Oswald, editors, *EUROCRYPT 2014*, volume 8441 of *LNCS*, pages 1–16. Springer, Heidelberg, May 2014.

[5] While PESTO does not support arbitrary thresholds, we observe that this is only due to the partial blindness of the OPRF, which is not needed in our situation since online brute-force throttling can be done purely based on validation of the MACs.

[10] E. Barker. NIST Special Publication 800-57 Part 1 Revision 5. Recommendation for Key Management: Part 1 – General. Technical report, National Institute for Standards and Technology - NIST, 2020. Accessed: 2023-03-08.

[11] C. Baum, T. K. Frederiksen, J. Hesse, A. Lehmann, and A. Yanai. PESTO: proactively secure distributed single sign-on, or how to trust a hacked server. *IEEE European Symposium on Security and Privacy Workshop*, 2020.

[12] C. Baum, E. Orsini, and P. Scholl. Efficient secure multiparty computation with identifiable abort. In M. Hirt and A. D. Smith, editors, *TCC 2016-B, Part I*, volume 9985 of *LNCS*, pages 461–490. Springer, Heidelberg, Oct. / Nov. 2016.

[13] P. Bogetoft, D. L. Christensen, I. Damgård, M. Geisler, T. Jakobsen, M. Krøigaard, J. D. Nielsen, J. B. Nielsen, K. Nielsen, J. Pagter, M. I. Schwartzbach, and T. Toft. Secure multiparty computation goes live. In R. Dingledine and P. Golle, editors, *FC 2009*, volume 5628 of *LNCS*, pages 325–343. Springer, Heidelberg, Feb. 2009.

[14] D. Boneh, X. Boyen, and H. Shacham. Short group signatures. In M. Franklin, editor, *CRYPTO 2004*, volume 3152 of *LNCS*, pages 41–55. Springer, Heidelberg, Aug. 2004.

[15] D. Boneh, B. Bünz, and B. Fisch. Batching techniques for accumulators with applications to IOPs and stateless blockchains. In A. Boldyreva and D. Micciancio, editors, *CRYPTO 2019, Part I*, volume 11692 of *LNCS*, pages 561–586. Springer, Heidelberg, Aug. 2019.

[16] J. Bootle, A. Cerulli, P. Chaidos, J. Groth, and C. Petit. Efficient zero-knowledge arguments for arithmetic circuits in the discrete log setting. In M. Fischlin and J.-S. Coron, editors, *EUROCRYPT 2016, Part II*, volume 9666 of *LNCS*, pages 327–357. Springer, Heidelberg, May 2016.

[17] B. Bünz, J. Bootle, D. Boneh, A. Poelstra, P. Wuille, and G. Maxwell. Bulletproofs: Short proofs for confidential transactions and more. In *2018 IEEE Symposium on Security and Privacy*, pages 315–334. IEEE Computer Society Press, May 2018.

[18] J. Camenisch, M. Drijvers, and A. Lehmann. Universally composable direct anonymous attestation. In C.-M. Cheng, K.-M. Chung, G. Persiano, and B.-Y. Yang, editors, *PKC 2016, Part II*, volume 9615 of *LNCS*, pages 234–264. Springer, Heidelberg, Mar. 2016.

[19] J. Camenisch, M. Drijvers, A. Lehmann, G. Neven, and P. Towa. Short threshold dynamic group signatures. In C. Galdi and V. Kolesnikov, editors, *SCN 20*, volume 12238 of *LNCS*, pages 401–423. Springer, Heidelberg, Sept. 2020.

[20] J. Camenisch, A. Lehmann, A. Lysyanskaya, and G. Neven. Memento: How to reconstruct your secrets from a single password in a hostile environment. In J. A. Garay and R. Gennaro, editors, *CRYPTO 2014, Part II*, volume 8617 of *LNCS*, pages 256–275. Springer, Heidelberg, Aug. 2014.

[21] J. Camenisch, A. Lehmann, and G. Neven. Optimal distributed password verification. In I. Ray, N. Li, and C. Kruegel, editors, *ACM CCS 2015*, pages 182–194. ACM Press, Oct. 2015.

[22] J. Camenisch and A. Lysyanskaya. An efficient system for non-transferable anonymous credentials with optional anonymity revocation. In B. Pfitzmann, editor, *EUROCRYPT 2001*, volume 2045 of *LNCS*, pages 93–118. Springer, Heidelberg, May 2001.

[23] J. Camenisch and A. Lysyanskaya. Dynamic accumulators and application to efficient revocation of anonymous credentials. In M. Yung, editor, *CRYPTO 2002*, volume 2442 of *LNCS*, pages 61–76. Springer, Heidelberg, Aug. 2002.

[24] J. Camenisch and A. Lysyanskaya. A signature scheme with efficient protocols. In S. Cimato, C. Galdi, and G. Persiano, editors, *SCN 02*, volume 2576 of *LNCS*, pages 268–289. Springer, Heidelberg, Sept. 2003.

[25] J. Camenisch and E. Van Herreweghen. Design and implementation of the idemix anonymous credential system. In V. Atluri, editor, *ACM CCS 2002*, pages 21–30. ACM Press, Nov. 2002.

[26] R. Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *42nd FOCS*, pages 136–145. IEEE Computer Society Press, Oct. 2001.

[27] R. Canetti. Universally composable signature, certification, and authentication. In *17th IEEE Computer Security Foundations Workshop, (CSFW-17 2004), 28-30 June 2004, Pacific Grove, CA, USA*, page 219. IEEE Computer Society, 2004.

[28] R. Canetti, R. Gennaro, S. Goldfeder, N. Makriyannis, and U. Peled. UC non-interactive, proactive, threshold ECDSA with identifiable aborts. In J. Ligatti, X. Ou, J. Katz, and G. Vigna, editors, *ACM CCS 2020*, pages 1769–1787. ACM Press, Nov. 2020.

[29] O. Catrina and S. de Hoogh. Improved primitives for secure multiparty integer computation. In J. A. Garay and R. D. Prisco, editors, *SCN 10*, volume 6280 of *LNCS*, pages 182–199. Springer, Heidelberg, Sept. 2010.

[30] M. Chase, C. Ganesh, and P. Mohassel. Efficient zero-knowledge proof of algebraic and non-algebraic statements with applications to privacy preserving credentials. In M. Robshaw and J. Katz, editors, *CRYPTO 2016, Part III*, volume 9816 of *LNCS*, pages 499–530. Springer, Heidelberg, Aug. 2016.

[31] D. Chaum. Blind signatures for untraceable payments. In D. Chaum, R. L. Rivest, and A. T. Sherman, editors, *CRYPTO'82*, pages 199–203. Plenum Press, New York, USA, 1982.

[32] D. Chaum. Security without identification: Transaction systems to make big brother obsolete. *Commun. ACM*, 28(10):1030–1044, 1985.

[33] D. Chaum and J.-H. Evertse. A secure and privacy-protecting protocol for transmitting personal information between organizations. In A. M. Odlyzko, editor, *CRYPTO'86*, volume 263 of *LNCS*, pages 118–167. Springer, Heidelberg, Aug. 1987.

[34] M. Chen, R. Cohen, J. Doerner, Y. Kondi, E. Lee, S. Rosefield, and a. shelat. Multiparty generation of an RSA modulus. In D. Micciancio and T. Ristenpart, editors, *CRYPTO 2020, Part III*, volume 12172 of *LNCS*, pages 64–93. Springer, Heidelberg, Aug. 2020.

[35] X. Chen, J. Li, X. Huang, J. Li, Y. Xiang, and D. S. Wong. Secure outsourced attribute-based signatures. *IEEE Trans. Parallel Distributed Syst.*, 25(12):3285–3294, 2014.

[36] D. Cooper, S. Santesson, S. Farrell, S. Boeyen, R. Housley, and W. Polk. Internet X.509 public key infrastructure certificate and certificate revocation list (CRL) profile. RFC 5280, May 2008.

[37] A. P. K. Dalskov, C. Orlandi, M. Keller, K. Shrishak, and H. Shulman. Securing DNSSEC keys via threshold ECDSA from generic MPC. In L. Chen, N. Li, K. Liang, and S. A. Schneider, editors, *ESORICS 2020, Part II*, volume 12309 of *LNCS*, pages 654–673. Springer, Heidelberg, Sept. 2020.

[38] I. Damgård. Payment systems and credential mechanisms with provable security against abuse by individuals. In S. Goldwasser, editor, *CRYPTO'88*, volume 403 of *LNCS*, pages 328–335. Springer, Heidelberg, Aug. 1990.

[39] I. Damgård, K. Damgård, K. Nielsen, P. S. Nordholt, and T. Toft. Confidential benchmarking based on multiparty computation. In J. Grossklags and B. Preneel, editors, *FC 2016*, volume 9603 of *LNCS*, pages 169–187. Springer, Heidelberg, Feb. 2016.

[40] I. Damgård, M. Keller, E. Larraia, V. Pastro, P. Scholl, and N. P. Smart. Practical covertly secure MPC for dishonest majority - or: Breaking the SPDZ limits. In J. Crampton, S. Jajodia, and K. Mayes, editors, *ESORICS 2013*, volume 8134 of *LNCS*, pages 1–18. Springer, Heidelberg, Sept. 2013.

[41] I. Damgård, V. Pastro, N. P. Smart, and S. Zakarias. Multiparty computation from somewhat homomorphic encryption. In R. Safavi-Naini and R. Canetti, editors, *CRYPTO 2012*, volume 7417 of *LNCS*, pages 643–662. Springer, Heidelberg, Aug. 2012.

[42] F. Denis, F. Jacobs, and C. A. Wood. RSA Blind Signatures. Internet-Draft draft-irtf-cfrg-rsa-blind-signatures-02, Internet Engineering Task Force, August 2021. Work in Progress.

[43] P. Feldman. A practical scheme for non-interactive verifiable secret sharing. In *28th FOCS*, pages 427–437. IEEE Computer Society Press, Oct. 1987.

[44] P. Feldman and S. Micali. Optimal algorithms for byzantine agreement. In *20th ACM STOC*, pages 148–161. ACM Press, May 1988.

[45] A. Fiat and A. Shamir. How to prove yourself: Practical solutions to identification and signature problems. In A. M. Odlyzko, editor, *CRYPTO'86*, volume 263 of *LNCS*, pages 186–194. Springer, Heidelberg, Aug. 1987.

[46] FIDO Alliance. How FIDO addresses a full range of use cases. Technical report, March 2022.

[47] T. K. Frederiksen. A holistic approach to enhanced security and privacy in digital health passports. In D. Reinhardt and T. Müller, editors, *16th ARES*, pages 133:1–133:10. ACM, 2021.

[48] T. K. Frederiksen, Y. Lindell, V. Osheter, and B. Pinkas. Fast distributed RSA key generation for semi-honest and malicious adversaries. In H. Shacham and A. Boldyreva, editors, *CRYPTO 2018, Part II*, volume 10992 of *LNCS*, pages 331–361. Springer, Heidelberg, Aug. 2018.

[49] R. Granger and A. Joux. Computing discrete logarithms. *Computational Cryptography: Algorithmic Aspects of Cryptology*, 2021. See https://ia.cr/2021/1140 and www.cambridge.org/9781108795937.

[50] J. Groth and A. Sahai. Efficient non-interactive proof systems for bilinear groups. In N. P. Smart, editor, *EUROCRYPT 2008*, volume 4965 of *LNCS*, pages 415–432. Springer, Heidelberg, Apr. 2008.

[51] C. Guo, J. Lin, Q. Cai, F. Li, Q. Wang, J. Jing, B. Zhao, and W. Wang. UPPRESSO: untraceable and unlinkable privacy-preserving single sign-on services. *CoRR*, abs/2110.10396, 2021.

[52] Y. Harchol, I. Abraham, and B. Pinkas. Distributed SSH key management with proactive RSA threshold signatures. In B. Preneel and F. Vercauteren, editors, *ACNS 18*, volume 10892 of *LNCS*, pages 22–43. Springer, Heidelberg, July 2018.

[53] D. Hardt. The OAuth 2.0 authorization framework. RFC 6749, October 2012.

[54] J. Håstad. Knuth prize lecture: On the difficulty of approximating Boolean max-CSPs. In M. Thorup, editor, *59th FOCS*, page 602. IEEE Computer Society Press, Oct. 2018.

[55] J. Herranz, F. Laguillaumie, B. Libert, and C. Ràfols. Short attribute-based signatures for threshold predicates. In O. Dunkelman, editor, *CT-RSA 2012*, volume 7178 of *LNCS*, pages 51–67. Springer, Heidelberg, Feb. / Mar. 2012.

[56] A. Herzberg, S. Jarecki, H. Krawczyk, and M. Yung. Proactive secret sharing or: How to cope with perpetual leakage. In D. Coppersmith, editor, *CRYPTO'95*, volume 963 of *LNCS*, pages 339–352. Springer, Heidelberg, Aug. 1995.

[57] ID4me Login Consortium. ID4me. https://id4me.org.

[58] T. P. Jakobsen, J. B. Nielsen, and C. Orlandi. A framework for outsourcing of secure computation. In G. Ahn, A. Oprea, and R. Safavi-Naini, editors, *CCSW*, pages 81–92. ACM, 2014.

[59] M. Jones, J. Bradley, and N. Sakimura. JSON web token (JWT). RFC 7519, May 2015.

[60] N. Kaaniche and M. Laurent. Attribute-based signatures for supporting anonymous certification. In I. G. Askoxylakis, S. Ioannidis, S. K. Katsikas, and C. A.

Meadows, editors, *ESORICS 2016, Part I*, volume 9878 of *LNCS*, pages 279–300. Springer, Heidelberg, Sept. 2016.
[61] S. Kamara, P. Mohassel, and M. Raykova. Outsourcing multi-party computation. Cryptology ePrint Archive, Report 2011/272, 2011. https://eprint.iacr.org/2011/272.
[62] S. Kamara, P. Mohassel, and B. Riva. Salus: a system for server-aided secure function evaluation. In T. Yu, G. Danezis, and V. D. Gligor, editors, *ACM CCS 2012*, pages 797–808. ACM Press, Oct. 2012.
[63] M. Keller, E. Orsini, and P. Scholl. MASCOT: Faster malicious arithmetic secure computation with oblivious transfer. In E. R. Weippl, S. Katzenbeisser, C. Kruegel, A. C. Myers, and S. Halevi, editors, *ACM CCS 2016*, pages 830–842. ACM Press, Oct. 2016.
[64] M. Keller, V. Pastro, and D. Rotaru. Overdrive: Making SPDZ great again. In J. B. Nielsen and V. Rijmen, editors, *EUROCRYPT 2018, Part III*, volume 10822 of *LNCS*, pages 158–189. Springer, Heidelberg, Apr. / May 2018.
[65] D. Khovratovich and J. Lee. Sovrin: digital identities in the blockchain era. https://sovrin.org/wp-content/uploads/AnonCred-RWC.pdf.
[66] N. Klingenstein. Attribute aggregation and federated identity. In *2007 International Symposium on Applications and the Internet - Workshops (SAINT 2007 Workshops), 15-19 January 2007, Hiroshima, Japan*, page 26. IEEE Computer Society, 2007.
[67] J. Li and K. Kim. Attribute-based ring signatures. Cryptology ePrint Archive, Report 2008/394, 2008. https://eprint.iacr.org/2008/394.
[68] T. Looker and O. Steele. BBS+ signatures 2020. https://identity.foundation/bbs-signature/draft-looker-cfrg-bbs-signatures.html.
[69] A. Lysyanskaya, R. L. Rivest, A. Sahai, and S. Wolf. Pseudonym systems. In H. M. Heys and C. M. Adams, editors, *SAC 1999*, volume 1758 of *LNCS*, pages 184–199. Springer, Heidelberg, Aug. 1999.
[70] H. Maji, M. Prabhakaran, and M. Rosulek. Attribute-based signatures: Achieving attribute-privacy and collusion-resistance. Cryptology ePrint Archive, Report 2008/328, 2008. https://eprint.iacr.org/2008/328.
[71] H. K. Maji, M. Prabhakaran, and M. Rosulek. Attribute-based signatures. In A. Kiayias, editor, *CT-RSA 2011*, volume 6558 of *LNCS*, pages 376–392. Springer, Heidelberg, Feb. 2011.
[72] D. Maram, H. Malvai, F. Zhang, N. Jean-Louis, A. Frolov, T. Kell, T. Lobban, C. Moy, A. Juels, and A. Miller. CanDID: Can-do decentralized identity with legacy compatibility, sybil-resistance, and accountability. In *2021 IEEE Symposium on Security and Privacy*, pages 1348–1366. IEEE Computer Society Press, May 2021.
[73] K. Moriarty, B. Kaliski, J. Jonsson, and A. Rusch. PKCS #1: RSA cryptography specifications version 2.2. RFC 8017, November 2016.
[74] N. Naik and P. Jenkins. uPort open-source identity management system: An assessment of Self-Sovereign Identity and user-centric data platform built on blockchain. In *IEEE International Symposium on Systems Engineering, ISSE 2020*, pages 1–7. IEEE.
[75] NIST. Submission requirements and evaluation criteria for the lightweight cryptography standardization process. Technical report, National Institute for Standards and Technology - NIST, 2018. Accessed: 2023-03-07.
[76] T. Okamoto and K. Takashima. Decentralized attribute-based signatures. In K. Kurosawa and G. Hanaoka, editors, *PKC 2013*, volume 7778 of *LNCS*, pages 125–142. Springer, Heidelberg, Feb. / Mar. 2013.
[77] OneLogin. OneLogin Trusted Experience Platform. https://www.onelogin.com/pages/identity-as-a-service-idaas.
[78] Organization for the Advancement of Structured Information Standards. Security assertion markup language (saml) v2.0, 2005.
[79] C. Paquin. U-Prove Technology Overview V1.1. Tech report, April 2013.
[80] T. P. Pedersen. Non-interactive and information-theoretic secure verifiable secret sharing. In J. Feigenbaum, editor, *CRYPTO'91*, volume 576 of *LNCS*, pages 129–140. Springer, Heidelberg, Aug. 1992.
[81] D. Pointcheval and O. Sanders. Reassessing security of randomizable signatures. In N. P. Smart, editor, *CT-RSA 2018*, volume 10808 of *LNCS*, pages 319–338. Springer, Heidelberg, Apr. 2018.
[82] T. Rabin. A simplified approach to threshold and proactive RSA. In H. Krawczyk, editor, *CRYPTO'98*, volume 1462 of *LNCS*, pages 89–104. Springer, Heidelberg, Aug. 1998.
[83] N. Sakimura, J. Bradley, M. Jones, B. de Medeiros, and C. Mortimore. OpenID connect core 1.0 incorporating errata set 1. https://openid.net/specs/openid-connect-core-1_0.html, 2014. Accessed: 2020-03-01.
[84] M. Schanzenbach, T. Kilian, J. Schütte, and C. Banse. Zklaims: Privacy-preserving attribute-based credentials using non-interactive zero-knowledge techniques. In M. S. Obaidat and P. Samarati, editors, *Proceedings of the 16th International Joint Conference on e-Business and Telecommunications, ICETE 2019 - Volume 2: SECRYPT, Prague, Czech Republic, July 26-28, 2019*, pages 325–332. SciTePress, 2019.
[85] S. F. Shahandashti and R. Safavi-Naini. Threshold attribute-based signatures and their application to anonymous credential systems. In B. Preneel, editor, *AFRICACRYPT 09*, volume 5580 of *LNCS*, pages 198–216. Springer, Heidelberg, June 2009.
[86] A. Shamir. How to share a secret. *Communications of the Association for Computing Machinery*, 22(11):612–613, Nov. 1979.
[87] G. Shanqing and Z. Yingpei. Attribute-based signature scheme. In *2008 International Conference on Information Security and Assurance (isa 2008)*, pages 509–511, 2008.
[88] M. Shirvanian, S. Jarecki, N. Saxena, and N. Nathan. Two-factor authentication resilient to server compromise using mix-bandwidth devices. In *NDSS 2014*. The Internet Society, Feb. 2014.
[89] N. P. Smart and Y. T. Alaoui. Distributing any elliptic curve based protocol. In M. Albrecht, editor, *Cryptography and Coding - IMACC*, volume 11929 of *LNCS*, pages 342–366. Springer, 2019.
[90] M. Sporny, D. Longley, M. Sabadello, D. Reed, O. Steele, and C. Allen. Decentralized Identifiers (DIDs) v1.0 Core architecture, data model, and representations. https://www.w3.org/TR/did-core/, 2021. Accessed: 2021-12-16.
[91] J. Sun, Y. Su, J. Qin, J. Hu, and J. Ma. Outsourced decentralized multi-authority attribute based signature and its application in iot. *IEEE Trans. Cloud Comput.*, 9(3):1195–1209, 2021.
[92] Verimi. Verimi ID Wallet. https://verimi.de/en.
[93] S. Weis. Threshsig. https://github.com/sweis/threshsig. Accessed: 2022-02-27.
[94] X. Yi and K.-Y. Lam. A new blind ECDSA scheme for bitcoin transaction anonymity. In S. D. Galbraith, G. Russello, W. Susilo, D. Gollmann, E. Kirda, and Z. Liang, editors, *ASIACCS 19*, pages 613–620. ACM Press, July 2019.
[95] F. Zhang, E. Cecchetti, K. Croman, A. Juels, and E. Shi. Town crier: An authenticated data feed for smart contracts. In E. R. Weippl, S. Katzenbeisser, C. Kruegel, A. C. Myers, and S. Halevi, editors, *ACM CCS 2016*, pages 270–282. ACM Press, Oct. 2016.
[96] F. Zhang, D. Maram, H. Malvai, S. Goldfeder, and A. Juels. DECO: Liberating web data using decentralized oracles for TLS. In J. Ligatti, X. Ou, J. Katz, and G. Vigna, editors, *ACM CCS 2020*, pages 1919–1938. ACM Press, Nov. 2020.

A COMPARISON OF AB-DSSO TO OTHER CRYPTOGRAPHIC PRIMITIVES

We discuss how ab-dSSO relates to concepts such as attribute-based signatures and attribute-based credentials.

Attribute-based credentials. An attribute-based credential (ABC) system (Fig. 8, right) lets a user obtain a credential cr_A over attributes A from a trusted credential issuer. To demonstrate statements P about their identity, the user can—without further interacting with the credential issuer—then derive presentation tokens π_P from their credential. While the idea of credentials dates back to Chaum [32], the first fully-secure ABC was proposed by Lysyanskaya *et al.* [69], with subsequent improvements by Camenisch and Lysyanskaya [22, 23], which we refer to as CL-ABCs. The general idea is that a credential consists of signed attributes, and presentations are non-interactive zero-knowledge proofs about the contents of the signatures. CL-ABCs, which work with Camenisch-Lysyanskaya signatures [22], have been turned into commercial products such as Microsoft's U-Prove [79] and IBM's Idemix [25], which can be implemented on SmartCards [6]. Recently, efforts were taken to build an ABC called BBS+ [68] from Boneh-Boyen-Shacham signatures [14] in combination with zero-knowledge proofs. It is currently not known how to efficiently revoke BBS+ credentials or how to demonstrate predicates over attributes.

A general drawback of ABCs is their reliance on credential issuers that must be trusted with not impersonating users. Further, ABCs deploy zero-knowledge proofs that must be computed by the user and are computationally heavy. It is also not trivial to make presentation tokens on policies that combine credentials from different credential issuers. Lastly, token presentations are zero-knowledge proofs that do not adhere to any standardized format that service providers can process.

Attribute-based signatures. Attribute-based signatures (ABS) [70] are digital signature schemes with a trusted AA who can generate

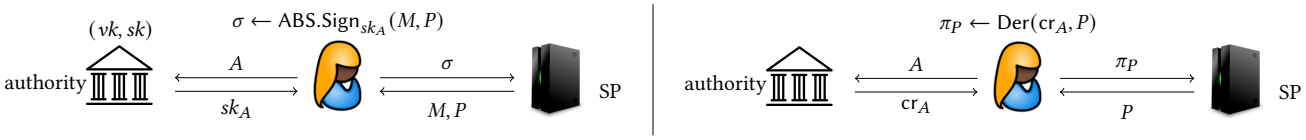

Figure 8: Cryptographic primitives for attribute-based access token generation, part I. Left: Attribute-Based Signatures (ABS), where signatures verify with respect to vk. Right: P-ABCs. Both options require costly computation on the user side, and the produced signatures/proofs are not of standardized format.

signing keys sk_A w.r.t. an attribute set A. Such a signing key can be used to sign a message-policy pair (M, P) iff $P(A) = 1$, i.e., the attribute set fulfills the policy. We illustrate ABS and their close relation to P-ABCs in Fig. 8 (left). To reduce the load on the signer, Chen *et al.* [35] study outsourced ABS, where an untrusted server assists the signer with the signature computation but cannot sign on its own, and Maji *et al.* [71] study multi-authority ABS, where secret keys from different trusted attribute authorities can be combined. This concept was later also realized in the outsourced setting [91]. One can naturally extend ABS to *distributed* ABS, as illustrated in Fig. 9 (left). While, intuitively, this primitive seems closely related to our notion of ab-dSSO, and it seems instructive to precisely understand the differences, we caution that there does not seem to exist any formal definition of distributed ABS in the literature yet. In a distributed ABS scheme, a set of identity providers assist the user in computing the attribute-based signature. An attacker must corrupt all IdPs to recover the signing key sk_A, and otherwise, the user cannot be impersonated by any subset of corrupted IdPs. To make the IdPs compute with the distributed sk_A, a user must authenticate themselves as the one having outsourced sk_A to the distributed IdP.

The main drawback of ABS is that, until today, no scheme is known that produces signatures of any standardized format. This and the strong trust model (the AA can impersonate the user) are the main reasons why ABS has not seen adoption in practice. In this work we thus change distributed ABS from Fig. 9 (left) in one crucial aspect: As illustrated in Fig. 9 (right), instead of letting the distributed IdP compute an ABS signature in a potentially non-standardized format, we decouple policy verification and signing. Namely, the IdPs first jointly check whether the policy is satisfied with the help of what can be seen as a shared (hiding and binding) commitment com_A on the user's attributes. If that check passes, the servers deploy a distributed signature scheme to sign the pair (M, P). This approach has two benefits: First, the hiding commitment to user attributes keeps attributes private, while ABS secret key sk_A unavoidably leaks information about A. Second, the output format can now be a standardized signature such as ECDSA. This comes at the cost of weaker guarantees in case all IdPs are corrupted: In a distributed ABS scheme such IdPs can compute only signatures for policies satisfied by A, while they can compute arbitrary signatures in our ab-dSSO scheme.

B ADDITIONAL PROTOCOL DESCRIPTION

We formalize our ab-dSSO protocol in Figures 5–7 in Sect. 4, with a high level blueprint in Fig. 4. Here we outline its steps and explain why they are needed.

Any protocol instance has to start with a single call to the *setup* phase. This phase uses the functionality for proactively secure threshold signatures ($\mathcal{F}_{\mathsf{psThrSig}}$, Fig. 18) to generate a public verification key vk, and n t-out-of-n-secret-shares of the corresponding signing key. Each share is sent to exactly one of the servers. The servers then use the outsourced MPC functionality $\mathcal{F}_{\mathsf{reqABB}}$ to sample random coefficients of a degree $t-1$ polynomial, $f_\Delta(\cdot)$. In MPC, the servers then evaluate this polynomial on each n points to compute a Shamir secret sharing of an unknown, yet random value Δ, which we will call the global MAC key. The point $f_\Delta(i)$ is opened towards each server $i \in [n]$ through the use of the macro Pedersen$^+$.Share. This macro ensures that each server's share of Δ is reshared with all other servers through backup shares, to allow for recovery during the *refresh* phase in case some servers get corrupted, drop dead, or lose their data.

Next, the *registration* phase is executed between a user U and all the n servers in order to add a set of attributes, A, to the user's account (under uid) in our ab-dSSO scheme. The user inputs to the outsourced MPC functionality $\mathcal{F}_{\mathsf{reqABB}}$: their unique user ID (uid), their attributes (A), their public attribute types $\mathtt{type}_{j\,j\in[m]}$, along with a proof ($\pi$) that these attributes have been certified by an attribute authority with a public key vk_{CA} that the servers trust. This certificate is verified by executing the verification function CAVfy in MPC. While this is expensive, the idea of our ab-dSSO scheme is to carry out this verification only once during registration, and then substitute this step in the online signing phase by something that is orders of magnitude more efficient, namely by a MAC verification. Proceeding with the description of the registration phase of our ab-dSSO scheme, the servers restore the global MAC key Δ, by inputting their Shamir shares of Δ into $\mathcal{F}_{\mathsf{reqABB}}$, where the private value $[\Delta]$ is computed using Lagrange interpolation. Then, in a similar manner to how Δ was constructed, compute an attribute-specific MAC key $\beta_{\mathsf{uid},j}$ for each attribute of a_j of the user with ID uid. That is, a value $\beta_{\mathsf{uid},j}$ is sampled through picking coefficients of a random $t-1$ degree polynomials, doing interpolation for n points, and then sharing each point using the macro Pedersen$^+$.Share to allow for recovery and refreshing. At the same time the servers also compute a MAC $v_{\mathsf{uid},j} = a_j \cdot \Delta + \beta_{\mathsf{uid},j}$ for each attribute a_j for user uid. The MAC $v_{\mathsf{uid},j}$ is then opened to the user, whereas each server learns a Shamir share of $\beta_{\mathsf{uid},j}$ (along with backup shares and validation information).

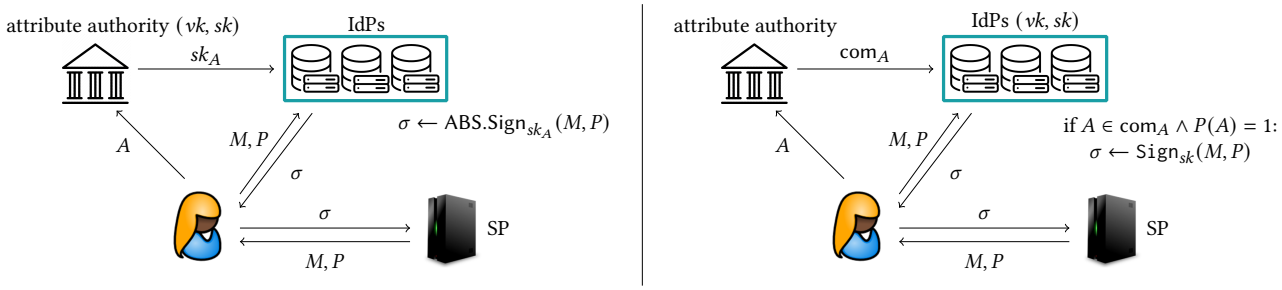

Figure 9: Cryptographic primitives for attribute-based access token generation, part II. Left: distributed Attribute-Based Signatures (dABS). Right: attribute-based distributed SSO (ab-dSSO). Both options are lightweight on the user and produce signatures (tokens) that verify under vk. The IdP computations are distributed and, unless all IdPs are corrupt, they do not reveal any information about A beyond $P(A)$. Both primitives require the user to authenticate to the IdPs, which we omit from the picture for simplicity.

After the *setup* and *registration* phases have been completed for a user U with uid, the user can collaborate with t servers in order to get a signed token on a message M and a policy validating a predicate P on a subset of U's attributes A. This is done by the user inputting each relevant attribute (a_j), along with the corresponding attribute-specific MAC ($v_{\mathsf{uid},j}$) into an outsourced MPC instance of $\mathcal{F}_{\mathsf{reqABB}}$. The servers then input their shares of the global MAC key Δ, along with the relevant attribute-specific MAC keys ($\beta_{\mathsf{uid},j}$), into $\mathcal{F}_{\mathsf{reqABB}}$. In MPC the MAC keys $[\Delta]$, $[\beta_{\mathsf{uid},j}]$ are computed from the servers' shares using Lagrange interpolation, and the MAC is validated by ensuring that $v_{\mathsf{uid},j} = a_j \cdot \Delta + \beta_{\mathsf{uid},j}$. The policy on the attributes is then also validated in MPC. If both validations pass, the servers use $\mathcal{F}_{\mathsf{psThrSig}}$ to construct a threshold signature on P and the message M. Each partial signature is sent to the user, who reconstructs the actual signature σ. Since the $t-1$ partial signatures leak nothing about the actual signature σ (which is used as a token against a relying party) a corrupt server cannot hijack an honest user's token.

Revocation of the a user's attributes is done by the servers deleting their shares of the attribute-specific MAC key for the attribute in question. Thus, this attribute can never be verified again. This step is very efficient thanks to the account-based model of our ab-dSSO scheme.

Verification of a signed token on message M and policy P is carried out by calling the verification method of $\mathcal{F}_{\mathsf{psThrSig}}$. By design, $\mathcal{F}_{\mathsf{psThrSig}}$ requires this verification method to be an efficiently computable algorithm (i.e., not a distributed or interactive procedure). In our instantiation of $\mathcal{F}_{\mathsf{psThrSig}}$, this will be the standard RSA signature verification algorithm.

If a server has been corrupted, but the adversary has been removed again, then it is needed to execute the *refresh* phase to refresh the cryptographic shares the servers hold, in order to devalidate the shares the adversary has extracted. If the adversary has deleted or modified a servers shares, then the refresh phase also ensures that the damaged server can restore correct shares again. The signing key shares are refreshed using $\mathcal{F}_{\mathsf{psThrSig}}$, and the MAC shares are refreshed using the macro Pedersen$^+$ in Fig. 10. Concretely, Pedersen$^+$.Recover is first used to recover the newly un-corrupted servers' "old", yet correct shares, in case the adversary has deleted or modified these. Afterwards, each server uses Pedersen$^+$.Share to perform a 0-sharing, and ask all other servers to add the resulting shares of 0 to their backup shares and validation information. This is done in order to rerandomize the backup shares and validation information the other servers hold on their shares. Finally the servers validate that the backup shares are still correct and consistent after the recovery and refresh steps, using Pedersen$^+$.Verify.

C DEPLOYMENT CONSIDERATIONS

C.1 Attribute Authorities (AA)

We first outline different approaches from the literature to implement the attribute authorities and corresponding oblivious certificate verification assumed in this work. This requires a method to convince the servers about the validity of a certified attribute, but without disclosing it.

Using DECO. The work DECO by Zhang *et al.* [96] shows how to leverage the authenticity of a TLS connection between an attribute authority and a user in a way that allows the user to selectively disclose data signed by the attribute authority, to a third party. In our situation the third party will be an IdP server. This allows the server to only learn relevant and certified attributes about the user. In the LAN setting this can be achieved with an overhead of less than 2 seconds, which can be considered reasonable as it is only needed during setup.

Furthermore, [96] also show how to construct zero-knowledge proofs on the data signed in the TLS connection, which allows the user to prove anything in relation to the data. We can leverage this in our scheme by executing DECO n times in parallel, once for each IdP server, and proceeding as follows:

(1) Let A denote the attributes certified by the attribute authority which the user wishes to prove some statement about towards a server.
(2) The user then constructs a commitment to A, which we denote by C, and define R to be the proof statement mentioned above, augmented with the verification that C is a commitment to A.

Global parameters: A constant c and static values $g, h \in \mathbb{F}_{p^c}$ s.t. no-one knows $\log_g(h)$.

PEDERSEN$^+$.MPC-SHARE: Executed between all parties in a set $\Sigma := \{S_1, \ldots, S_n\}$, each with input $[s_i]$, as follows.

- The servers in Σ use $\mathcal{F}_{\text{reqABB}}$ as follows:
  - Generate $[a_{i,1}], \ldots, [a_{i,t-1}] \xleftarrow{\text{rnd}} \mathcal{F}_{\text{ABB}}^{ssid}$. Let $[a_{i,0}] := [s_i]$ and $P_{[s_i]}(X) := [a_{i,0}] + [a_{i,1}]X + \cdots + [a_{i,t-1}]X^{t-1} \mod p$.
  - For each $j \in [n]$ compute $[s_{i,j}] = P_{[s_i]}(j)$ and send $([s_{i,j}], S_j) \xrightarrow{\text{open}} \mathcal{F}_{\text{reqABB}}^{ssid}$ for $j \in [n]$ and $([a_{i,k}], S_i) \xrightarrow{\text{open}} \mathcal{F}_{\text{reqABB}}^{ssid}$ for each $k \in [t-1]$. *//Server S_i learns the coefficients of $P_{[s_i]}$ and all servers learns a point on that polynomial.*
- Server S_i samples $r_{i,k} \leftarrow_R \mathbb{F}_{p^c}$ for $k \in [t-1] \cup \{0\}$ and let $t_i := r_{i,0}$ and denote $P_{t_i}(X) := r_{i,0} + r_{i,1}X + \cdots + r_{i,t-1}X^{t-1}$.
- S_i computes $A_{i,k} \leftarrow g^{a_{i,k}} \cdot h^{r_{i,k}} \in \mathbb{F}_{p^c}$ [a]. and broadcasts $(ssid, A_{i,0}, \ldots, A_{i,t-1})$ to all servers and sends $(ssid, t_{i,j} \leftarrow P_{t_i}(j))$ to server S_j for $j \in [n] \setminus \{i\}$.
- Each server $S_j \in \Sigma$ executes VERIFY on $(s_{i,j}, t_{i,j}, A_{i,0}, \ldots, A_{i,t-1})$ and aborts if reject is returned.
- Each S_i **returns** $(ssid, \{\texttt{priv} := (s_{i,1}, t_{i,1}), \ldots, (s_{i,n}, t_{i,n})\}, \{\texttt{pub} := A_{i,0}, \ldots, A_{i,t-1}\})$.

PEDERSEN$^+$.SHARE: Executed by each party $S_i \in \Sigma$ on input s_i, t_i, with $s_i \in \mathbb{F}_p$ and $t_i \in \mathbb{F}_{p^c}$. Server S_i proceeds as follows.

- Generate $a_{i,1}, \ldots, a_{i,t-1} \leftarrow_R \mathbb{F}_p$. Let $a_{i,0} := s_i$ and $P_{s_i}(X) := a_{i,0} + a_{i,1}X + \cdots + a_{i,t-1}X^{t-1}$.
- Sample $r_{i,1}, \ldots, r_{i,t-1} \leftarrow_R \mathbb{F}_{p^c}$ and let $r_{i,0} := t_i$ and denote $P_{t_i}(X) := r_{i,0} + r_{i,1}X + \cdots + r_{i,t-1}X^{t-1}$.
- Compute $A_{i,k} \leftarrow g^{a_{i,k}} \cdot h^{r_{i,k}} \in \mathbb{F}_{p^c}$ and broadcast $(ssid, A_{i,0}, \ldots, A_{i,t-1})$ to all servers and sends $(ssid, s_{i,j} \leftarrow P_{s_i}(j), t_{i,j} \leftarrow P_{t_i}(j))$ to server S_j for $j \in [n] \setminus \{i\}$.
- **Return** $(ssid, \{\texttt{priv} := (s_{i,1}, t_{i,1}), \ldots, (s_{i,n}, t_{i,n})\}, \{\texttt{pub} := A_{i,0}, \ldots, A_{i,t-1}\})$.

PEDERSEN$^+$.VERIFY: Executed by any party $S_i \in \Sigma$ on input $(s_{j,i}, t_{j,i}, A_{j,0}, \ldots, A_{j,t-1})$. Accept if $g^{s_{j,i}} \cdot h^{t_{j,i}} = \prod_{k=0}^{t-1}(A_{j,k})^{i^k}$ and otherwise reject.

PEDERSEN$^+$.RECONSTRUCT: Executed by any party $S_i \in \Sigma$ on input $(T, (s_{j,i}, t_{j,i})_{j:\, S_j \in T})$, with $T \subseteq \Sigma$ of size t as follows.

- Compute $\omega_{T,j} := \prod_{k \neq j:\, S_k \in T} k/(k-j) \mod p$.
- Compute $s_i \leftarrow \sum_{j:\, S_j \in T} s_{j,i}\omega_{T,j} \bmod p$ and $t_i \leftarrow \sum_{j:\, S_j \in T} t_{j,i}\omega_{T,j} \in \mathbb{F}_{p^c}$ and **return** (s_i, t_i).

PEDERSEN$^+$.RECOVER: Executed by each server in Σ on input $(s_i, t_i, \{s_{j,i}, t_{j,i}, A_{j,0}, \ldots, A_{j,t-1}\}_{j=1}^n)$.

- Broadcast $(A_{j,0}, \ldots, A_{j,t-1})_{j=1}^n$.
- Upon receiving such tuples from all the other servers in Σ, if there exists a tuple that was broadcast by at least t servers, overwrite $(A_{j,0}, \ldots, A_{j,t-1})_{j=1}^n$ with the values from that tuple. Abort if no such tuple exists (one necessarily does in case $n - t \geq t$).
- If $g^{s_i} \cdot h^{t_i} \neq A_{i,0}$, i.e. if s_i is inconsistent with value committed in $A_{i,0}$, then do as follows:
  - Broadcast to all servers in Σ a request to privately send the back-up shares for S_i they hold, i.e. $s_{i,j}, t_{i,j}$ for server $S_j \in \Sigma$.
  - Overwrite the values $s_{i,j}, t_{i,j}$ with those received from server in $S_j \in \Sigma$. Then execute VERIFY on $(s_{i,j}, t_{i,j}, A_{i,0}, \ldots, A_{i,t-1}$ for each $j \in [n]$ and add $(s_{i,j}, t_{i,j})$ to an initially empty set T for each pair $(s_{i,j}, t_{i,j})$ where VERIFY returns accept. *//If $n - t \geq t$, there must be at least t correct shares.*
  - Overwrite s_i, t_i with the reconstructed values returned from PEDERSEN$^+$.RECONSTRUCT on input $(s_{j,i}, t_{j,i})_{j:\, S_j \in T}$. any subset of t correct shares. I.e. $(s_i, t_i) :=$ PEDERSEN$^+$.RECONSTRUCT$(\texttt{Reconstruct}, sid, T, (s_{j,i}, t_{j,i})_{j:\, S_j \in T})$
- **Return** $(s_i, t_i, \{s_{j,i}, t_{j,i}, A_{j,0}, \ldots, A_{j,t-1}\}_{j=1}^n)$.

[a] We implicitly map an element $x \in \mathbb{F}_p$ to an element in $\mathbb{F}_{p^c}$ by interpreting x as the constant term of the degree $c-1$ polynomial $\mathbb{F}_p[X]$, which represent an element in $\mathbb{F}_{p^c}$, while keeping all non-constant terms as 0.

Figure 10: Macro PEDERSEN$^+$ for Pedersen verifiable secret sharing based on values stored in MPC over an extension field $\mathbb{F}_{p^c}$. The Macro is used to refresh the MAC shares of our ab-dSSO scheme, i.e., it is called by Fig. 7.

(3) The user defines π to contain C and CAVfy to validate that C is to a commitment to $a_1, \ldots, a_m \in A$ given as input to $\mathcal{F}_{\text{reqABB}}$.

Through this approach we obtain a linking of the validity of the signature from TLS into the MPC protocol, while the attribute authority is oblivious to the fact that this protocol gets carried out.

If the commitments are Pedersen commitments [80] over elliptic curves, then the approach by Smart and Alaoui [89] or Dalskov *et al.* [37] for doing efficient MPC over elliptic curves could be used to efficiently compute and validate the commitment in MPC.

A similar approach to getting attribute certificates from an attribute authority, through an authenticated TLS channel, can be taken with Town Crier, although that scheme relies on secure hardware. In particular SGX, which has already been shown to be easily compromisable [95].

When it comes to revocation it can be done using a time-out approach. That is, attributes issued through a DECO approach have a certain life-time associated and when it expires the IdP servers will remove the associated attributes and request the user to get them reissued.

Using bearer tokens or X.509. In the setting of bearer tokens [78, 83] or X.509 [36] certificates, the attributes are certified through a signature by an authority. The servers must remain oblivious to the content of the token, and hence validate its signature and any other relevant information, such as expiration, in MPC. That is, the function CAVfy must validate the signature against the key vk_{CA} and other aspects, such as expiration, against the current time. While validating a signature in MPC on hidden input can be expensive, efficiency increases significantly if a few simple assumptions hold: assume a hash-and-sign signature scheme is used (e.g. (EC)DSA or RSA) and that the token contains a high amount of entropy, either through a serial number, or a user-defined field that can be populated with randomness. In this situation, the hash digest can be opened as a public value. This means that the signature can be verified outside of MPC.

In relation to OAuth [53] two types of tokens are issued. First, a short-lived "access token", which contains data that the servers can use to get a "refresh token" by the attribute authority. The refresh token can then be used long-term by the recipient to get new access tokens issued by the authority, without the interaction of the user. If the user gets their privileges revoked, or they don't trust the server any more, then the attribute authority will stop accepting the refresh token. If the citizen's attributes are *only* contained in the initial access token, then this flow still allows a server to use the refresh token to get new access tokens to ensure that the citizen's attributes are still valid. Once it stops being valid they will revoke the associated attributes or the user. This approach however, requires that any access tokens the server requests, using the refresh token, does not contain the citizen's attributes. If the tokens are instead issued more like an x509 certificate, then the IdP servers can regularly check the certificate revocation list, supposed to be published by the attribute authority. This list will contain fingerprints of revoked certificates and the IdP servers can then remove attributes associated with the fingerprint of the revoked certificate.

Using a decentralized attribute authority. While our main focus in this work has been to construct a standards-compliant identity management system, this necessary means that the attribute authorities are not distributed. Thus they provide single points of failures (on the attributes they certify respectively). However, if attribute authorities are willing to use a distributed approach for issuance then our entire system could rely entirely on distributed security. Consider for example a distributed attribute authority that would construct an elliptic curve Pedersen commitment to the attributes and sign this with a distributed signature scheme. If the authority sends the opening information to the user, then this would allow using public verification of the signature and then to use the approach of Dalskov *et al.* [37]. to reconstruct, and hence verify, the Pedersen commitment efficiently in MPC. Concretely this would mean that CAVfy gets split into a public and private part, where the public part validates the signature by the distributed attribute authority on the commitment. For the private part we must assume the user use (INPUT, ...) to private input the opening information of the commitment, then $\mathcal{F}_{\mathsf{reqABB}}$ recomputes the commitment and opens this to all the servers. The server can then validate that this was in fact the commitment signed by the distributed attribute authority.

Using a pragmatic approach to privacy. While the approaches above are all feasible they are relatively expensive. Depending on the model of security assumed, it might be permissible to have the servers learn the attributes in question, either through DECO without a zero-knowledge proof and MPC validation, or through a bearer token or X.509-style certificate. While on the surface this seem to go directly against the privacy goals we have tried to achieve, we do note that is actually not the case if the entities running the servers are actually trusted and the motivation for distributing the server is to ensure protection against hacker or internal attacks, rather than a deliberate malicious legal entity. In this situation it can be deemed acceptable that the server learn the users attribute during registration as they will not be stored in persistent storage. Furthermore even if a server is compromised during registration it is still prevent malicious issuance of tokens as *all* servers must agree on such issuance. In that situation the registration can be realized efficiently by having π contain the signed token/certificate by the attribute authority and the function CAVfy does not execute *any* private function but instead only validates the signed token/certificate in plain, against the attributes $a_1, \ldots, a_m$ using the approach discussed in App. D.1.

C.2 User deployment

Our ab-dSSO scheme implicitly assumes the user holds a secure state, concretely consisting of their attributes and the MACs on these, this is in contrast to PASTA [2] or PESTO [11] which only require the user to possess an easy-to-remember password. SSO protocols are often used in a web-based context, where no local state is necessarily stored. In such a context a user would traditionally authenticate using a password (and perhaps a second factor) towards the IdP to prove they own the uid they claim, and then ask the IdP to construct a token for them. Thus PASTA [2] and PESTO [11] follow this traditional paradigm. However, in practice authenticating purely based on a password might not provide sufficient security against credential stuffing attacks, hence a secondary authentication factor might need to be supplied as well. Instead of restricting our formal security model to a password model, we only consider the core cryptographic material needed and leave it as a deployment, or a protocol-composition option, how to manage or augment this.

However we note, that to deploy our ab-dSSO scheme in such a password-based context, the user could use password-authenticated threshold secret sharing to store their MAC values $v_{\mathsf{uid},j}$ in a threshold or distributed manner [20, 52]. Alternatively, if authentication is allowed to happen via a phone app, then secure hardware, such as the SE/TEE, could be leveraged to store the MAC values, and optionally additional authentication keys. Finally, an approach combining password-authenticated threshold secret sharing along with an app leveraging secure hardware, could be deployed for optimal security [47].

If we assume the user has (somewhat) secure storage available, then the password security can also be augmented with a possessive element of multi factor authentication by letting them be one of the authentication servers [88] or by co-authenticating the request using secure hardware [47].

C.3 Unlinkability and untraceability

The tokens issued in real-world SSO systems [53, 78, 83] contain, at the minimum, a unique token ID, unique identifier of the user, unique identifier of the service provider, and a validity time. These together ensure that the token can work as a bearer token for authentication at a service provider. The unique user ID allows the notion of a user-account at the service provider's end, while the service provider ID for example protects against a malicious service provider reusing a user's token towards another service provider. The validity time ensures that a temporary corruption of a user cannot be used to issue tokens for the future. Such contents make the token traceable and linkable (cf. Sect. 3).

Remember that unlinkability means colluding service providers cannot lean which users they have in common. More formally, let $\mathsf{uid}_0, \mathsf{uid}_1$ denote two registered users, and let M, M', P, P' denote two adversarially chosen messages and policies that do not contain uid_0 and uid_1, and P, P' are both satisfied by both accounts. Let b denote a random bit, and σ a token on M, P generated for uid_0, and σ_b a token on M', P' generated for uid_b. Then, given access to σ, σ_b and the transcripts of the two token generations, no adversary shall be able to guess b with probability significantly greater than $1/2$. Note the unlinkability property implies that tokens are *anonymous*, i.e., don't reveal which uid account at the IdPs was used to generate them, *unless* this information is already leaked by M, P itself. In other words, an ab-dSSO system guarantees that the cryptography does not leak additional information about uid towards the SP.

In order to exploit the unlinkability our ab-dSSO scheme provides, one could use *pseudonyms*, for example one pseudonym per service provider to ensure unlinkability of tokens created for different providers. Usage of fresh pseudonyms for every token generation in our ab-dSSO scheme results in full unlinkability, and still has use cases such as proof of negative disease testing [47], although

compatibility with standard token format [53, 78, 83] needs to be checked.

Concretely, such pseudonyms can be realized for a user with unique ID uid, service provider with ID spid as follows: During the *setup* phase each server S_i generates a random point r_i on a random degree $t-1$ polynomial, with constant term r, in a group where the discrete logarithm problem is hard. This can be done using Feldman secret sharing. The servers then select a cryptographic hash function H, whose image is the set of DL-secure group generators in the field where r lives. They use H to compute $H(\mathsf{uid}, \mathsf{spid}) = g$. During the *signing* phase the servers compute $g^r = \mathsf{pseudo}$ through polynomial interpolation on the values g^{r_i}. The value pseudo will then be the pseudonym used for user uid and service provider spid. During the *refresh* phase r_i can get refreshed in a similar manner as the MAC keys and MAC shares using Feldman secret sharing. It is easy to see that the pseudonym pseudo cannot be linked to a user/service provider unless r is know[6]. Since pseudo is only related to the content of the tokens relayed by a user to a service provider, it does not affect the IdP servers' ability to revoke attributes of a user with uid, when learning about a revocation need from the attribute authorities.

When it comes to untraceability, remember it means that a colluding IdP and service provider cannot trace if a token belongs to the same user. Thus untraceability is intuitively very hard to achieve in practice as holding identifiable user-information might be *legally* required of both the IdP and certain service providers. Thus even if the token itself does not create a trace, meta-information associated with the users on *each* side would. Furthermore, in our case the data in the token must also be non-unique, i.e. without granular time-stamps and other information, as this would make it trivial to trace. However, if we assume these constraints are met, then our concrete ab-dSSO system does achieve some form of protection against IdPs, since it *hides the combined token* from the IdP server, as long as at most $t-1$ of them are corrupt. This is due to the distributed nature of the IdP and the user being the one who assembles token shares. While the use-case of no identifiable metadata stored on both the IdP and SP, and non-unique tokens might seem contrived, we note that this is exactly the desired case for vaccine validation. I.e. we do not wish the service provider learning the identity of the user who is proving they have a valid covid vaccine (e.g., M =`valid-covid-vacc`). Hence, for non-unique and SP-anonymous token contents our ab-dSSO system *does* prevent IdPs from tracking which service providers a user wants to log in to. However, such token formats are not allowed in standards such as SAML or OIDS with which we would like an ab-dSSO scheme to be compatible with. Specifically these standards require some notion of a uid to be included and hence we cannot hope to have both compatibility with standards *and* protection against a tracking IdP at the same time. We consider compatibility of greater importance, and hence refrain from demanding untraceability. Still, for future SSO standards allowing such anonymous token contents, our concrete ab-dSSO scheme can provide protection against curious IdPs.

[6]The servers are however able to link pseudo to a user and service provider since they know the uid and spid when computing pseudo, and learn pseudo in plain.

D MPC PRELIMINARIES

D.1 Secure Multi-Party Computation.

We are building our protocol on top of any already existing MPC scheme which offers the standard interface of an arithmetic black-box over a field larger than 2^s for security parameter s. We describe the ideal functionality in Fig. 11. $\mathcal{F}_{\mathsf{ABB}}(n,p)$ allows n servers $S_1,\ldots,S_n$ to evaluate any arithmetic function over field $\mathbb{F}_p$, on secret/private inputs provided by the servers. Since $\mathcal{F}_{\mathsf{ABB}}()$ accepts inputs only from a specific set of hard-coded servers, and also gives output only to those servers, it requires strong server authentication from any protocol realizing it. $\mathcal{F}_{\mathsf{ABB}}(n,p)$ allows the adversary to (unidentifiable) abort at any point. An abort is implicitly modeled by an adversary blocking messages of the honest servers.

The functionality is parametrized by the number of servers executing the computation, n, and the prime modulo which the computation is over, p. The functionality accepts messages from servers $S_1,\ldots,S_n$, for identifiers $\{S_i\}_{i\in[n]}$ that are encoded in *sid*, and from arbitrary other parties U. On input from any party in $\{U, S_i\}_{i\in[n]}$, the functionality sends an immediate output to all S_i, including the identifier of the requesting party, and excluding private values.

Input: On input (INPUT, *sid*, *ssid*, x, mode) from some party [I.1] $pid \in \{U, S_i\}_{i\in[n]}$ for a fresh *ssid*, if mode = priv then keep x private. Then do:
- If mode = rnd, sample $x \leftarrow_R \mathbb{F}_p$;
- Store $(ssid, x)$.

Compute: On input (COMPUTE, *sid*, $ssid_1$, $ssid_2$, $ssid_3$, mode) for a fresh $ssid_3$ from servers S_i for all $i \in [n]$, do:
- Retrieve $(ssid_1, x)$, $(ssid_2, y)$ or else ignore the query.
- If mode = mult then store $(ssid_3, x \cdot y \mod p)$.
- If mode = add then store $(ssid_3, x + y \mod p)$.

Output: On input (OUTPUT, *sid*, *ssid*, $\mathcal{P}$) from $pid \in \{S_i\}_{i\in[n]}$ for some set of party identifiers $\mathcal{P}$, do:
- Retrieve $(ssid, x)$ or else ignore the query.
- If (OUTPUT, *sid*, *ssid*, $\mathcal{P}$) was received from all $S_i, i \in [n]$ then send a delayed output (OUTPUT, *sid*, *ssid*, x) with private x to all parties in $\mathcal{P}$.

Figure 11: Ideal MPC functionalities. Arithmetic black-box: Without boxed code the figure depicts functionality $\mathcal{F}_{\mathsf{ABB}}(n,p)$, which implements secure arithmetic field operations between n fixed servers $\{S_1,\ldots,S_n\}$. Outsourced MPC: With boxed code included, the figure depicts $\mathcal{F}_{\mathsf{reqABB}}(n,p)$, which extends $\mathcal{F}_{\mathsf{ABB}}(n,p)$ by allowing input from arbitrary requestors U, who do not have to be among the n servers carrying out the secure computation.

We now introduce some shorthand notation for protocols modularly using $\mathcal{F}_{\mathsf{ABB}}$. To refer to values $(ssid, x)$ stored within $\mathcal{F}_{\mathsf{ABB}}$, we use either x or $[x]$: if the value is not known to all S_i, $i \in [n]$, we call it *secret* and use $[x]$, and otherwise we call it *public* and refer to it as x. We then write, e.g., $[z] \leftarrow [x] + [y] \mod p$ as shorthand for querying $\mathcal{F}_{\mathsf{ABB}}$'s COMPUTE interface in mode = add with the corresponding identifiers of x, y and z, and similarly for other functions that can be combined from addition and multiplication. For example, exponentiation $[z] \leftarrow [x]^y$ with public y can be realized by multiplying $[x]$ y times with itself. We use $x \xrightarrow{\mathsf{priv}} \mathcal{F}_{\mathsf{ABB}}^{sid}$ as shorthand for sending (INPUT, *sid*, *ssid*, x, priv) to

$\mathcal{F}_{\text{ABB}}$ with a fresh *ssid*, or even use $\{x, y\} \xrightarrow{\text{priv}} \mathcal{F}_{\text{ABB}}^{sid}$ for issuing one such query with x and one with y. Similarly, we use $x \xrightarrow{\text{pub}} \mathcal{F}_{\text{ABB}}^{sid}$ for public inputs. We use $([x], \mathcal{P}) \xrightarrow{\text{open}} \mathcal{F}_{\text{ABB}}^{sid}$ as shorthand for sending $(\text{OUTPUT}, sid, ssid, \mathcal{P})$ to $\mathcal{F}_{\text{ABB}}$ that has stored $(ssid, x)$. We denote by $[r] \xleftarrow{\text{rnd}} \mathcal{F}_{\text{ABB}}^{sid}$ a call $(\text{INPUT}, sid, ssid, \bot, \texttt{rnd})$ to $\mathcal{F}_{\text{ABB}}$ with a fresh *ssid* upon which $\mathcal{F}_{\text{ABB}}$ has sampled $r \leftarrow_R \mathbb{F}_p$ and stored $(ssid, r)$, and extend this notation also to sampling multiple values $[r], [r']$ as $[r], [r'] \xleftarrow{\text{rnd}} \mathcal{F}_{\text{ABB}}^{sid}$. Functionality $\mathcal{F}_{\text{ABB}}$ is more general than similar functionalities from the literature, because it allows for the sampling of random elements and output to specific servers only. Most protocols in the literature only shows how to get output to *all* servers. However, our more general interface is realizable by the standard interfaces of arithmetic black boxes found in the literature. For example, to sample a random value, every server samples an additive share $r_i \leftarrow_R \mathbb{F}_p$ and sends $r_i \xrightarrow{\text{priv}} \mathcal{F}_{\text{ABB}}$, and then servers jointly compute $[r] = \sum_{i\in[n]}[r_i]$. If at least one server is honest, r is uniformly random. Similarly, to output $[x]$ to only server S_i, S_i sends $r \xrightarrow{\text{priv}} \mathcal{F}_{\text{ABB}}$ for a random r, then servers jointly compute $[z] = [x] + [r]$ and have this value be opened to all servers. The random r information theoretically hides x to all servers $j \neq i$, but server S_i can restored $x = z - r \mod p$. $\mathcal{F}_{\text{ABB}}$ can be realized using SPDZ [41, 63]. SPDZ is be implemented in the *preprocessing model*, where a function-independent computationally secure preprocessing protocol is executed before the function or data of the computation is known. This is called the *offline phase*. While such preprocessing requires computation assumptions, the corresponding *online phase* (which uses the preprocessed data once the function and data to be computed on becomes known) is purely information theoretically against a static, active adversary corrupted up to $n-1$ servers [41]. Concretely such preprocessing can for example be realized using UC-secure oblivious transfer and a pseudorandom function [63]

Theorem D.1 ([63]). *Assume the existence of UC-secure oblivious transfer and a pseudorandom function (on a seed of at least the length of the security parameter), then $\mathcal{F}_{\text{ABB}}$ can be UC-securely realized against a static and malicious adversary corrupting at most $n-1$ servers where $p \geq 2^s$.*

The above security statement is *not* considering proactive security, while, looking ahead, the upcoming section will introduce a signature scheme that is secure against such adversary. The reason why we do not require $\mathcal{F}_{\text{ABB}}$ to ensure proactive security is that our SSO protocol will use many short-lived instances of $\mathcal{F}_{\text{ABB}}$, such that the corruption schema will not change during the livetime of each $\mathcal{F}_{\text{ABB}}$ instance. Hence, we do not require proactive security of this building block. Next to the above theorem, other assumptions are possible, e.g., using semi-homomorphic encryption [64].

Figure 11 additionally provides an augmented functionality $\mathcal{F}_{\text{reqABB}}$, which accepts arbitrary users next to $S_1, \cdot, S_n$ to give input or receive output from a secure computation ([I.1]). Such a protocol is known as *outsourced MPC*, and closely reflects a real-world scenario where a few servers execute an MPC computation, but with a potentially large set of clients providing private input and receiving private output of such a computation. The goal of outsourced MPC is to reduce the computation load on the client. We describe in App. D.1 how functionality $\mathcal{F}_{\text{reqABB}}$ can be realized from the literature.

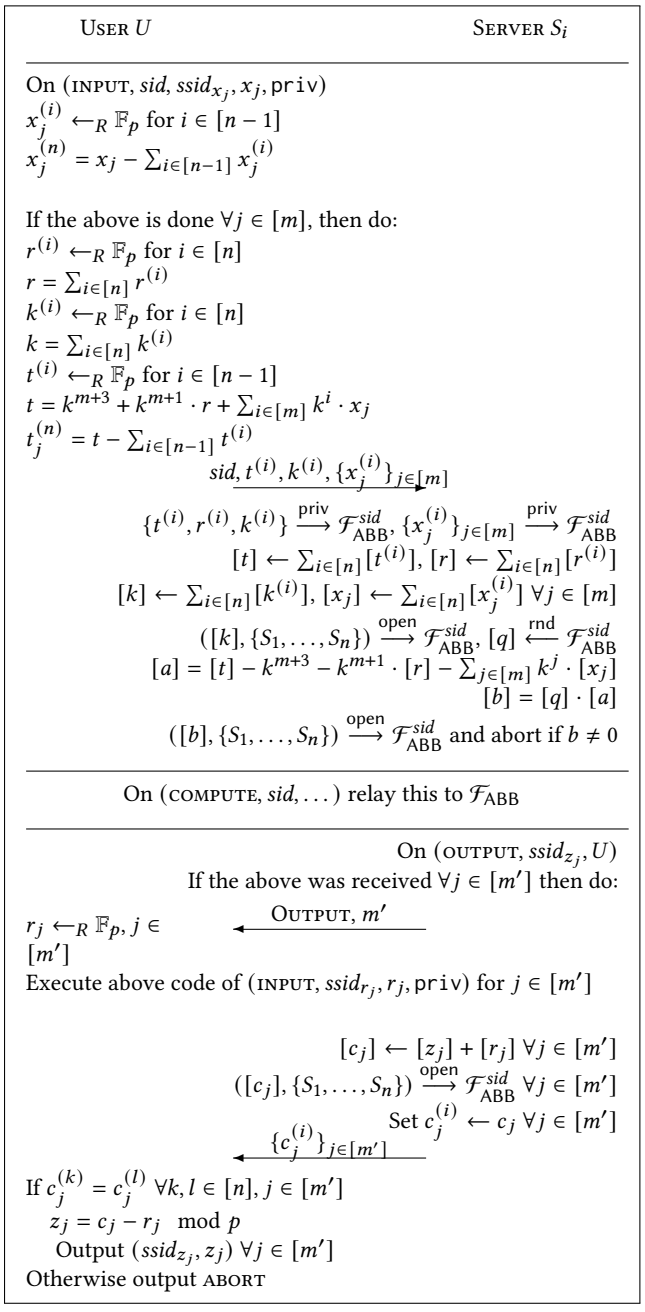

Figure 12: Protocol $\prod_{\text{reqABB}}$ realizing functionality $\mathcal{F}_{\text{reqABB}}$ in the $\mathcal{F}_{\text{ABB}}$-hybrid model. The top part shows how a batch of m inputs $x_1, \ldots, x_m$ of the user can be input into $\mathcal{F}_{\text{ABB}}$ by the servers in an oblivious way, yielding $[x_1], \ldots, [x_m]$. The bottom part shows how a batch of m' values $[z_1], \ldots, [z_{m'}]$ can be revealed to the user. COMPUTE inputs to servers are relayed to $\mathcal{F}_{\text{ABB}}$.

D.2 Outsourced computation

We describe here how $\mathcal{F}_{\mathsf{reqABB}}$ can be realized from the literature. First, note that $\mathcal{F}_{\mathsf{reqABB}}$ is not equivalent to $\mathcal{F}_{\mathsf{ABB}}$, since the clients do not necessarily participate in the computation, i.e., they do not issue (COMPUTE, $ssid_1$, $ssid_2$, $ssid_3$, mode) commands. It is this small change that allows for the realizing protocol to require minimal interaction and computation from clients, and, e.g., perform expensive interactive multiplication steps only among the servers. This is in particular the case since many MPC schemes require computation between every set of computing parties in order to execute a multiplication. Thus the computation scales quadratically in the amount of parties participating in the computation. While we of course could use $\mathcal{F}_{\mathsf{ABB}}$ to implement a similar functionality for any reasonable usage $\mathcal{F}_{\mathsf{reqABB}}$, the point is that it could easily become prohibitory expensive when scaling to many clients if they need to perform a quadratic amount of communication in the *total* amount of clients and server for *each* multiplication to be carried out.

Many papers have considered different notions of this idea, including for a single server carrying out a computation on some notion of encrypted data received by clients [61], or multiple servers [13, 39, 62]. However most protocols for outsourced computation require a concrete realization of $\mathcal{F}_{\mathsf{ABB}}$ and thus do not present a protocol for $\mathcal{F}_{\mathsf{reqABB}}$ in the $\mathcal{F}_{\mathsf{ABB}}$-hybrid model. The earliest work we know of which describe a highly efficient approach to outsourcing secure computation in the $\mathcal{F}_{\mathsf{ABB}}$-hybrid model is described by Jakobsen *et al.* [58]. Their scheme only requires the client to execute simple operation, independent of the function to be carried out using $\mathcal{F}_{\mathsf{ABB}}$, and only requires a constant amount of interaction by *each* client with the servers and does not require clients to interact with each other.

Their underlying idea is simply to have parties compute a secret-shared MAC from an algebraic manipulation detection code on their input, along with some randomness, and then distribute these shares to the servers. The servers then relay this into the MPC computation, which verifies the MAC to ensure that no corrupt server has tried to manipulate the clients' shares. Once the computation is done, the randomness supplied by the client is used to one-time pad the output that the client should learn, and the padded value is opened towards all servers. Each server then relays this to the appropriate client. The clients can then easily remove the padding and learn their respective output. Furthermore, since each client is supposed to receive the same output from all servers, it can also ensure that no server has tried to manipulate it, by validating that they receive the same value from all the servers.

Jakobsen *et al.* proves their scheme UC-secure assuming access to an underlying MPC scheme with ideal functionality similar to $\mathcal{F}_{\mathsf{ABB}}$. However, the box for their ideal functionality [58, Fig. 3] is slightly different from the one we present in $\mathcal{F}_{\mathsf{reqABB}}$. The difference is that Jakobsen *et al.* assume *all* clients give input and *all* clients get output, and that the output follows after all computations have been carried out. It is easy to see that all clients giving input and receiving output is simply a super-set of some parties giving input and some parties giving output as a dummy input and output can always be used. However, their functionality is not *reactive* (reactive meaning that output and computation are interleaved multiple times). That said, since $\mathcal{F}_{\mathsf{ABB}}$ is inherently reactive, the outsourced MPC scheme can also be this. We can conclude this from the following observations: 1) the client's output of the computation is indistinguishable from random to an adversary, as it is one-time padded by randomness additively shared by the client. 2) all of the client's inputs (except one uniformly random value) are statically indistinguishable from random if at most $n-1$ malicious servers are corrupted. Hence what is learned by the servers is indistinguishable from the clients' input and output. This allows the clients to do give input and receive output multiple times, regardless of the underlying computation being undertaken.

Details. We here sketch the protocol of Jakobsen *et al.* for allowing clients to receive input and output in Fig. 12 adapted to the ideal functionality in Fig. 11. In particular, unlike Jakobsen *et al.*, we separate the clients' steps for giving true input and giving random input (used to pad the output). This is to allow a fully reactive version of outsourced MPC. However, if it is known in advance how many outputs each client should get, this can be done in a single step which is exactly what Jakobsen *et al.* describe in their paper. We note that the underlying computation (i.e. calls to $\mathcal{F}_{\mathsf{reqABB}}$.(COMPUTE, $ssid_1$, $ssid_2$, $ssid_3$, mode)) can be carried out with the equivalent calls to the $\mathcal{F}_{\mathsf{ABB}}$ functionality.

D.3 MPC with partially public functions

We note that both $\mathcal{F}_{\mathsf{ABB}}$ and $\mathcal{F}_{\mathsf{reqABB}}$ can be assumed to implement computation of a series of arbitrary functions on possibly private input and possibly public output with a persistent state through the following folklore approach: Any computation where all values used are assumed or allowed to be public, the servers simply use $\mathcal{F}_{\mathsf{ABB}}$, $\mathcal{F}_{\mathsf{reqABB}}$ to output these public values to all servers and all server carry out the computation on these values in plain. Finally they input the public result back into $\mathcal{F}_{\mathsf{ABB}}$, $\mathcal{F}_{\mathsf{reqABB}}$ where it can then be computed on with private and hidden values. More concretely this is done through interleaved usage of commands (INPUT, ...), (OUTPUT, ...), and (COMPUTE, ...). That is, each time series of commands (OUTPUT, ...) is issued it defines the computation of a function. This output can then be used outside of the functionality in $\mathcal{F}_{\mathsf{ABB}}$, $\mathcal{F}_{\mathsf{reqABB}}$. If the outputs are public values, then they can be used to do some other *public* computation whose result, x can be inputted back into $\mathcal{F}_{\mathsf{ABB}}$, $\mathcal{F}_{\mathsf{reqABB}}$ through (INPUT, *sid*, *ssid*, x, pub). Thus knowledge of the function to compute using $\mathcal{F}_{\mathsf{ABB}}$, $\mathcal{F}_{\mathsf{reqABB}}$ can be used to separate it into public and private parts. We formalize the details of this below.

Assume the series of commands of the form (COMPUTE, *sid*, $ssid_1$, $ssid_2$, $ssid_3$, mode) to $\mathcal{F}_{\mathsf{reqABB}}$ is known before parties start giving input to $\mathcal{F}_{\mathsf{reqABB}}$ under *sid*. In this case we can assume the computation to be done is expressed as a Directed Acyclic Graph (DAG) where on a list of inputs (through commands of the form (INPUT, *sid*, *ssid*, x, mode), of which the leaves can be given as output through commands of the form (OUTPUT, *sid*, *ssid*, x, $\mathcal{P}$). That each, each *ssid* identifies a node in the DAG and each *ssid* used in a command of the form (OUTPUT, *sid*, *ssid*, x, $\mathcal{P}$) is a leaf[7] Such a DAG can be expressed as a function $f : \mathbb{Z}_p^a \to \mathbb{Z}_p^b$ where a expresses the amount of calls of

[7]We assume without loss of generality that once an *ssid* has been used in an OUTPUT command it will not be used in a COMPUTE command. If it needs to be it can always be given as input through the command (INPUT, *sid*, *ssid*, x, pub).

the form (INPUT, *sid*, *ssid*, x, mode) and b expresses the amount of calls of the form (OUTPUT, *sid*, *ssid*, $x, \mathcal{P}$). What is explained above is actually weaker than what is offered by $\mathcal{F}_{\mathsf{reqABB}}$. In fact what is explained above is known as *Secure Function Evaluation* (SFE) or *non-reactive* MPC. However $\mathcal{F}_{\mathsf{reqABB}}$ actually offers us to execute a series of functions $f_1, f_2, \ldots$ where $f_i : \mathbb{Z}_p^{a'} \to (\mathbb{Z}_p^{\bar{b}}, \mathbb{Z}_p^{b'})$ and b' expresses the amount of calls of the form (OUTPUT, *sid*, *ssid*, $x, \mathcal{P}$) for function f_i and $\bar{b}$ is the amount of calls to of the form (COMPUTE, *sid*, $ssid_1$, $ssid_2$, $ssid_3$, mode) or (INPUT, *sid*, *ssid*, x, mode) in f_i. That is, $\bar{b}$ expresses the amount of nodes in the DAG of f_i minus the nodes whose *ssid* has been used in call (OUTPUT, ...). It is now clear that if the output of function f_i is public to all parties, they can use this, and potentially auxiliary input, to compute another *public* function g_i on this. The output of g_i can obliviously be used as *public* input to any function f_j for $j \geq i$. In fact, the output of g_i might influence which function f_j to compute. Thus considering the calls to $\mathcal{F}_{\mathsf{ABB}}$, $\mathcal{F}_{\mathsf{reqABB}}$ as interleaved partial functions with a shared state allows us to "take out" any public computations of this and carry it out in plain by each of the servers.

With this discussion in mind for computation of functions, which consists of parts where the values can be publicly known, allows one to realize $\mathcal{F}_{\mathsf{ABB}}$, $\mathcal{F}_{\mathsf{reqABB}}$ more efficiently by executing the public part outside $\mathcal{F}_{\mathsf{ABB}}$, $\mathcal{F}_{\mathsf{reqABB}}$, by all the servers. That is it is possible to implement functionalities $\mathcal{F}_{\mathsf{ABB}}^{\mathcal{C}}$, $\mathcal{F}_{\mathsf{reqABB}}^{\mathcal{C}}$ for a specific, known computation $\mathcal{C}$ describing multiple public and private functions, in the $\mathcal{F}_{\mathsf{ABB}}$, $\mathcal{F}_{\mathsf{reqABB}}$ hybrid model. Concretely this is then done by using calls to (OUTPUT, ...) to output values that can be learned publicly and have parties locally execute (COMPUTE, *sid*, $ssid_1$, $ssid_2$, $ssid_3$, mode) on these public values. For public values that must then be used in a (COMPUTE, ...) call with a private value are given as input through (INPUT, ...).

In App. C we discuss how we can take advantage of this to compute the setup phase of our scheme more efficiently in several real world situations.

E PROACTIVELY-SECURE THRESHOLD SIGNATURES

E.1 Security model

We introduce a UC functionality $\mathcal{F}_{\mathsf{psThrSig}}$ to model proactively-secure threshold signatures in Fig. 18. We use the writing conventions introduced in Sect. 2. $\mathcal{F}_{\mathsf{psThrSig}}$ acts as a trusted signing authority: It maintains a key pair (vk, sk) of some signature scheme (KGen, Sign, Vfy) and issues and verifies signatures with these keys. Although $\mathcal{F}_{\mathsf{psThrSig}}$ can use any set of algorithms (KGen, Sign, Vfy), we are particularly interested in parametrizing $\mathcal{F}_{\mathsf{psThrSig}}$ with signature algorithms that are existentially unforgeable, and which allow for threshold computation of signatures.

We now explain the functionality in detail, using the [X.Y] markings in the functionality code, and indicating in boldface the properties from above. We first do an honest walk-through and explain handling of corrupt users and servers separately afterwards.

Key generation. $\mathcal{F}_{\mathsf{psThrSig}}$ is parametrized with a digital signature scheme SIG = (KGen, Sign, Vfy) and, if [K.1] all servers assist in the setup procedure, $\mathcal{F}_{\mathsf{psThrSig}}$ [K.2] generates its own key pair (vk, sk). From that point on, the other interfaces of $\mathcal{F}_{\mathsf{psThrSig}}$ can be called

Key Generation. Executed by each server $S_i \in \Sigma$.

On input (KeyGen, *sid*), server S_i calls $\mathcal{F}_{\mathsf{DKG}}$ on input (Sample, *sid*, *ssid*).

Upon receiving (Receipt, *sid*, *ssid*) from $\mathcal{F}_{\mathsf{DKG}}$, server S_i calls $\mathcal{F}_{\mathsf{DKG}}$ on (Output, *sid*, *ssid*).

Upon receiving (Output, *sid*, *ssid*, N, d_0, d_i) from $\mathcal{F}_{\mathsf{DKG}}$, server S_i proceeds as follows.

- Call the coin-tossing functionality $\mathcal{F}_{\mathsf{CT}}$ on input (Toss, *ssid*, N).
- Upon receiving (Random, *ssid*, g) from $\mathcal{F}_{\mathsf{CT}}$, run $\text{FELDMAN}_{\mathbb{Z}_N}$.SHARE({param := g, t, n}, *ssid*, d_i) → (*ssid*, {priv := $\delta_{i,1}, \ldots, \delta_{i,n}$}, {pub := $A_{i,0}, \ldots, A_{i,t-1}$}).
- Upon receiving, from all S_j for $j \in [n] \setminus \{i\}$, Feldman shares $\delta_{j,i}$ and polynomial-coefficient commitments $(A_{j,0}, \ldots, A_{j,t-1})$ proceed as follows:
  - Abort if $\text{FELDMAN}_{\mathbb{Z}_N}$.VERIFY$(A_{j,0}, \ldots, A_{j,t-1}, \delta_{j,i})$ rejects for any $j \neq i$.
  - Compute $A_k \leftarrow \prod_{j=1}^{n} A_{j,k}$ for all $k = 0, \ldots, t-1$.
  - Abort if all server shares do not sum up to $e^{-1} \bmod \varphi(N)$, i.e. if $A_0^e \neq g^{n!} \bmod N$ (note that $A_0 = \prod_{j=1}^{n} A_{j,0} = g^{n! \cdot \sum_j d_j} = g^{n! \cdot d} \bmod N$).
  - Compute $\delta_i \leftarrow \sum_{j \in [n]} \delta_{j,i}$ (for any $T \subseteq \Sigma$ of size t, setting $\omega_i := \omega_{T,i} := n! \prod_{j \neq i:\, S_j \in T} j/(j-i) \in \mathbb{Z}$, it follows that $(d - d_0) \cdot (n!)^2 = \sum_{j \in [n]} d_j = \sum_{j \in [n]} \sum_{i:\, S_i \in T} \delta_{j,i} \omega_i = \sum_i \left(\sum_j \delta_{j,i} \right) \omega_i = \sum_i \delta_i \omega_i$).
  - Distribute t-out-of-n back-up shares of δ_i, i.e. run $\text{FELDMAN}_{\mathbb{Z}_N}$.SHARE({param := g, t, n}, $ssid'_i, \delta_i$) → ({priv := $\delta'_{i,1}, \ldots, \delta'_{i,n}$}, {pub := $A'_{i,0}, \ldots, A'_{i,t-1}$}).
  - Upon receiving, from all S_j for $j \neq i$, Feldman shares $\delta'_{j,i}$ and polynomial-coefficient commitments $(A'_{j,0}, \ldots, A'_{j,t-1})$,
    - abort if $\text{FELDMAN}_{\mathbb{Z}_N}$.VERIFY$(A'_{j,0}, \ldots, A'_{j,t-1}, \delta'_{j,i})$ rejects for any $j \neq i$.
    - abort if the newly shared values are inconsistent with the additive shares, i.e. if $A'_{j,0} \neq \prod_{\ell=0}^{t-1} A_\ell^{n! \cdot j^\ell} \bmod N$ for any $j \neq i$ (in an honest protocol execution, $A'_{j,0} = g^{\delta_j \cdot n!} = g^{n! \cdot \sum_{k=1}^{n} \delta_{k,j}} = \prod_k g^{\delta_{k,j} n!} = \prod_k \left(\prod_{\ell=0}^{t-1} A_{k,\ell}^{j^\ell} \right)^{n!} = \prod_\ell A_\ell^{n! \cdot j^\ell} \bmod N$).
- If $i = 1$, for this server S_i, then compute integers u and v such that $u \cdot e + v \cdot (n!)^2 = 1$ (they exists since $e > n$ by assumption and is prime) and broadcast (sid, u, v).
- If $i > 1$, upon receiving (sid, u, v) from S_1, abort if $u \cdot e + v \cdot (n!)^2 \neq 1$.
- Set $\text{epoch}_i \leftarrow 0$ and create a key-share record (KeyShare, $\text{epoch}_i, N, e, d_0, u, v, \delta_i, (\delta'_{j,i}, (A'_{j,k})_{k=0}^{t-1})_{j \in [n]}$).
- Broadcast (KeyConf, S_i) to all other servers.

Signing. Executed by each server $S_i \in T \subseteq \Sigma$, for any T of size t.

On input (Sign, *ssid*, M), U broadcasts (*ssid*, M) to all servers in T.

Upon receiving (Sign, *ssid*, M) from U, S_i outputs (Sign, *ssid*, M).

On input (ProceedSign, *sid*, *ssid*), S_i proceeds as follows.

- Retrieve the key-share record.
- Compute $\sigma_i \leftarrow H(M)^{\delta_i} \bmod N$ and send $(ssid, N, e, d_0, u, v, \sigma_i)$ to user U.

Upon receiving tuples $(ssid, N_i, e_i, d_{0,i}, u_i, v_i, \sigma_i)$ from all servers $S_i \in T$, user U does the following.

- Check that the values $N_i, e_i, d_{0,i}, u_i, v_i$ received from all servers are the same (they are subsequently denoted by N, e, d_0, u, v), and abort if it is not the case.
- For all $i \in T$, compute $\omega_i := \omega_{T,i} := n! \prod_{j \neq i:\, S_j \in T} j/(j-i)$.
- Compute $\sigma \leftarrow H(M)^u \cdot \left(H(M)^{d_0 \cdot (n!)^2} \prod_i \sigma_i^{\omega_i} \right)^v \bmod N$. (Note that $(d - d_0) \cdot (n!)^2 = \sum_{j \in [n]} \sum_{i:\, S_i \in T} \delta_{j,i} \omega_i = \sum_i \left(\sum_j \delta_{j,i} \right) \omega_i = \sum_i \delta_i \omega_i$.)
- If $H(M) = \sigma^e \bmod N$ then output (Signature, M, σ), else abort.

Verification. On input (Verify, *sid*, $vk' := (N', e'), M, \sigma$), verifier V sets $f \leftarrow 1$ if $H(M) = \sigma^{e'} \bmod N'$ and $f \leftarrow 0$ otherwise, and outputs (Verified, vk', m, σ, f).

Figure 13: Protocol Π_{psThrSig} for a threshold RSA signature scheme (refresh protocol in Fig. 14) using macro FELDMAN of Fig. 15.

Refresh. Executed by each server $S_i \in \Sigma$.

- On input $(\mathtt{Refresh}, sid, S_i)$, retrieve record $(\mathtt{KeyShare}, \mathsf{epoch}_i, N, e, d_0, u, v, \delta_i, (\delta_{j,i}, (A_{j,k})_{k=0}^{t-1})_{j\in[n]})$ and broadcast $(A_{j,0}, \ldots, A_{j,t-1})_{j\in[n]}$.
- Upon receiving such tuples from all the other servers in Σ, if there exists a tuple that was broadcast by at least t servers, overwrite $(A_{j,0}, \ldots, A_{j,t-1})_{j=1}^{n}$ with the values from that tuple if at least one value differs. If no such tuple exists, abort.
- If $g^{\delta_i \cdot n!} \neq A_{i,0} \bmod N$, i.e. if δ_i is inconsistent with the originally shared value, then:
  - Broadcast to all servers in Σ a request to privately send the back-up shares for S_i they hold.
  - Upon receiving back-up shares $\delta_{i,j}$ from all servers in Σ, run $\textsc{Feldman}_{\mathbb{Z}_N}.\textsc{Verify}(A_{i,0}, \ldots, A_{i,t-1}, \delta_{i,j})$. If $n - t \geq t$, there must be at least t correct shares. Let $T \subseteq \Sigma$ of size t such that the shares received by each $S_i \in T$ is valid. Then, let $\delta_i \leftarrow \textsc{Feldman}_{\mathbb{Z}_N}.\textsc{Reconstruct}(T, (\delta_{i,j})_{j:\, S_j\in T})$.
- Run procedure $\textsc{Feldman}_{\mathbb{Z}_N}.\textsc{Share}(\{\mathtt{param} := g, t, n\}, ssid_i, \delta_i) \rightarrow (ssid_i, \{\mathtt{priv} := \delta'_{i,1}, \ldots, \delta'_{i,n}\}, \{\mathtt{pub} := A'_{i,0}, \ldots, A'_{i,t-1}\})$.
- Upon receiving, from all S_j for $j \neq i$, Feldman shares $\delta'_{j,i}$ and polynomial-coefficient commitments $(A'_{j,0}, \ldots, A'_{j,t-1})$,
  - abort if $\textsc{Feldman}_{\mathbb{Z}_N}.\textsc{Verify}(A'_{j,0}, \ldots, A'_{j,t-1}, \delta'_{j,i})$ rejects for any $j \neq i$,
  - abort if the newly shared values are inconsistent with the ones from the previous refresh, i.e. if $A'_{j,0} \neq A_{j,0}$.
- Choose a set $U'_i \subseteq [n]$ of size t uniformly at random, broadcast it, and compute $\omega_{U'_i,j} := n! \prod_{k\neq j:\, S_k\in U} k/(k-j)$ for all j such that $S_j \in U'_i$.
- Compute $\delta'_i \leftarrow \sum_{j:\, S_j\in U_i} \delta'_{j,i}\omega_{U_i,j}$. (For any $T \subseteq \Sigma$ of size t, $(d - d_0) \cdot (n!)^2 = \sum_j \delta_j \omega_{U_i,j} = \sum_j \left(\sum_{i:\, S_i\in T} \delta'_{j,i}\omega_{T,i}\right)\omega_{U_i,j} = \sum_i \left(\sum_j \delta'_{j,i}\omega_{U_i,j}\right)\omega_{T,i}$.)
- Distribute t-out-of-n back-up shares of δ'_i, i.e. run $\textsc{Feldman}_{\mathbb{Z}_N}.\textsc{Share}(\{\mathtt{param} := g, t, n\}, ssid'_i, \delta'_i) \rightarrow (ssid'_i, \{\mathtt{priv} := \delta''_{i,1}, \ldots, \delta''_{i,n}\}, \{\mathtt{pub} := A''_{i,0}, \ldots, A''_{i,t-1}\})$.
- Upon receiving, from all S_j for $j \neq i$, Feldman shares $\delta''_{j,i}$ and polynomial-coefficient commitments $(A''_{j,0}, \ldots, A''_{j,t-1})$,
  - Abort if $\textsc{Feldman}_{\mathbb{Z}_N}.\textsc{Verify}(A''_{j,0}, \ldots, A''_{j,t-1}, \delta''_{j,i})$ rejects for any $j \neq i$.
  - abort if the newly shared values are inconsistent with the previous ones, i.e. if $A''_{j,0} \neq \prod_{\ell=0}^{t-1}\left(\prod_k (A'_{k,\ell})^{\omega_{U_j,k}}\right)^{n!\cdot j^\ell} \bmod N$ for any $j \neq i$ (in an honest protocol execution, $A''_{j,0} = g^{\delta'_j \cdot n!} = g^{n!\cdot\sum_{k:\, S_k\in U_j}\delta'_{k,j}\omega_{U_j,k}} = \prod_k g^{\delta'_{k,j}\omega_{U_j,k}n!} = \prod_k \left(\prod_{\ell=0}^{t-1}(A'_{k,\ell})^{j^\ell}\right)^{\omega_{U_j,k}n!} = \prod_\ell \left(\prod_k (A'_{k,\ell})^{\omega_{U_j,k}}\right)^{n!\cdot j^\ell} \bmod N$).
- Set $\mathsf{epoch}_i \leftarrow \mathsf{epoch}_i + 1$, update the key-share record to $(\mathtt{KeyShare}, \mathsf{epoch}_i, N, e, d_0, u, v, \delta'_i, (\delta''_{j,i}, (A''_{j,k})_{k=0}^{t-1})_j)$ and output $(\mathtt{Epoch}, sid, \mathsf{epoch}_i)$.

Figure 14: Protocol Π_{psThrSig} for a threshold RSA signature scheme - refresh phase, using macro $\textsc{Feldman}$ of Fig. 15.

Global parameters: integer $N > 1$, $g \in \mathbb{Z}_N^*$, integers $n \geq t \in \mathbb{Z}_{>1}$.

- $\textsc{Feldman}_{\mathbb{Z}_N}.\textsc{Share}$. Executed between a dealer D and a set $\Sigma := \{S_1, \ldots, S_n\}$ of parties (D may be in Σ).

On **input** $(\mathtt{Share}, sid, ssid, s)$, with $s \in [-nN^2, nN^2]$, dealer D proceeds as follows.

- Generate $a_1, \ldots, a_{t-1} \leftarrow_R [-n(n!)^2N^3, n(n!)^2N^3]$. Let $a_0 := s \cdot n!$ and $P := a_0 + a_1X + \cdots + a_{t-1}X^{t-1}$.
- For $i \in [n]$, compute $P(i) \in \mathbb{Z}$. Compute also $A_j \leftarrow g^{a_j} \bmod N$ for $j \in [0, t-1]$.
- Broadcast $(A_0, \ldots, A_{t-1})$, and for $i \in [n]$, send $(ssid, z_i \leftarrow P(i))$ to S_i.
- **Return** $(ssid, \{\mathtt{priv} := z_1, \ldots, z_n\}, \{\mathtt{pub} := A_0, \ldots, A_{t-1}\})$.

- $\textsc{Feldman}_{\mathbb{Z}_N}.\textsc{Verify}$. Executed by any party $S_i \in \Sigma$.

On **input** $(\mathtt{Verify}, sid, A_0, \ldots, A_{t-1}, z_i)$, accept if $g^{z_i} = \prod_{j=0}^{t-1}(A_j)^{i^j} \bmod N$ and otherwise reject.

- $\textsc{Feldman}_{\mathbb{Z}_N}.\textsc{Reconstruct}$. Executed by any party $S_i \in \Sigma$.

On **input** $\left(\mathtt{Reconstruct}, sid, T, (z_j)_{j:\, S_j\in T}\right)$, with $T \subseteq \Sigma$ of size t, party S_i does the following.

- Find the smallest prime $q > 2nN^2$.
- Compute and **return** $s \leftarrow (n!)^{-1} \cdot \sum_{j:\, S_j\in T} z_j\omega_j \bmod q$, with $\omega_j := \prod_{k\neq j} k/(k-j) \bmod q$.

Figure 15: Macro $\textsc{Feldman}$ for Feldman's Verifiable Secret-Sharing scheme in $\mathbb{Z}_N$ [82] using by $\mathcal{F}_{\mathsf{psThrSig}}$ in Fig. 13 and Fig. 14.

The functionality is parametrised by a public prime exponent e and a prime length ℓ and interacts with servers $\Sigma = \{S_1, \ldots, S_n\}$ (specified in sid), and an adversary $\mathcal{A}$ which may statically corrupt a set $C \subset \Sigma$ of servers and abort the session at any time.

Adversarial Shares. Upon receiving $(\mathtt{Shares}, sid, ssid, p_i, q_i)$ from $S_i \in C$, with $p_i, q_i \in \mathbb{Z}_{2^\ell}$, record $(sid, ssid, S_i, p_i, q_i)$ and send $(\mathtt{Receipt}, sid, ssid)$ to S_i.

Share Generation. Upon receiving $(\mathtt{Sample}, sid, ssid)$ from an honest S_i, record $(sid, ssid, S_i)$ and send an immediate output $(\mathtt{Receipt}, sid, ssid)$ to S_i.

Prime Generation. Upon receiving $(\mathtt{Output}, sid, ssid)$ from any party S_i, if a tuple $(sid, ssid, S_i)$ has been recorded for all honest parties S_i, and a tuple $(sid, ssid, S_i, p_i, q_i)$ has been recorded for each corrupted party $S_i \in C$, sample random ℓ-bit value pairs (p_i, q_i) for each honest party S_i so that the following constraints are satisfied.

- $p_1 = q_1 = 3 \bmod 4$
- $p_i = q_i = 0 \bmod 4$ for $i \in [n]\backslash\{1\}$
- $p = \sum_{i\in[n]} p_i$ and $q = \sum_{i\in[n]} q_i$ are prime
- $\gcd(p \cdot q, e) = \gcd(p \cdot q, p + q - 1) = 1$.

Next, create a record $(\mathtt{Primes}, sid, ssid, p, q)$ and send $(\mathtt{Generation}, sid, ssid, N = p \cdot q)$ to all parties.

Output.

Upon receiving $(\mathtt{Proceed}, sid, ssid, \{d_i\}_{S_i\in C})$ from $\mathcal{A}$, with $d_i \in [-nN^2, nN^2]$ for all $S_i \in C$, proceed as follows.

- Retrieve record $(\mathtt{Primes}, sid, ssid, p, q)$.
- Generate $d_i \leftarrow_R [-n \cdot N^2, n \cdot N^2]$ for each honest S_i.
- Let $d \in \mathbb{Z}$ be such that $ed = 1 \bmod \varphi(N)$ and $d_0 \leftarrow d - \sum_{i\in[n]} d_i$.
- Let $N \leftarrow p \cdot q$. For all $i \in [n]$, send to server S_i a delayed output $(\mathtt{Output}, sid, ssid, N, d_0, d_i)$ with private d_i.

Figure 16: Functionality $\mathcal{F}_{\mathsf{DKG}}$.

and $\mathcal{F}_{\mathsf{psThrSig}}$ will produce and verify signatures w.r.t this one key pair. $\mathcal{F}_{\mathsf{psThrSig}}$ must be parametrized with a standardized signature scheme to ensure **standardized token format**.

Signing. Any user U can request a signature from $\mathcal{F}_{\mathsf{psThrSig}}$ by providing the message to be signed M. The servers get [S.1] notified about the signing request, including the message to be signed (i.e.,

The functionality interacts with servers $\Sigma = \{S_1, \ldots, S_n\}$ (specified in *sid*) and an adversary $\mathcal{A}$ that statically corrupts a subset $C \subset \Sigma$ of servers and can abort the session at any time.

Toss. Upon receiving (Toss, *sid*, M) from all parties, with M a positive integer, sample a uniformly random element $x \in_R \mathbb{Z}_M$ and send a delayed output (Random, *sid*, x) to all parties.

Figure 17: Functionality $\mathcal{F}_{\mathsf{CT}}$.

The functionality is parametrised by algorithms SIG = (KGen, Sign, Vfy), a security parameter λ, a threshold t and initially undefined integer counters $\mathsf{epoch}_1, \ldots, \mathsf{epoch}_n$. It interacts with servers $\Sigma = \{S_1, \ldots, S_n\}$ (specified in *sid*), arbitrary users, verifiers and an adversary $\mathcal{A}$ that can delay any message or cause an abort of any phase at any point.

On receiving (KeyGen, *sid*) from server S_i
- Ignore if a record (key, *sk*, *vk*) exists
- If (KeyGen, *sid*) was received [K.1] from all S_i
  - [K.2] Record (key, *sk*, *vk*), $(vk, sk) \leftarrow_R \mathsf{KGen}\left(1^\lambda\right)$
  - For $i = 1, \ldots, n$ do:
    - On [K.3] (KeyConf, *sid*, S_i) from $\mathcal{A}$, set $\mathsf{epoch}_i \leftarrow 0$ and send an immediate output (KeyConf, *sid*, *vk*) to S_i.

On receiving (Sign, *sid*, *ssid*, M) from a user U
- Create a record (session, *sid*, *ssid*, M)
- Send a [S.1] delayed output (Sign, *sid*, *ssid*, M) to all $S_i \in \Sigma$.

On receiving (ProceedSign, *sid*, *ssid*) from a server $S_i \in \Sigma$
- Ignore if no record (session, *sid*, *ssid*, M) exists
- If there exists $\mathsf{epoch} \in \mathbb{N}$ such that (ProceedSign, *sid*, *ssid*) was received from [PS.1] t servers S_i while $\mathsf{epoch}_i = \mathsf{epoch}$
  - Send [PS.2] (sign-ok, *sid*, *ssid*) to $\mathcal{A}$ and receive back (sign-ok, *sid*, *ssid*)
  - [PS.3] $\sigma \leftarrow \mathsf{Sign}(sk, M)$
  - Create a [PS.4] record (sigrec, *vk*, $M, \sigma, 1$), abort if a record (sigrec, *vk*, $M, \sigma, 0$) already exists.
  - Send a [PS.5] delayed output (Signature, M, σ) with [PS.6] private σ to U.

On receiving (Verify, *sid*, vk', M, σ) from a verifier V
- If a [V.1] record (sigrec, vk', M, σ, f') exists then $f \leftarrow f'$
- Else
  - If $vk' = vk$ then $f \leftarrow 0$, else [V.2] $f \leftarrow \mathsf{Vfy}(vk', M, \sigma)$
  - Create a record (sigrec, vk', M, σ, f)
- Send (Verified, vk', M, σ, f) to V.

Refresh

On receiving (Refresh, *sid*, e) from server S_i
- Ignore the input if [Rf.1] $\mathsf{epoch}_i \neq e - 1$ or if $\mathtt{refreshing}_i = 1$
- Set [Rf.2] $\mathtt{refreshing}_i \leftarrow 1$
- If [Rf.3] $\mathtt{refreshing}_i = 1$ for all $i \in [n]$:
  - On [Rf.4] (Refresh, *sid*, e, S_i) from $\mathcal{A}$, set [Rf.5] $\mathsf{epoch}_i = e$ and send an immediate output (Epoch, *sid*, e) to S_i

On receiving (Refresh, *sid*, abort) from $\mathcal{A}$, set [Rf.6] $\mathtt{refreshing}_i = 0$ for all $i \in [n]$.

Figure 18: Functionality $\mathcal{F}_{\mathsf{psThrSig}}$ modeling proactively secure threshold signing. Main change from [5, 28] is that signatures are not adversarially-determined and hence can represent secret objects, such as tokens in our setting.

signature are not blind). To signal their participation in a signing sessions, servers can input ProceedSign with the corresponding subsession identifier. This allows to implement arbitrary rate-limiting policies on a per-request basis. As soon as the threshold of t participating servers (who all need to be in the same signing epoch) is reached [PS.1], $\mathcal{F}_{\mathsf{psThrSig}}$ [PS.3] generates a signature σ for message M using the Sign algorithm and *sk*, installs a [PS.4] signature record that will allow successful verification of σ, and [PS.5] outputs the signature to U.

Verification and revocation. Everyone can check validity of signatures σ on messages M under arbitrary verification keys vk'. If $vk' = vk$, i.e., verification is requested for the verification key associated with $\mathcal{F}_{\mathsf{psThrSig}}$, then the signature records are used to determine whether σ verifies or not: the output is set to true only if a record (sigrec, *vk*, M, σ, true) exists ([V.1]). Since such records only get created through successful signing requests above, **unforgeability** is enforced. For $vk' \neq vk$, $\mathcal{F}_{\mathsf{psThrSig}}$ uses the Vfy algorithm to determine the result ([V.2]). Allowing verification for "incorrect" public keys is necessary to avoid that $\mathcal{F}_{\mathsf{psThrSig}}$ implies a trusted certification authority.

Adversarial influence and leakage. The adversary [PS.2] can prevent signature generation by not sending sign-ok for the corresponding subsession. $\mathcal{F}_{\mathsf{psThrSig}}$ keeps the adversary from learning signatures generated by honest users ([PS.6]), making it a suitable building block for applications where the possession of signatures has a certain value.

Refresh. $\mathcal{F}_{\mathsf{psThrSig}}$ keeps track of which "key epoch" a server currently is in by maintaining [K.3,Rf.5] individual counters epoch_i. Individual counters allow to analyze protocols that admit asynchronous completion of the refresh phase, i.e., one server signs under the new keys already while another one is still waiting for the final message in a refresh protocol. Avoiding such situations would require costly methods to enforce synchrony, and we hence design $\mathcal{F}_{\mathsf{psThrSig}}$ with individual counters, to enable the analysis of more efficient SSO schemes where servers might be "off" by 1 regarding their epochs. Any server can indicate that it wants to move to the next epoch. The functionality acts only [Rf.1] if the server does not skip any epoch, and if it has not already started refreshing in the current epoch. As soon as all servers [Rf.3] joined the refreshing procedure in the current epoch, [Rf.5] the next epoch is entered and the servers are notified about it. The adversary can [K.3,Rf.4] delay completion of individual servers, and it can cause individual servers to [Rf.6] abort their ongoing refresh procedure.

E.1.1 Signature schemes realizing $\mathcal{F}_{\mathsf{psThrSig}}$. There already exists threshold signatures which are secure against a proactive adversary maliciously corrupting up to $t - 1$ parties in the UC-model for both ECDSA [28] and RSA [5]. Although the protocol by Almansa *et al.* requires $t = n$. Rabin [82] showed a proactively secure threshold signature scheme satisfying a game-based security notion for any $t < \lceil n/2 \rceil$. None of these schemes are suitable to instantiate $\mathcal{F}_{\mathsf{psThrSig}}$, because of the following reasons:
- All of [82] require all servers to participate in the signing procedure, while in our setting we require only t servers to be online. Although Rabin show how to handle missing or corrupt servers during signing; it comes at the cost of leaking their private key shares to the honest servers.
- [5] is secure against adaptive corruptions, but requires $t = n$. While in our setting we only consider adversaries who fix the corruption set at the start of each epoch, i.e. static corrupt. Yet we require a scheme which works for any $t < \lceil n/2 \rceil$.

- [5, 82] require the trusted generation of safe primes, which we ideally would like to avoid.
- [5, 28] are proven in a model where the adversary chooses how signatures look like (which is common in UC notions for signatures). Our setting however requires treating signatures as secret objects.
- [28], and threshold ECDSA protocols in general, have a significantly more expensive signing phase than their RSA-based counterparts due to the non-linear algebra in ECDSA signatures.

Our construction, Π_{psThrSig}, shown in Fig. 13 and Fig. 14 take departure in the proactively secure signature scheme of Rabin [82], which we now explain in a nutshell. First, during setup additive shares of the private key are generated, along with a witness for each share and then Feldman's verifiable secret sharing scheme (VSS) is used to threshold-reshare those additive shares. The additive shares are used to sign, along with zero-knowledge proofs proving that the shares used to do partial signing are consistent with the witness. If a party is unavailable or cheats, then the threshold shares are used to reconstruct the additive share of this party towards the remaining parties. This allows the other parties to still complete the signing phase. For refreshing, the additive shares are reshared additively, and proven consistent with the witness from setup. Again if a party is cheating or not participating, then the threshold shares are used to recover their additive share to allow the other parties to complete the refreshing.

We note that the shares in play becomes quite large, due to the need to emulate computation of a group of unknown by the use of the integers. However this does not pose a significant issue when using the threshold signatures in our ab-dSSO scheme since each server only need to participate in a *single* threshold signing scheme.

Our realization of $\mathcal{F}_{\mathsf{psThrSig}}$. Based on Rabin's scheme we first show that safe primes are not necessary for security. Then we modify the signing phase to only require t participating servers, by allowing it to abort in case any of them behave maliciously[8]. This allow us to skip the additive share part of Rabin's construction and directly generate and distribute threshold shares of the signing key.

Figure 13 and Fig. 14 present our threshold RSA signature scheme that realizes the functionality given in Fig. 18. The building block Feldman secret sharing, over $\mathbb{Z}_N$ due to usage of an RSA group, is depicted in Fig. 15 for completeness. We next comment on a subtlety that seem to have been overlooked in the literature regarding the verifiable VSS in $\mathbb{Z}_N$.

On the Verifiability of Feldman Secret-Sharing in $\mathbb{Z}_N$. We recap Feldman's verifiable secret sharing (VSS) scheme in Fig. 15, where Rabin claims [82, Theorem 1] that the proof of the verifiability property (as defined by Feldman and Micali [44]) of Feldman's original scheme [43] also applies to the adaptation of the scheme to $\mathbb{Z}_N$. The scheme in Fig. 15 can be claimed to be verifiable only if for a tuple $(A_0, \ldots, A_{t-1})$ broadcast during a sharing phase and t correct shares z_i, i.e., such that $g^{z_i} = \prod_{j=0}^{t-1} (A_j)^{i^j}$, the value s reconstructed from the t shares is such that $A_0 = g^{s \cdot n!}$, and there exists integers $a_1, \ldots, a_{t-1}$ such that $A_i = g^{a_i}$ and $P(i) = z_i$ for all $i \in [t-1]$.

[8]We note that the entire protocol does not abort, only the specific execution of the signing phase. The user can try again with a different set of server, but we do *not* identify the servers who are misbehaving.

However, the setting in the case of $\mathbb{Z}_N$ is different from Feldman's original proof [43] as there is no guarantee that a value A_i broadcast by a dealer during the sharing phase is in the subgroup $\langle g \rangle \subseteq \mathbb{Z}_N^*$ generated by g. Indeed, if the verifying parties are labeled $x_1, \ldots, x_t \in \{1, \ldots, n\}$, the verification equations can be written as

$$\begin{bmatrix} g^{z_1} \\ \vdots \\ g^{z_t} \end{bmatrix} = \begin{bmatrix} 1 & x_1 & \cdots & x_1^{t-1} \\ 1 & x_2 & \cdots & x_2^{t-1} \\ & & \vdots & \\ 1 & x_t & \cdots & x_t^{t-1} \end{bmatrix} * \begin{bmatrix} A_0 \\ \vdots \\ A_{t-1} \end{bmatrix},$$

with $*$ denoting the action of $\mathbb{Z}$ on $\mathbb{Z}_N^*$. Let V be the above Vandermonde matrix. Multiplying the equation on the left by $W := \mathrm{adj}(V)$, the adjugate matrix of V, one gets

$$W * \begin{bmatrix} g^{z_1} \\ \vdots \\ g^{z_t} \end{bmatrix} = \begin{bmatrix} A_0^{\det(V)} \\ \vdots \\ A_{t-1}^{\det(V)} \end{bmatrix}.$$

In other words, the verification equations only imply that $A_i^{\det(V)} \in \langle g \rangle$ for all $i = 0, \ldots, t-1$.

One way to enforce that each A_i is in $\langle g \rangle$ would be to have the dealer prove it in zero-knowledge to each party Σ, and a party in Σ would broadcast a complaint in case the proof fails. If at least t complaints are broadcast, then all honest parties reject their shares and the sharing phase is aborted. It is worth noting that running such zero-knowledge proofs would not greatly affect the practical efficiency of the SSO scheme as they would only occur during refreshing phases. Concretely it can be realized with constant group elements and exponentiations using the approach of Boneh *et al.* [15] in the generic group model, or with $O(s)$ overhead in the standard model using a Schnorr proof with a one-bit challenge.

Key Generation. During the key-generation phase, all n servers first generate an RSA modulus N and n-out-of-n additive shares $d_1, \ldots, d_n$ of the secret key (up to a public offset d_0 that is only useful to simplify the simulation). We abstract from this task through usage of a distributed key generation functionality, $\mathcal{F}_{\mathsf{DKG}}$, in Fig. 16. Protocols to UC securely realize it exist against a malicious adversary, e.g. [34]. Next, each server S_i computes t-out-of-n shares of their additive shares to enable threshold signing with new shares δ_i, and they all verify that the shared values indeed add up to $e^{-1} \bmod \varphi(N)$, with e the public RSA exponent. Each server S_i then distributes back-up shares of their t-out-of-n shares δ_i, and verifies that the shared values are consistent with the additive shares d_i. Finally static and public value u and v are computed from e and $(n!)^2$ using the extended Euclidean Algorithm, and stores them for later use during the signature computation. We note that for simplicity we assume in $\mathcal{F}_{\mathsf{DKG}}$, in Fig. 16, that e is prime; as this is seen in the literature [48] and ensures that $\gcd(e, (n!)^2) = 1$. In practice e is in fact picked as a prime, e.g. 3 or 65,537.

Signing and Verification. To sign a message M, each server simply computes a partial signature $H(M)^{\delta_i} \bmod N$ and sends it to the requesting user, along with additional parameters to the signature scheme, d_0, u, v, e, N. The user can then aggregate t such partial signatures to compute a full, standard RSA signature that can be verified in the classical way. In particular, the user takes advantage of u and v to remove $(n!)^2$ from the exponent, which arrives as

an artifact of the Feldman secret sharing over an RSA group. The removal of this term is needed, as otherwise the scheme would not allow signatures to be verified using the standard RSA verification algorithm. Note that the servers computing partial signatures are unaware of the other signing servers, which contrasts from the scheme from [5, 82], and that is precisely why the shares δ_i servers use to sign (and not just the back-up shares) are t-out-of-n shares and not n-out-of-n shares.

Refresh. At the beginning of each refresh phase, each server first compares the public part of its state with that of the other servers, since its state may have been tampered with, or even erased if the server is just now recovering from a malicious corruption. Assuming an honest majority, honest servers agree on those public values that they can use to also check if the share they hold is correct. If a server's private shares cannot be used to validate the public data, then it can broadcast a request to be sent the back-up shares of the secret-key share δ_i it computed in the last refresh protocol execution (or during key generation if this is the first refresh). Because the refresh protocol is aborted if any value sent is invalid, even if the server were corrupt during that last refresh execution, the key share computed was necessary valid since the protocol successfully terminated. Each server S_i proceeds by distributing t-out-of-n shares of δ_i, checks that the shared values are consistent with the ones computed in the previous refresh execution, and then computes from the values it received, a new share δ_i' for the next epoch. This phase ends with each servers distributing back-up shares of their new share δ_i' and verifying the consistency of the newly shared values with the one computed during the *last* refresh execution.

Theorem E.1. *Protocol Π_{psThrSig} in Fig. 13 and Fig. 14 securely realizes $\mathcal{F}_{\mathsf{psThrSig}}$ in Fig. 18 parametrized by non-distributed RSA algorithms* (KGen, Sign, Vfy)*, a computational security parameter λ, a threshold t in the $(\mathcal{F}_{DKG}, \mathcal{F}_{CT})$-hybrid model against a static, proactive and malicious adversary, assuming secure and server-side authenticated channels between users and each S_i, a broadcast channel among all S_i, and $n \geq 2t$, under the strong RSA assumption and the assumption that Feldman verifiable secret sharing is verifiable over $\mathbb{Z}_N$ [82] and $nN > 2^\lambda$.*

Proof Sketch of Thm. E.1. *Correctness.* Correctness of signing and verification follows the same arguments as previous works of proactively secure RSA using Feldman secret sharing [5, 82]. One difference is the lack of the use of value $\delta_{i,0}$ during refresh which we instead consider server S_i's own share. This does not affect correctness since it holds that $\delta_i = \sum_j \delta_{j,i}$.

Security. Most of the simulation proof follows previous approaches [5, 82], in particular when it comes to simulating Feldman verifiable secret sharing. Although our protocol is slightly different since we use t-out-of-n shares and t-out-of-n back-up shares of these shares, along with a slightly different robustness model. Hence we do some extra steps in the key generation phase, to ensure consistent sharing of back-up shares. The same is true in the refresh phase where we must validate and reshare or recover back-up shares.

For the key generation, we can generally not be sure that that each party correctly t-out-of-n shares their n-out-of-n share, received from $\mathcal{F}_{\mathsf{DKG}}$. However, because the shares should all add up to $d \equiv e^{-1} \mod \phi(N)$, it is sufficient to validate this by doing interpolation in the exponent, using the commitments, $A_{j,0}, \ldots, A_{j,n}$ to ensure that indeed d will have been shared. Next see that we validate the construction of t-out-of-n back-up shares based on an arbitrary subset of t shares. This might seem insufficient, but because we ensure the degree of the polynomial defined from the shares is $t-1$ (through $A'_{j,0}, \ldots, A'_{j,t-1}$), it is enough, since each server will validate their own share based these commitments.

When it comes refreshing, in situations where there is no need to use the recovery shares, then again the correctness follows from previous works, using the same approach as us. If there is a need to recover, i.e. when the adversary has modified the shares of a corrupt party, that then becomes honest again, then we notice that our proactive corruption model (Fig. 1) guarantees us that there will be an honest majority among all the servers *excluding* the server that tries to recover their shares. This is because a given choice of corruption by the adversary covers two contiguous refresh phases. Hence, if a corrupt party becomes honest at the beginning of a refresh phase, it would still "count" as a corrupt party in the previous corruption span. Thus, we can conclude that there must still be an honest majority of servers besides the one recovering. Thus, the recovering server can always identify the set of correct public values. This allows the server to correctly validate their shares using $A_{i,0}$, and in case of failure, receive t correct back-up shares and recover and validate their share δ_i. The last argument is a bit handwavy, but follows by the proof of Rabin [82], who showed that if N is of unknown factorization, then the Feldman VSS is sufficient hiding and binding of δ_i in the commitment $A_{i,0}$. Hiding follows since a view of $t-1$ shares ensures uniform probability view of other shares, since they are computed from polynomial interpolation of a degree $t-1$ polynomial of a secret with exponentially large entropy. Binding follows from the hardness of solving the discrete logarithm in an RSA group with base g (which is equivalent to factoring [7] which the RSA assumption reduces to). Thus validation can only pass if the initially committed shares were used or elements $A_{i,\cdot}$ outside the group of g were used such that the polynomial interpolation in the exponent when computing $A_{i,0}$ would thus as a minimum require the adversary to construct a commitment $A_{i,\cdot} = h^{\epsilon_i}$ where $\langle h \rangle \neq \langle g \rangle$ which is equivalent to the strong RSA assumption. Although this argument is not fully formal and closely relates to the paragraph on verifiability of Feldman secret sharing above.

In relation to the validation of back-up shares, we specifically notice that abort of the given refresh phase is allowed by the adversary. Hence an arbitrary set of t back-up shares is sufficient to try to reconstruct. Besides the different robustness and sharing model, our protocol has some main differences so it is important to argue that they do not compromise the overall security.

First of all, previous work [5, 82] requires N to be generated from safe primes. Since this makes secure and efficient realization of $\mathcal{F}_{\mathsf{DKG}}$ very difficult, we forgo that requirement. The only place where this requirement is used is to argue that the random element $g \in \mathbb{Z}_N^*$, used in the Feldman secret sharing, has exponentially large order. However, even if N is not constructed from two primes, if random elements do not have high order then they can be used

to break the RSA assumption. To see this consider the following observation:

Observation E.2. *If N is an RSA modulus, i.e. a product of two large primes, and e is the public exponent for RSA encryption, i.e. $\gcd(e, \phi(N)) = 1$, then a random element $g \leftarrow_R \mathbb{Z}_N^*$ has large order.*

Observation explanation. Let e be an RSA 'encryption exponent' for N such that $\gcd(e, \varphi(N)) = 1$. For an element $g \in \mathbb{Z}_N^*$ let d be the discrete logarithm of g to basis g^e, i.e., we have $(g^e)^d = g \bmod N$ and d is the 'decryption exponent' for g. If the order of g is not large, this logarithm can be found with Shank's BSGS algorithm (in time proportional to the square root of the order of g). Once d is found, it can be used to break all RSA challenges $m^e \mapsto m$ where m has the same (small) order as g. That is, the RSA assumption cannot hold if randomly picked elements $g \in \mathbb{Z}_N^*$ do not have large order. □

Next we argue that the lack of knowledge of the legitimate signature σ on M by the simulator is not an issue as long as we allow aborts and validate that a signature will verify before outputting it during the **Signing** phase. The simulator can validate each signature share σ_i that the adversary produced on behalf of each corrupt server, against the simulated shares of the secret key d the adversary was given during key generation for each corrupt server. If the set of malicious shares is inconsistent with the simulated shares then the simulator knows that it should abort the signing request since it will not become a valid signature (since the simulator knows the message m and RSA is a permutation). On the other hand, if the adversary actually knows the true signature σ, which happens when they corrupt the user U, then the simulator will also have received this signature from the ideal functionality and can thus simulate in the same manner of the previous works.

Simulation.

We show the simulation of Π_{psThrSig} in Fig. 19 and here sketch why an adversary who can distinguish between this and the real execution with non-negligible probability can break the RSA assumption (assuming that verifiability [44] of Feldman commitments in $\mathbb{Z}_N$ for RSA modulo N holds [82]).

For the simulation of KeyGen $\mathcal{S}$ can simulate $\mathcal{F}_{\mathsf{DKG}}$ and return $vk = (N, e)$ indistinguishable from the real execution since it relays share choices of $\mathcal{A}$ and $vk = (N, e)$ from $\mathcal{F}_{\mathsf{psThrSig}}$. However for future use $\mathcal{S}$ must simulate d_i for the honest parties. For all honest parties these are picked exactly according to $\mathcal{F}_{\mathsf{DKG}}$. However the special value d_0 is simulated using $d = 0$ instead of the true d, since it is unknown to $\mathcal{S}$. We observe that $d_0 \mod n \cdot N^2$ is uniformly random distributed in both the real world and the simulation. This is so since there is at least one honest party, S_i, for which variable $d_i \leftarrow_R [-n \cdot N^2, n \cdot N^2]$. Now because the true d is less than N and the simulated $d_0 = 0 - \sum_{i \in [n]} d_i$, we observe that the simulated d_0 will have a difference of at most N when compared with the true d_0. However, since $d_0 \mod n \cdot N^2$ will be uniformly random, this will be statistically indistinguishable if $nN > 2^\lambda$. This follows from noise drowning, resulting from least one honestly, randomly generated d_i.

Next see that the simulation of g will be indistinguishable from the true g, since it is simulated from a random $\hat{g}$, picked from the same distribution ($\mathbb{Z}_N^*$) as the true g, which follows from observation E.2. We then observe that $\hat{g} = g^d \bmod N$ since $g^{e \cdot d} \equiv g \bmod N$. Thus the shares generated in $\text{Feldman}_{\mathbb{Z}_N}$.Share will be indistinguishable from the true shares since Feldman verifiable secret sharing is hiding (and hence the fact that $d_{j'} = 0$ cannot be seen from the shares) and the commitment part of sharing will be simulated to be of the correct distribution for the *true* d, since $A_{j',0} = \hat{g}/g^{d_0 n!} \prod_{i \in [n] \setminus \{j'\}} A_{i,0} \bmod N$. More specifically the shares $\delta_{j',1}, \ldots, \delta_{j',n}$ from $\text{Feldman}_{\mathbb{Z}_N}$.Share will interpolate to 0, but when verifying each share, using $A_{j',0} \neq g^0 \mod N$, it will verify against whatever constant is in the exponent of $A_{j',0}$. While the simulator does not know the value of this constant, we see that $g^{d_0 \cdot n!} \prod_{j=1}^{n} A_{j,k} = g^{n!(d_0 + \sum_{j=1}^{n} d_j)} \bmod N = g^{n! d} \bmod N$ as in the real execution since $\hat{g} = g^{d^{-1} \mod \phi(N)} \mod N$. Finally since the generator d contains exponentially large in the security parameter, except with negligible probability, these values will also be statistically indistinguishable.

$\mathcal{S}$ runs the real protocol Π_{psThrSig} with several exceptions listed below. $\mathcal{S}$ maintains simulated instances of $\mathcal{F}_{\mathsf{CT}}$ (one) and $\mathcal{F}_{\mathsf{DKG}}$ (one) and follows their code depicted in Fig. 14 and Fig. 13.

On receiving (KeyGen, *sid*) **from** $\mathcal{F}_{\mathsf{psThrSig}}$**:**

- Emulate $\mathcal{F}_{\mathsf{DKG}}$ and simulate the honest parties by following the real protocol and extract p_i, q_i, d_i for S_i corrupted by $\mathcal{A}$, through their messages Shares and Proceed to $\mathcal{F}_{\mathsf{DKG}}$ and receive $vk = (N, e)$ from $\mathcal{F}_{\mathsf{psThrSig}}$.
- Simulate the honest parties' shares $d_i \leftarrow_R [-n \cdot N^2, n \cdot N^2]$ and set $d_0 = 0 - \sum_{i \in [n]} d_i \in [-n \cdot N^2 + N, n \cdot N^2]$.
- Emulate $\mathcal{F}_{\mathsf{CT}}$ by first sampling $\hat{g} \leftarrow_R \mathbb{Z}_N^*$ and then returning $g := \hat{g}^e \bmod N$.
- Run $\text{Feldman}_{\mathbb{Z}_N}$.Share for the honest servers based on the emulated values of d_i, except for one honest server $S_{j'}$ where $\mathcal{S}$ sets $d_{j'} := 0$ and when simulating $\text{Feldman}_{\mathbb{Z}_N}$.Share on 0 it sets commitment $A_{j',0} = \hat{g}/\left(g^{d_0 n!}\right) \prod_{i \in [n] \setminus \{j'\}} A_{i,0}\Big) \bmod N$ instead $A_{j',0} = g^0 = 1$.
- Simulate the rest of KeyGen using the emulated values.

On receiving (Sign, *ssid*, M) **from** $\mathcal{F}_{\mathsf{psThrSig}}$**:**

- If U is honest proceed as follows:
  - For each message $(ssid, N, e, d_0, u, v, \sigma_i)$ received from each corrupt S_i send to U.
  - Let C denote the set of indexes of corrupt servers and check that $\prod_{i \in [C]} \sigma_i = H(M)^{\sum_{i \in [C]} \delta_i} \bmod N$ for δ_i computed during the simulation of $\text{Feldman}_{\mathbb{Z}_N}$.Share in KeyGen above or refreshed in Refresh below, based on the extracted values of the corrupt parties and simulated values of the honest parties. If not, or if the public values N, e, d_0, u, v are inconsistent with the simulated ones in KeyGen, then abort at the point where (sign-ok, *sid*, *ssid*) is supposed to be sent to $\mathcal{F}_{\mathsf{psThrSig}}$, otherwise send (sign-ok, *sid*, *ssid*) to $\mathcal{F}_{\mathsf{psThrSig}}$.
  - Simulate the rest of Sign using the emulated values.
- If U is corrupt proceed as follows:
  - Simulate the honest parties by following the protocol, except for one honest party $S_{j'}$ where $\mathcal{S}$ computes $\sigma_{j'} = \sigma / H(M)^u \left(H(M)^{d_0 \cdot (n!)^2} \prod_{i \in [n] \setminus \{j'\}} H(M)^{\delta_i \cdot \omega_i}\right)^v \bmod N$ and sends $(ssid, N, e, d_0, u, v, \sigma_{j'})$ to U.

On receiving (Verify, *sid*, *vk*, M, σ) **from** $\mathcal{F}_{\mathsf{psThrSig}}$**:**

- Proceed as the real protocol with the simulated shared constructed in the previous phases.

On receiving (Refresh, *sid*, S_i) **from** $\mathcal{F}_{\mathsf{psThrSig}}$**:**

- Proceed as the real protocol with the simulated shared constructed in the previous phases.

Figure 19: Simulator Π_{psThrSig} for Theorem E.1.

The simulator then simulates the validation of the Feldman shares, just like the honest servers would. Due to the correctness of Feldman sharing, and the check that $A_0^e = g^{n!}$, if any malicious servers sends a commitment that is not consistent with their shares of d then the validation will fail and hence the simulation will abort. From this we also observe that the simulator will receive shares $\delta_{j,i}$ for every honest server S_i from each corrupt server S_j. Thus if the Feldman validation succeeds it implies that corrupt server S_i shared values consistent with d_i, and hence the simulator compute δ_i for all $i \in [n]$ by interpolating the malicious shares and combining them with the $\delta_{j,i}$ values it picked for the honest servers S_j. The rest of the simulation of KeyGen is then carried out using these emulated values.

For the **Signing** phase if U is honest, $\mathcal{S}$ receives (Sign, *sid*, *ssid*, M) from $\mathcal{F}_{\text{psThrSig}}$ and forwards (ProceedSign, *sid*, *ssid*) to $\mathcal{F}_{\text{psThrSig}}$ on behalf of the corrupted parties and extract the messages (*ssid*, N, e, d_0, u, v, σ_i) from the corrupted servers. It is easy to see that the **Signing** phase only completes if the corrupt servers send shares of the signature which together is the value "correctly" constructed using the simulated values δ_i. Because RSA is a permutation, there exist only one signature for the choice of message and keys. Hence any malicious change of the adversarial multiplicative share of the signature would cause a failure in the real world when signature is verified, in the same manner as in the simulation.

If U is corrupted, $\mathcal{S}$ extracts message (Sign, *sid*, *ssid*, M) and forwards it to $\mathcal{F}_{\text{psThrSig}}$ and in turn receives the message (Signature, M, σ) from the ideal functionality. Then note that, based on the simulated $\sigma_{j'}$ that $\sigma / \left(H(M)^u \left(H(M)^{d_0 \cdot (n!)^2} \prod_{i \in [n] \setminus \{j'\}} H(M)^{\delta_i \cdot \omega_i} \right)^v \right) \bmod N \equiv^s H(M)^{\delta_i}$. Even in the case where all servers S_i for $i \in [n] \setminus \{j'\}$ are corrupt and can choose non-random values of δ_i, the true value $\delta_{j'}$ is still sampled uniformly random. Hence the distributed of simulated $\sigma_{j'}$ will be indistinguishable from the real distribution since the distributions $H(M)^{\delta_{j'}} \bmod N$ except with negligible probability. See the proof of theorem 2 by Almansa *et al.* [5] for more detail.

For the **Verify** simulate in the straight forward manner by executing verification algorithm.

For the **Refresh** simulate in the straight forward manner using the arguments of VSS secret sharing. We first see that the adversary cannot force the honest servers to accept wrong public values, since there will be a majority of honest servers emulating the sharing of correct public values. Similarly we see that the check $g^{\delta_i \cdot n!} = A_{i,0}$ for honest server S_i will ensure that recovery steps are taken if the server has been corrupted and had their value δ_i changed.

For the resharing part of refresh the same argument applies. This implies that resharing will be consistent with the shares committed to previously by the honest servers. For the rest of the resharing phase the simulator validates the new shares of the corrupt parties against the shares they *should* hold. I.e. the values that the simulator originally picked for them, which it can do since it receives at least t shares of the corrupt servers' t-out-of-n sharing. If the shares are not correct then the simulator aborts this execution of the refresh phase. We now see that, using similar arguments as for the sharing in key generation, that the commitments are all consistent with previous shares, which the simulator either extracted from the initial corrupt servers, or defined for the honest servers.

When it comes to the general indistinguishability aspects of these distribution we note that it follows from the underlying security of the VSS secret sharing [43]. □

F FULL PROOF OF THEOREM 4.1

We prove the theorem using a sequence of games, starting with the real execution, parties running our ab-dSSO scheme and a (dummy) adversary, and ending with an ideal execution where parties hand their inputs to $\mathcal{F}_{\text{ab-dSSO}}$, which interacts with a simulator $\mathcal{S}$. The distinguishing environment is allowed to determine inputs to parties, see their outputs, and interact with the adversary. Let $\mathsf{win}(\mathbf{G}_x)$ denote the probability that the environment outputs 1 in game $\mathbf{G}_x$, over the random coins of all machines involved in the execution of the game.

Proof. **Game 1: The real execution.** In the real execution the parties run our ab-dSSO scheme, assisted by hybrid functionalities $\mathcal{F}_{\text{psThrSig}}$ and $\mathcal{F}_{\text{reqABB}}$.

Game 2: Abort upon MAC collision. We abort the protocol run in case of $\mathcal{A}$ substituting any value $\mathtt{type}_j$ in a message (*ssid*, uid, M, P, $(\mathtt{type}_j)_{j \in [m]}$) with $\mathtt{type}_{j'}$ and any S_i creating an output (Sign, *ssid*, ...). Any such server would use mac share $\beta_{i,\mathsf{uid},j'}$ instead of $\beta_{i,\mathsf{uid},j}$. The probability that $[v_{\mathsf{uid},j}] = [a_j] \cdot [\Delta] + [\beta_{\mathsf{uid},j}] \mod p$ is $1/p$ since $\beta_{\mathsf{uid},j}$ is uniformly random and information-theoretically hidden from the adversary who can corrupt at most $t-1$ servers. Hence, this and the previous game are indistinguishable except with probability bounded by the Birthday Bound, where Q_r is the number of registration queries.

$$|\Pr[\mathsf{win}(\text{Game 2})] - \Pr[\mathsf{win}(\text{Game 1})]| \leq \frac{Q_r \cdot m}{2 \cdot p}$$

Game 3: Abort upon MAC forgery. We abort the protocol execution whenever a corrupt party inputs a value $v_{\mathsf{uid},j}$ into $\mathcal{F}_{\text{reqABB}}$ that was never output by $\mathcal{F}_{\text{reqABB}}$, but which leads to a server S_i producing a corresponding (Sign, ...) output (i.e., $v_{\mathsf{uid},j}$ passes the mac verification step). Due to $\beta_{\mathsf{uid},j}$ being drawn uniformly random and information-theoretically hidden from the adversary who can corrupt at most $t-1$ servers, the probability that this happens is bound by $1/p$ and hence this and the previous game are indistinguishable except with probability bounded by the Birthday Bound, where Q_{ps} is the number of ProceedSign inputs to servers.

$$|\Pr[\mathsf{win}(\text{Game 3})] - \Pr[\mathsf{win}(\text{Game 2})]| \leq \frac{Q_{ps} \cdot m}{2 \cdot p}$$

Game 4: Add simulator and functionality machines. We regroup the real execution into one ITI called simulator or $\mathcal{S}$ ($\mathcal{S}$ now includes $\mathcal{A}$, $\mathcal{F}_{\text{psThrSig}}$, and $\mathcal{F}_{\text{reqABB}}$. We add an ITI $\mathcal{F}$ with interfaces Register, ProceedReg, Sign and Revoke of $\mathcal{F}_{\text{ab-dSSO}}$, modified s.t. $\mathcal{F}$ discloses attributes and proofs to $\mathcal{A}$ (we stress that we take away this knowledge from $\mathcal{S}$ in later games).

We augment the code of $\mathcal{S}$ to act when getting notifications from $\mathcal{F}$ or instructions from $\mathcal{A}$.

- Notification about input (Register, *ssid*, uid, $(\mathtt{type}_j)_{j \in [m]}$) from $\mathcal{F}$: simulate n messages (*ssid*, uid, $(\mathtt{type}_j)_{j \in [m]}$) from the user who provided this input, to every S_i.
- Notification about input (ProceedReg, *sid*, *ssid*, S_i) from $\mathcal{F}$: $\mathcal{S}$ simulates usage of $\mathcal{F}_{\text{reqABB}}$ by the honest servers, and

delivers a (ProceedReg, ...) output towards S_i as soon as that server has received the MAC share from $\mathcal{F}_{\text{reqABB}}$.
- Notification about input (Sign, *ssid*′, uid, $M, P, (\text{type}_j)_{j\in[m]}$) and consecutive message (Sign, *ssid*′, true) from $\mathcal{F}$: simulate message (*ssid*′, uid, $M, P, (\text{type}_j)_{j\in[m]}$ towards every S_i.
- As soon as a simulated S_i outputs (Registered, *ssid*, uid) $\mathcal{S}$ delivers the corresponding output of $\mathcal{F}$ to S_i.
- As soon as a simulated S_i outputs tuple (Sign, *ssid*, uid, M, P, $(\text{type}_j)_{j\in[m]}$) $\mathcal{S}$ delivers the corresponding output of $\mathcal{F}$ to S_i.
- If $\mathcal{A}$ delivers message (*ssid*, uid, $(\text{type}'_j)_{j\in[m]}$) to S_i then send (Register, *ssid*, $(\text{type}'_j)_{j\in[m]}, S_i$) to $\mathcal{F}$.

We let $\mathcal{S}$ extract all secrets from corrupted parties' inputs to $\mathcal{F}_{\text{reqABB}}$ and $\mathcal{F}_{\text{psThrSig}}$ and send the corresponding inputs to $\mathcal{F}$ on behalf of those corrupted parties. E.g., if a corrupt user U attempts to register by sending A, π to $\mathcal{F}_{\text{reqABB}}$ instance *ssid*, $\mathcal{S}$ sends (Register, *ssid*, uid, A, π) to $\mathcal{F}$ on behalf of U, where uid is extracted from A.

To argue indistinguishability, it is important to note that $\mathcal{F}$ does not yet verify signatures, and that the simulated protocol run uses the same inputs as in the previous game (since still in this game the simulator knows passwords, attributes, and proofs). Hence the correct content and scheduling of outputs of $\mathcal{F}$ such that they are indistinguishable from the real protocol execution is enough to argue perfect indistinguishability of this and the previous game. The scheduling is straightforward to verify given the list of $\mathcal{S}$'s actions above. The contents of KeyConf, Register, Registered, and Sign outputs are the same in this and in the previous game (i.e., the corresponding protocol parts are correctly doing what we expect them to do) as we argue below.

- Output (Register, *ssid*, uid, $(\text{type}_j)_{j\in[m]}$): uid and type values are either from the input or adversarial values, and the Register interface in $\mathcal{F}$ ensures that also adversarial values reach S_i even though they might not correspond to what the user sent.
- Output (Registered, *ssid*, uid): if such output is produced, then uid is guaranteed to match the user's input uid to Register since the user inputs A, which contains uid, into $\mathcal{F}_{\text{reqABB}}$ without the adversary being able to tamper. The registration protocol then lets the server check consistency of uid in the attribute set (using $\mathcal{F}_{\text{reqABB}}$, so in MPC) with uid received from the user, and aborts if they do not match. This behavior is the same in the Register and ProceedReg interfaces of $\mathcal{F}$, where bit b is set to 0 (yielding an abort) in case $\mathcal{A}$ tampers with uid. It remains to check whether, in both games, the output is produced in exactly the same cases. In the previous game, our ab-dSSO scheme produces such output only if registration is successfully concluded, namely if uid and types were not modified by the adversary, $\mathcal{F}_{\text{reqABB}}$ output got delivered by the adversary, uid is contained in A, and π verifies. These checks are carried out by setting bit b by $\mathcal{F}$ in this game, and hence this and the previous game do not differ in the cases where Registered output is produced.
- Output (Sign, *ssid*, uid, $M, P, b, (\text{type}_j)_{j\in[m]}$):
  - *Indistinguishability of the content of this output.* Both M and P from the previous game are as input by the user, since the user directly inputs them into $\mathcal{F}_{\text{psThrSig}}$ as message, and $\mathcal{F}_{\text{psThrSig}}$ does not allow the adversary to tamper with the signed message. In this game, $\mathcal{F}$ also does not allow tampering with M or P either, so these contents are equal in both games. For uid and the types, $\mathcal{A}$ can tamper with both our ab-dSSO scheme in the message sent from U to every S_i. However, tampering with uid results in S_i aborting the signing request due to a mismatch with the uid contained in A, so only a correct uid can be contained in the Sign output. Tampering with types in a way that is not detected our ab-dSSO scheme by a server is already exluded in Game Game 2. This behavior is reflected in this game in $\mathcal{F}$, where both uid and types are guaranteed to appear in the Sign output as they were input by the client.
  - *Indistinguishability of the occurrence of this output.* In the previous game, a server S_i our ab-dSSO scheme produces this output only if neither the user nor a server drops this query. A drop occurs at the user side if the policy is not fulfilled, or if the uid does not appear among the attributes, or if the user does not have a mac stored for some attribute type. On the server side, a drop occurs if there is a uid mismatch between the received message and the uid among the attributes stored in $\mathcal{F}_{\text{reqABB}}$, or if the user had input the wrong mac $v_{\text{uid},j}$ for any j (i.e., does not possess the claimed attribute), or if the policy does not verify. Note that it is crucial to repeat some checks on both sides as users can be corrupted and skip them. In $\mathcal{F}$, the corresponding checks are conducted by the signing interface, and output is only produced if all of them pass. The only difference appears in the check for a user possessing an attribute, which is mac verification in the protocol versus record existence in $\mathcal{F}$. Due to the unforgeability of verifying macs as of Game Game 3, the outcome of both checks are equal in this and the previous game.

We hence have

$$\Pr[\text{win(Game 4)}] = \Pr[\text{win(Game 3)}].$$

Game 5: Add the Refresh interfaces to $\mathcal{F}$. We add both Refresh interfaces to $\mathcal{F}$. We change $\mathcal{S}$ as follows. When a simulated server S_i successfully concludes the refresh protocol (i.e., holds the MAC keys for the next epoch e, and has received (Epoch, *sid*, e) from $\mathcal{F}_{\text{psThrSig}}$), $\mathcal{S}$ sends (Refresh, *sid*, e, S_i) to $\mathcal{F}_{\text{ab-dSSO}}$. In case any simulated server S_i runs into an abort, or $\mathcal{A}$ sends (Refresh, *sid*, abort), $\mathcal{S}$ sends (Refresh, *sid*, abort) to $\mathcal{F}_{\text{ab-dSSO}}$.

In case of a refresh, indistinguishability follows from the fact that the refresh interfaces of $\mathcal{F}_{\text{psThrSig}}$ and $\mathcal{F}_{\text{ab-dSSO}}$ are equivalent (i.e., the security of refresh of signature keys is outsourced to the proof of Theorem E.1), and the correctness of the resharing of all elements from $\{\Delta_i, \left\{\{\beta_{\text{uid},j}\}_{j\in[|A_{\text{uid}}|]}\right\}_{\text{uid}\in Y}\}$ in Fig. 6, which follows from the correctness of the VSS scheme in Fig. 10. We hence have

$$\Pr[\text{win(Game 5)}] = \Pr[\text{win(Game 4)}].$$

Game 6: Remove Feldman verification. Simulate the logic of Pedersen$^+$.Verify from Fig. 10 on (Verify, *sid*, $s_{j,i}, t_{j,i}, A_{j,0}, \ldots, A_{j,t-1}$) to accept if and only if these matches the values $\mathcal{S}$ constructed (for honest parties) or received (for corrupt parties) during one of the other Pedersen$^+$ macro calls (or the expected arithmetic computation of these). Concretely Verify is a local procedure that $\mathcal{S}$ simulates for the honest parties on values simulated (for honest

parties) or received from malicious parties, or publicly known combinations therefore. Hence the simulator knows which values are expected.

Similarly in Recover, we let the validation of $g^{s_i}h^{t_i} \neq A_{i,0}$ be replaced by the simulator validating if (s_i, t_i) are associated to the $A_{i,0}$ $\mathcal{S}$ constructed (for honest parties) or received (for corrupt parties) or combined.

Following security of Pedersen verifiable secret sharing [80], the only way to make this call output accept in this game for $s'_{j,i} \neq s_{j,i}$ and $t'_{j,i} \neq t_{j,i}$ is to solve the discrete log of h to base g. This means that the hybrids will always output the same unless the adversary can solve the discrete log problem.

Let $\mathsf{Adv}^{\mathsf{eDL}}(p^c, g, h)$ be the advantage of an adversary trying to solve the discrete logarithm problem of h with base g in the extension field $\mathbb{F}_{p^c}$. Furthermore let Q_{ref} be the number of `Refresh` queries and let Q_r be the number of `Register` queries, for which each user has at most m attributes.

$$|\Pr[\mathsf{win}(\text{Game } 6)] - \Pr[\mathsf{win}(\text{Game } 5)]| \leq \frac{3 \cdot (t-1) \cdot (Q_r \cdot m + 1) \cdot Q_{ref}}{2 \cdot p^c} + \mathsf{Adv}^{\mathsf{eDL}}(p^c, g, h)$$

Game 7: Remove MPC sharing. Simulate the protocol of Pedersen$^+$. MPC-Share from Fig. 10 by extracting s_i from $[s_i]$ (which is possible since the simulator simulates $\mathcal{F}_{\mathsf{reqABB}}$), remembering it (and the variable with which it is associated) and instead use $s_i = 0$ and simulate the protocol of MPC-Share as the real protocol using this.

Similarly we simulate the protocol of Pedersen$^+$.Reconstruct by returning the correct values s_i, t_i which were stored when executing Pedersen$^+$.MPC-Share if the values reconstructed by executing correctly matches the values used in MPC-Share. If instead s_i, t_i were created with Pedersen$^+$.Share then execute the protocol as normal. Note that the simulator can recognize the two cases from whether the t_i it restores is one it picked previously. Since they are picked over a computationally large domain, the chance of collision is negligible.

At any point where $\mathcal{F}_{\mathsf{reqABB}}$ is used, $\mathcal{S}$ will use the real, extracted s_i instead of 0 as given by the honest parties, thus the hybrids remain indistinguishable.

Let Q_r be the number of `Register` queries, for which each user has at most m attributes.

$$|\Pr[\mathsf{win}(\text{Game } 7)] - \Pr[\mathsf{win}(\text{Game } 6)]| \leq \frac{(t-1) \cdot (Q_r \cdot m + 1)}{2 \cdot p^c}$$

Game 8: Remove Feldman sharing. Simulate the protocol of Pedersen$^+$.Share from Fig. 10 by setting $s_i = 0$ and sampling t_i uniformly at random in situations where $t_i \neq 0$ and otherwise proceed as the protocol describes.

Since the adversary can at most corrupt $t-1$ servers, this remains indistinguishable from the previous hybrid since: 1) The values $s_{i,j}, t_{i,j}$ are generated from a random degree $t-1$ polynomial and $\mathcal{A}$ learns at most $t-1$ of these, thus they are indistinguishable from random. 2) A value $A_{i,j}$ is a statistically hiding commitment since the values $r_{i,k}$ are randomly sampled from $\mathbb{F}_{p^c}$. 3) The values used when calling Share are either known ($s_i = t_i = 0$) or random. See Refresh. More specifically s_i will be a share of G_i which is a random additive share for which more than $t-1$ are needed to recover the true value and any use of this will already be handled by the simulator when executing $\mathcal{F}_{\mathsf{reqABB}}$ as explained in the hybrid above.

Let Q_{ref} be the number of `Refresh` queries and let Q_r be the number of `Register` queries, for which each user has at most m attributes.

$$|\Pr[\mathsf{win}(\text{Game } 8)] - \Pr[\mathsf{win}(\text{Game } 7)]| \leq \frac{2 \cdot (t-1) \cdot (Q_r \cdot m + 1) \cdot Q_{ref}}{2 \cdot p^c}$$

Game 9: Abort upon forgery. We let the simulator abort in case $\mathcal{Z}$ inputs $(\mathtt{Verify}, M, P, \sigma, vk)$ to an honest party, where

- vk is equal to the verification key generated by $\mathcal{F}_{\mathsf{psThrSig}}$ within the `KeyGen` interface,
- $\mathcal{F}_{\mathsf{psThrSig}}$ does not have a record $(\mathtt{sigrec}, M, P, \sigma, vk, 1)$,
- and $\mathsf{Vfy}(vk, (M, P), \sigma) = 1$.

These conditions imply that σ was never created by $\mathcal{F}_{\mathsf{psThrSig}}$ using the Sign algorithm and sk and message (M, P). At the same time, sk is hidden within $\mathcal{F}_{\mathsf{psThrSig}}$ and hence the abort corresponds to $\mathcal{Z}$ holding a forgery. Indistinguishability from the previous game follows from the existential unforgeability of the signature scheme $(\mathsf{KGen}, \mathsf{Sign}, \mathsf{Vfy})$ that $\mathcal{F}_{\mathsf{psThrSig}}$ is parametrized with. The reduction is tight since there is only one signing key.

$$|\Pr[\mathsf{win}(\text{Game } 9)] - \Pr[\mathsf{win}(\text{Game } 8)]| \leq \mathsf{Adv}^{\mathsf{EUF-CMA}}_{\mathsf{SIG}}(1^\lambda)$$

Game 10: Let $\mathcal{F}$ compute and verify signatures. We add the `KeyGen`, `ProceedSign` and `Verify` interfaces to $\mathcal{F}$. We change $\mathcal{S}$ as follows.

- When $\mathcal{A}$ delivers $\mathcal{F}_{\mathsf{reqABB}}$ outputs during `KeyGen` towards S_i, $\mathcal{S}$ sends $(\mathtt{KeyConf}, sid, S_i)$ to $\mathcal{F}$.
- When $\mathcal{F}_{\mathsf{psThrSig}}$ produces output `Signature` towards a user, $\mathcal{S}$ sends the corresponding `sign-ok` message to $\mathcal{F}$.

It is straightforward to verify that the modifications to $\mathcal{S}$ in this game do not lead to any difference in *when* or *if* $\mathcal{Z}$ obtains a particular output. The only difference hence lies in the way signatures σ are computed and verified: in the previous game, signatures were computed and verified by $\mathcal{F}_{\mathsf{psThrSig}}$, and in this game they are computed and verified by $\mathcal{F}$. Both functionalities have equal `KeyGen`, `ProceedSign` and `Verify` interfaces (except that $\mathcal{F}$ includes information about attributes in records and messages to $\mathcal{A}$, which does not influence the `sigrec` records). In particular, both functionalities choose a single key pair for computing signatures. Indistinguishability hence follows from $\mathcal{F}_{\mathsf{psThrSig}}$ and $\mathcal{F}$ both using the very same algorithms $(\mathsf{KGen}, \mathsf{Sign}, \mathsf{Vfy})$, and it hence follows that

$$\Pr[\mathsf{win}(\text{Game } 10)] = \Pr[\mathsf{win}(\text{Game } 9)].$$

Quick orientation. We are at a point in the proof where we added all interfaces to $\mathcal{F}$. The only difference between $\mathcal{F}$ and our target functionality $\mathcal{F}_{\text{ab-dSSO}}$ is that $\mathcal{F}$ informs the simulator about proofs π and attribute sets A. Hence, it remains to take these items away from $\mathcal{S}$, in a way that is indistinguishable for the environment.

Game 11: Simulate without proofs. We modify $\mathcal{F}$ to not forward proofs π from `Register` inputs to $\mathcal{S}$. At the same time, we modify

$\mathcal{S}$ to work without these proofs: upon receiving Register for an honest user U, $\mathcal{S}$ chooses a dummy value $\pi_{\mathcal{S}}$ to input into $\mathcal{F}_{\mathsf{reqABB}}$ on behalf of U. During ProceedReg, when servers jointly compute $\mathsf{CAVfy}(vk_{\mathsf{CA}}, \cdot, [\pi_{\mathcal{S}}])$, $\mathcal{S}$ sets the output of this computation to be equal to bit b^{vf} received from $\mathcal{F}$'s Sign interface.

Indistinguishability follows from the fact that b^{vf} is computed in the very same way as our ab-dSSO scheme via $\mathcal{F}_{\mathsf{reqABB}}$ (i.e., in the previous game), namely as $\mathsf{CAVfy}(vk, A, \pi)$. Hence this and the previous game have equal output distributions towards $\mathcal{Z}$, i.e.,

$$\Pr[\mathsf{win}(\text{Game 11})] = \Pr[\mathsf{win}(\text{Game 10})].$$

Game 12: Simulate registration without the attributes. We modify $\mathcal{F}$ to not forward attributes from Registerinputs to $\mathcal{S}$ anymore. At the same time, we modify $\mathcal{S}$ to work without these attributes, and instead use the bit b^{uid} received from $\mathcal{F}$ during Register: $\mathcal{S}$ chooses dummy attributes to input into $\mathcal{F}_{\mathsf{reqABB}}$ on behalf of an honest user, and instead of letting a simulated user check whether $a_j = \mathsf{uid}$ through $\mathcal{F}_{\mathsf{reqABB}}$, $\mathcal{S}$ lets that user abort if $b^{\mathsf{uid}} = 0$.

Regarding indistinguishability, observe that our ab-dSSO scheme, the only purpose of honest users inputting their attributes into $\mathcal{F}_{\mathsf{reqABB}}$ is to (a) verify the proof, and (b) check whether uid is contained in the attributes. In this game, the binary outcomes of both these computations are replaced by equivalent information computed by $\mathcal{F}$, and hence the output distribution of this game is equal to the previous:

$$\Pr[\mathsf{win}(\text{Game 12})] = \Pr[\mathsf{win}(\text{Game 11})].$$

Game 13: Simulate signing without the attributes. We modify $\mathcal{F}$ to not forward attributes from Sign inputs to $\mathcal{S}$ anymore. At the same time, we modify $\mathcal{S}$ to work without these attributes, and instead use the bits $b^{\mathsf{pol}}, b^{\mathsf{uid}}, b^{\mathsf{type}}$ received from $\mathcal{F}$ during Sign, very similar to the previous game: $\mathcal{S}$ uses dummy attributes as input to $\mathcal{F}_{\mathsf{reqABB}}$ and substitutes $\mathcal{F}_{\mathsf{reqABB}}$'s attribute-dependent computations with the received bits from $\mathcal{F}$. That is, $\mathcal{S}$ aborts upon wrong uid if $b^{\mathsf{uid}} = 0$. $\mathcal{S}$ lets a simulated party ignore a Sign input if the corresponding b^{pol} is 0, or if b^{type} is 0.

The indistinguishability argument is as in the previous game, and we have

$$\Pr[\mathsf{win}(\text{Game 13})] = \Pr[\mathsf{win}(\text{Game 12})].$$

We now have $\mathcal{F} = \mathcal{F}_{\text{ab-dSSO}}$ and hence Game 13 is equal to the ideal execution with $\mathcal{F}_{\text{ab-dSSO}}$ and a simulator $\mathcal{S}$ that we built gradually during the sequence of games. It is depicted in Fig. 20. This concludes the proof.

□

$\mathcal{S}$ runs the real protocol with several exceptions listed below. $\mathcal{S}$ maintains simulated instances of $\mathcal{F}_{\mathsf{reqABB}}$ (many) and $\mathcal{F}_{\mathsf{psThrSig}}$ (one) and follows their code depicted in Fig. 11 and Fig. 18.

On receiving (KeyGen, *sid*, S_i) **from** $\mathcal{F}_{\textbf{ab-dSSO}}$**:**
- When $\mathcal{A}$ delivers $\mathcal{F}_{\mathsf{reqABB}}$ outputs during KeyGen towards server S_i, $\mathcal{S}$ sends (KeyConf, *sid*, S_i) to $\mathcal{F}_{\text{ab-dSSO}}$ (Game 10).

On receiving $(\mathtt{Register}, ssid, \mathsf{uid}, (\mathsf{type}_j)_{j\in[m]}, U)$ and subsequently $(\mathtt{Register}, ssid, b^{\mathsf{vf}}, b^{\mathsf{uid}})$**from** $\mathcal{F}_{\textbf{ab-dSSO}}$**:**
- If $b^{\mathsf{uid}} = 0$ then ignore the query. (Game 12)
- Otherwise, do:
  - If U is corrupt and subsequently sends π and $(a_j)_{j\in[m]}$ to $\mathcal{F}_{\mathsf{reqABB}}$, send input $(\mathtt{Register}, ssid, \mathsf{uid}, (\mathsf{type}_j : a_j)_{j\in[m]}$ to $\mathcal{F}_{\text{ab-dSSO}}$ on behalf of the corrupt U (Game 4).
  - If U is honest, (Game 11) pick random proof $\pi_{\mathcal{S}}$ and (Game 12) random attribute set $A_{\mathcal{S}}$ of length m, and start the simulated U with $\pi_{\mathcal{S}}, A_{\mathcal{S}}$. (Game 4)

Upon $\mathcal{A}$ **sending message** $(ssid, \mathsf{uid}, (\mathsf{type}_j)_{j\in[m]})$ to S_i on behalf of U, $\mathcal{S}$ sends a corresponding Register message to $\mathcal{F}_{\text{ab-dSSO}}$ (Game 4).

On receiving (ProceedReg, *ssid*, S_i) **from** $\mathcal{F}_{\textbf{ab-dSSO}}$**:**
- If a simulated server S_i outputs $(\mathtt{Registered}, ssid, \mathsf{uid})$, $\mathcal{S}$ delivers the corresponding output of $\mathcal{F}_{\text{ab-dSSO}}$ to S_i (Game 4).

On receiving $(\mathtt{Sign}, ssid', \mathsf{uid}, M, P, (\mathsf{type}'_j)_{j\in[m']}, U)$ and consecutively $(\mathtt{Sign}, ssid', b^{\mathsf{pol}}, b^{\mathsf{uid}}, b^{\mathsf{type}})$ **from** $\mathcal{F}_{\textbf{ab-dSSO}}$**:**
- If U is honest and $b^{\mathsf{pol}} = b^{\mathsf{uid}} = b^{\mathsf{type}} = 1$, then pick a random attribute set $A_{\mathcal{S}}$ of length m' and run the simulated U with $A_{\mathcal{S}}$ (Game 13). Otherwise ignore the query.

On receiving $(\mathtt{sign\text{-}ok}, ssid')$ from $\mathcal{F}$:
- When $\mathcal{F}_{\mathsf{psThrSig}}$ produces output $(\mathtt{Signature}, ssid', *)$ to a user U, $\mathcal{S}$ sends $(\mathtt{sign\text{-}ok}, ssid')$ to $\mathcal{F}$.

Figure 20: Simulator for Theorem 4.1.