Private Access Control for Function Secret Sharing

Sacha Servan-Schreiber
MIT CSAIL

Simon Beyzerov
MIT PRIMES

Eli Yablon
MIT PRIMES

Hyojae Park
MIT PRIMES

***Abstract*—Function Secret Sharing (FSS; Eurocrypt 2015) allows a dealer to share a function f with two or more evaluators. Given secret shares of a function f, the evaluators can locally compute secret shares of $f(x)$ for any input x, without learning information about f in the process.**

In this paper, we initiate the study of access control for FSS. Given the shares of f, the evaluators can ensure that the dealer is authorized to share the provided function. For a function family $\mathcal{F}$ and an access control list defined over the family, the evaluators receiving the shares of $f \in \mathcal{F}$ can efficiently check that the dealer knows the access key for f.

This model enables new applications of FSS, such as: (1) anonymous authentication in a multi-party setting, (2) access control in private databases, and (3) authentication and spam prevention in anonymous communication systems.

Our definitions and constructions abstract and improve the concrete efficiency of several recent systems that implement ad-hoc mechanisms for access control over FSS. The main building block behind our efficiency improvement is a discrete-logarithm zero-knowledge proof-of-knowledge over secret-shared elements, which may be of independent interest.

We evaluate our constructions and show a 50–$70\times$ reduction in computational overhead compared to existing access control techniques used in anonymous communication. In other applications, such as private databases, the processing cost of introducing access control is only 1.5–$3\times$, when amortized over databases with 500,000 or more items.

1. Introduction

Function secret sharing (FSS) [6, 8] is at the core of many privacy-preserving systems, including private databases [14, 15, 40], private telemetry [5], privacy-preserving machine learning [30, 33], distributed oblivious RAM (ORAM) [18], anonymous communication [13, 19, 31, 39], and efficient multi-party computation [9]. Since these applications involve the processing of private user data, often in settings where users may be behaving maliciously, access control becomes an important problem [19, 31, 39]. For example, in applications of FSS that involve privately reading from—or writing to—a database [5, 18, 19, 31, 39], access control is necessary to prevent malicious users from accessing or overwriting data belonging to other users.

FSS lets any user (called the *dealer*) distribute succinct secret shares of a function to a set of function evaluators. These evaluators can efficiently—and non-interactively—evaluate the function on a common input x to obtain secret shares of $f(x)$. FSS guarantees that the function remains private to strict subsets of the evaluators, which means that the evaluators do not learn anything about f (beyond the function family that f belongs to).

In this paper, we investigate the problem of privately enforcing access control in the context of FSS. We identify several existing applications of FSS that construct different ad-hoc solutions for access control [19, 31, 39], highlighting the utility of formally studying this paradigm.

For example, FSS is often used for private information retrieval (PIR) [6, 11, 22, 40]. In PIR, a dealer secret shares a selection function f_i with the evaluators. The evaluators use the shares to evaluate f_i on a database $\mathcal{DB}$ and send back shares $[f_i(\mathcal{DB})]$, which encodes the ith item in the database. The dealer then recovers the ith item by combining the returned shares. Importantly, the evaluators who are given shares of f_i learn nothing about f_i (beyond the fact that f_i is from the "selection function" family) and therefore do not learn which item the dealer retrieved from $\mathcal{DB}$.

In the PIR setting, access control could require that only users with an *access key* for the ith item in the database can successfully share f_i with the evaluators. More specifically, in applications involving e-commerce [25, 26], web queries [40], and media consumption [24], where users are only entitled to retrieve some (but not all) items in the database, such access control is imperative. Likewise, in private information *writing* applications, such as anonymous communication systems [19, 31, 39] and private telemetry [5, 12], access control is crucial to prevent attacks by malicious users sending invalid writes (e.g., by overwriting mailboxes of honest users [19, 31, 39]). Only users with a valid access key for the ith database row should be able to write to it.

Defining the problem of private access control. We model access control as a one-to-one mapping between functions and keys. Each function (in a family of functions) is mapped to a verification key and an access key. The evaluators hold the verification keys and a dealer has one or more access keys (we discuss key distribution in Section 3.4). A dealer secret shares the function f_i through FSS and, using the corresponding access key, provides a proof π proving access rights to f_i under some subset of verification keys. The evaluators (whom we also call the *verifiers*) can check the proof π using the verification keys (without learning which keys were used) and decide whether or not the dealer is entitled to an evaluation of the function f_i. For example, in the PIR setting, knowledge of the access key for the

selection function f_i, allows a user to distribute secret shares of f_i to the evaluators along with a proof π. The evaluators check π before evaluating the function to ensure that the user is entitled to retrieve the ith item (without learning i).

Challenges. Privately enforcing access control over a secret-shared function is challenging. As mentioned above, the evaluators are oblivious to the function they are evaluating, which bars obvious approaches to access control. That is, access control must maintain the privacy of the function (see Section 1.1). Additionally, FSS is concerned with *efficiency* (computation and communication overheads for the dealer and the evaluators). As such, the access control mechanism must preserve the efficiency guarantees of the FSS scheme. Finally, it is important to consider *malicious* evaluators that may try to exploit the access control mechanism to violate privacy (this is a problem when designing *any* form of verification over FSS [5, 8, 16]). Preventing malicious evaluators from violating privacy, without relying on strong assumptions or inefficient solutions, can be difficult [4, 8, 16].

Goals. We identify *efficiency* and *malicious security* as the primary goals when modeling and designing access control for FSS. More specifically—and following the requirements of FSS—we will require *communication efficiency* and *minimal interaction* between verifiers. Our definition (described in Section 3) captures these efficiency requirements by demanding (1) succinct proofs, (2) no interaction with the prover, and (3) at most *one* message exchanged between verifiers to check access rights (note that this is in fact optimal as it is necessary to exchange one message to verify any proof over secret-shares [4]). Importantly, by minimizing interaction between verifiers, we also obtain security against any subset of malicious verifiers. More concretely, (3) ensures that any construction satisfying our model never provides "feedback" to any subset of malicious verifiers which, in turn, ensures that malicious verifiers obtain no information through the access control mechanism.

1.1. Background on FSS

FSS [6] takes a different approach to "traditional" secret sharing of data. With traditional secret sharing, a dealer shares a value v with a set of s parties such that (1) knowledge of up to some threshold number of shares does not reveal any information on v and (2) shares can be efficiently recombined to recover v. FSS applies the same idea to *functions* where the dealer instead secret shares a description of a function $f : \{0,1\}^n \to \{0,1\}^*$ with a set of s evaluators. Denote the shares of f as $[f]$. The evaluators can then locally compute shares $[y] := [f(x)]$ on any input x. Informally, FSS must satisfy three properties:

- **Correctness.** The jth evaluator can evaluate their secret share of f on an input x to obtain a secret share of $f(x)$.
- **Privacy.** A secret share $[f]_i$ reveals no information on f.
- **Efficiency.** The secret shares $[f]$ are succinct (sublinear in the size of the truth table for f).

Boyle et al. [6, 8] provide constructions for several function families. Specifically, they describe efficient FSS constructions for NC^0 functions, constant conjunction search queries, and interval functions. Their constructions are based only on the assumption that one-way functions exist [6]. The main primitive behind their results is an FSS family for distributed point functions (DPFs) [6, 8, 22]. Subsequent work of Boyle et al. [8] extends DPFs to FSS for *decision trees* and products of distinct secret-shared functions. FSS schemes from stronger cryptographic assumptions yield constructions for *all* efficient function families [6, 7, 17].

1.2. Prior work

Express [19], Spectrum [31], and Sabre [39] use FSS for anonymous communication. In these systems, users privately write messages into mailboxes using a DPF.

To prevent malicious users from corrupting mailboxes belonging to honest users, all three systems require a form of access control applied over DPFs, which they enforce through a lightweight multi-party computation protocol.

The access control mechanism in Express associates each mailbox with a secret "address." The evaluators keep the addresses secret. Only a dealer with knowledge of a mailbox address can successfully write to that mailbox.

Unfortunately, the mechanism used in Express has several drawbacks: (1) it does not generalize to larger families of DPF and currently remains specific to the two-party DPF construction for point functions [6, 8], (2) the use of "addresses" for access control requires a large output range and leads to a $5\times$ computational overhead for the evaluators (and, more importantly, prevents optimization techniques for DPFs [8]), and (3) the multi-party computation requires extra communication between evaluators and the dealer. In contrast, our model and constructions (Sections 4 and 5) require only one message exchanged between verifiers and no interaction with the dealer. Additionally, our constructions make minimal assumptions on the underlying DPF scheme, making them compatible with DPF optimizations [8, 16].

Sabre extends Express by developing a different access control mechanism using zero-knowledge proofs over secret shares. Their techniques reduce the computational overhead on the evaluators (especially in the context of anonymous communication where many users are assumed to be malicious) at the cost of significantly increasing communication between the dealer and the evaluators. Like Express, Sabre is designed around two-party DPF constructions [6, 8] and requires a round of interaction between the evaluators to enforce access control. The access control mechanism of Sabre is actually a special case of a generic approach to access control realized via zero-knowledge proofs over secret-shares, which we describe in Section 7. However, their techniques are tailored to the anonymous communication setting where (1) a large number of users are assumed to be malicious and (2) where access control is verified in batches.

Spectrum provides yet a different technique for access control via secret-shared hashing. The idea behind Spectrum is to have the evaluators "hash" the value being written to

each mailbox using a unique hash key associated with the mailbox. The evaluators only process writes from a dealer that can prove, in zero-knowledge, that it knows the resulting hash value (which, in turn, proves that the dealer knows the hash key of the mailbox). The technique is communication efficient and only requires one message exchanged between verifiers to enforce access control (which aligns with our modelling of access control).

In Section 4, we start by abstracting the access control construction of Spectrum. We then generalize it further and develop new techniques to realize more efficient private access control schemes. In Section 8, we show that our improved constructions reduce the computational overhead by 50–70× in both Express and Spectrum and have 1,000× smaller proof sizes compared to Sabre.

1.3. Contributions

We make the following five contributions:

- A model for Private Access Control Lists (PACLs) for FSS. Our definitions capture the functionality requirements of several existing applications of access control for FSS [19, 31, 39] and demand a stringent set of efficiency requirements that align closely with the goals of FSS.
- PACL constructions for black-box DPFs and lightweight FSS classes derived from DPFs. Our constructions are secure against malicious provers, guarantee privacy in the face of malicious verifiers, have a constant-factor overhead (relative to sharing and evaluating the function), and can be used as drop-in replacements in existing applications for significant efficiency improvements.
- For the special case of *verifiable* FSS [16], we construct an optimized public-key PACL for black-box verifiable DPFs (which gives rise to PACLs for a large class of verifiable FSS). For this construction, we develop a zero-knowledge proof of discrete-logarithm knowledge over *secret-shared* group elements. Our construction has 2,400× smaller proof sizes compared to a naïve approach and is possibly of independent interest.
- PACLs for FSS for functions in $P/poly$ (not just classes of FSS derived from DPFs). Our generic construction is based on non-interactive zero-knowledge proofs over secret shares, instantiated in the random oracle model.
- An open-source implementation which we evaluate on several canonical applications, such as anonymous user authentication in a distributed setting, access control in private databases, and anonymous communication.

2. Overview

Here, we define **non-private** access control for functions. We define **private** access control for FSS in Section 3.

2.1. Access Control Lists (ACLs)

We define ACLs from a cryptographic lens in order to facilitate the definitions of *private* ACLs, which we introduce in Section 3. Specifically, we define ACLs as a set of access and verification keys associated with a collection of objects (in our case, functions).

Definition 2.1 (Access Control Lists)**.** Let $\lambda \in \mathbb{N}$ be a security parameter, $\mathcal{F}$: $\{0,1\}^n \to \{0,1\}^*$ be a function family, and $f_i \in \mathcal{F}$. An ACL scheme consists of an access control list Λ_λ (parameterized by λ) containing verification keys and an efficiently computable predicate $\mathsf{CheckAccess}(\Lambda_\lambda, f_i, \mathsf{sk})$ that outputs yes if and only if the access key sk satisfies a relation R with respect to the verification key associated with f_i in Λ_λ. For notational convenience, we let $N := |\mathcal{F}|$ and omit the λ subscript when it is clear from context.

We note that Definition 2.1 is general and not specific to FSS (indeed, Definition 2.1 does not even capture the notion of secret shares or distributed verifiers). We will define these notions in Section 3 when formalizing private ACLs for FSS. It is also natural to equip Definition 2.1 with *completeness* and *soundness* properties (with respect to an adversary). These are likewise deferred to the formalization of private ACLs in Definitions 3.1 and 3.2.

We now describe instantiations of CheckAccess from Definition 2.1. We will port these to private ACLs for FSS in Section 3. We observe that, in most cases, the goal of CheckAccess is to check if the provided access key matches with some unique verification key associated with the function f_i. We call this the *match predicate*. However, it is also possible that a function is associated with multiple different verification keys. Such a predicate is especially useful for maintaining efficient access key revocation in a setting with many users. A key can be removed for a given function *without* impacting the validity of the remaining keys. In this scenario, we instantiate CheckAccess as an *inclusion predicate* over a set of verification keys associated with the function.

Match predicate. The *match* predicate consists of an efficiently computable relation $R(\cdot,\cdot)$, N verification $(\mathsf{vk}_1, \ldots, \mathsf{vk}_N)$ and access $(\mathsf{sk}_1, \ldots, \mathsf{sk}_N)$ keys, such that $R(\mathsf{vk}_i, \mathsf{sk}_j) = 1$ if and only if $i = j$ and each tuple is uniquely associated with a canonical instance of $f_i \in \mathcal{F}$. CheckAccess is defined as:

$\mathsf{CheckAccess}(\Lambda_\lambda, f_i, \mathsf{sk})$

parse $\Lambda_\lambda = (\mathsf{vk}_1, \ldots, \mathsf{vk}_N)$

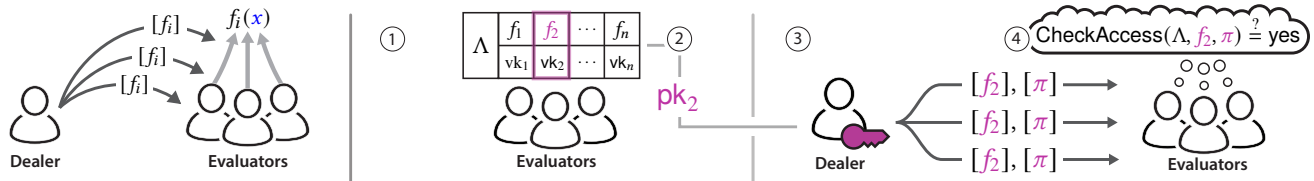
if $R(\mathsf{vk}_i, \mathsf{sk}) = 1$ **return** yes, **else return** no

That is, CheckAccess outputs yes if and only if the provided sk is related to the verification key associated with f_i.

Inclusion predicate. A generalization of the match predicate satisfying Definition 2.1 is the *inclusion* predicate that associates each function with $\ell \geq 1$ verification keys,

$$\Lambda_\lambda := \begin{pmatrix} (\mathsf{vk}_{1,1} & \ldots & \mathsf{vk}_{1,\ell}) \\ & \vdots & \\ (\mathsf{vk}_{N,1} & \ldots & \mathsf{vk}_{N,\ell}) \end{pmatrix}.$$

Any key in the row $(\mathsf{vk}_{i,1} \ldots \mathsf{vk}_{i,\ell})$ can be used to satisfy the relation for f_i. CheckAccess is defined as:



Baseline FSS w/o Access Control
A dealer privately shares a function with a set of evaluators; they can then evaluate the secret-shared function on a common input x to obtain shares of $f(x)$.

FSS w/ Access Control
Evaluators store a PACL for a function family; each public verification key is paired with an access key, which a dealer must know to share the corresponding function.

FSS w/ Access Control
Using the access key, a dealer can provide secret shares of the function f and proof π proving access rights to f. The evaluators collectively enforce access control by checking that π is accepting without learning anything about f in the process.

Figure 1: Overview of FSS and our access control model. **Left:** In FSS *without* access control, a dealer distributes shares of a function f_i with the evaluators. ① A setting *with* access control where the evaluators have Λ and ② the dealer has an access key pk_2 for f_2. ③ the dealer (acting as the prover) with knowledge of the access key for f_2 distributes secret shares of the function f_2 and proof shares π to the evaluators. ④ The evaluators collectively check access rights to f_2 using the proof shares $[\pi]$ and the access control list Λ.

$\mathsf{CheckAccess}(\Lambda_\lambda, f_i, \mathsf{sk})$

parse $\Lambda_\lambda = \big((\mathsf{vk}_{1,1}, \ldots, \mathsf{vk}_{1,\ell}), \ldots, (\mathsf{vk}_{N,1}, \ldots, \mathsf{vk}_{N,\ell})\big)$
if $\exists \mathsf{vk}_{i,j}$ such that $R(\mathsf{vk}_{i,j}, \mathsf{sk}) = 1$ **return** yes,
else return no

That is, CheckAccess outputs yes if and only if sk matches with *any* of the verification keys associated with f_i.

Boolean predicate. A generalization of the inclusion predicate satisfying Definition 2.1 is a monotone boolean predicate (ANDs and ORs) over a list of CheckAccess outputs. Here, Λ_λ consists of ℓ sublists $\Lambda_\lambda^{(1)}, \ldots, \Lambda_\lambda^{(\ell)}$ and a predicate P defined over the bits $b_i \leftarrow \mathsf{CheckAccess}_i(\Lambda_\lambda^{(i)}, f_i, \mathsf{sk})$ for $i \in \{1, \ldots, \ell\}$. $\mathsf{CheckAccess}(\Lambda_\lambda, f_i, \mathsf{sk})$ outputs yes if and only if $P(\mathsf{sk}, b_1, \ldots, b_\ell) = 1$.

2.2. Notation and cryptographic preliminaries

Notation. We use $x \leftarrow \mathsf{Alg}$ to denote assignment from a possibly randomized algorithm Alg and $x \leftarrow_R D$ to denote a random sample from a distribution D. We denote linear secret shares of x (resp. function secret shares of f) as $[x]$ (resp. $[f]$) and $[x]_i$ (resp. $[f]_i$) as the ith secret share in the set of shares encoding x (resp. f). We say an algorithm is *efficient* if it runs in probabilistic polynomial time.

Linear Secret Sharing. A linear secret-sharing (LSS) scheme [36] consists of two (possibly randomized) algorithms $\mathsf{Share}_{(\mathbb{F},t,s)}$ and Recover. Share generates s shares of a secret value in the field $\mathbb{F}$ such that (1) any subset of t or more shares can be combined using the linear function Recover to reveal the encoded value in the field $\mathbb{F}$, (2) no subset of fewer than t shares provides any information on the secret, and (3) shares can be added together to obtain a new share encoding the sum of the secrets.

Remark 1. A consequence of the linearity of Recover is that it can be evaluated "in the exponent" of a group. That is, given $g^{[v]_1}, \ldots, g^{[v]_t}$, it is possible to efficiently compute $g^v := g^{\mathsf{Recover}([v]_1, \ldots, [v]_t)}$. For simplicity of notation, we define ExpRecover: $\mathbb{G}^t \to \mathbb{G}$ to be the efficiently computable function which takes as input $(g^{[v]_1}, \ldots, g^{[v]_t})$ and outputs g^v with $v := \mathsf{Recover}([v]_1, \ldots, [v]_t)$.

Discrete logarithm problem and assumption. Let $\lambda \in \mathbb{N}$ be a security parameter. For a cyclic group $\mathbb{G}$ of prime order $p = p(\lambda)$ with generator g, the Discrete Logarithm (DL) assumption states that no efficient algorithm $\mathcal{A}$ can find $x \in \mathbb{Z}_p$ satisfying $y = g^x$ for a uniformly random $y \in \mathbb{G}$ [28].

Function Secret Sharing. FSS is a generalization of LSS; rather than secret-sharing a *value*, FSS captures the notion of secret sharing *functions*.

Definition 2.2 (FSS [6])**.** Let $2 \le t \le s$ be integers and $\mathcal{F} : \{0,1\}^n \to \{0,1\}^*$ be a family of functions and let $N = |\mathcal{F}|$. A (t, s)-FSS scheme for $\mathcal{F}$ consists of efficiently computable (possibly randomized) algorithms Gen and Eval with the following syntax:

- $\mathsf{Gen}(1^\lambda, f) \to (\kappa_1, \ldots, \kappa_s)$. Takes as input a security parameter 1^λ and function $f \in \mathcal{F}$. Outputs s evaluation keys $\kappa_1, \ldots, \kappa_s$.
- $\mathsf{Eval}(\kappa_i, x) \to [y]_i$. Takes as input an evaluation key κ_i and $x \in \{0,1\}^n$. Outputs secret share $[y]_i$.

The functionality must satisfy the following properties:

- **Correctness.** A (t, s)-FSS scheme is correct if for all subsets $I \subseteq \{1, \ldots, s\}$ where $|I| \ge t$, there exists an efficient output decoder Decode such that for all $f \in \mathcal{F}$:
$$\Pr\left[\begin{array}{l}(\kappa_1, \ldots, \kappa_s) \leftarrow \mathsf{Gen}(1^\lambda, f) : \\ \mathsf{Decode}(\{[y]_i \leftarrow \mathsf{Eval}(\kappa_i, x) \mid i \in I\}) = f(x)\end{array}\right] = 1.$$
- **Privacy.** For all $I \subset \{1, \ldots, s\}$ subset of indices such that $|I| < t$, let D_I be the distribution over $\{\kappa_i \mid i \in I\}$ where κ_i is sampled according to $\mathsf{Gen}(1^\lambda, f)$. A (t, s)-FSS scheme $(\mathsf{Gen}, \mathsf{Eval})$ is private if there exists an efficient simulator $\mathcal{S}$ such that $D_I \approx_c \mathcal{S}(1^\lambda, I)$.
- **Efficiency.** A (t, s)-FSS scheme is efficient if (1) each key κ_i is at most $O(\lambda N^\epsilon)$ in size, for any $\epsilon < 1$ (possibly dependent on n) and (2) Decode runs in time $O(\lambda s)$.

Following Boyle et al. [6], we assume Decode is a linear function of the inputs and therefore let Decode := Recover. As such, we also write $[f]_i$ to denote the ith FSS key κ_i and $[f(x)]_i$ to denote the ith share of the evaluation $\mathsf{Eval}(\kappa_i, x)$.

3. Private Access Control Lists

In this section, we formalize the notion of private ACLs applied to FSS (Definition 2.2). A private ACL (PACL) is instantiated between a prover and a set of s verifiers. The prover holds an access key sk and the function $f_i \in \mathcal{F}$. The verifiers hold secret-shares $[f_i]$ and have the access control list Λ (see Definition 2.1) for the function family $\mathcal{F}$. The verifiers determine whether or not CheckAccess outputs yes, without learning f_i. See Figure 1 for an overview.

Efficiency constraints. As highlighted in Section 1, a PACL scheme is efficient if it has a small communication overhead for the prover (relative to sharing f) and at most one message exchanged between verifiers. By requiring that only *one*, constant-sized message is exchanged between verifiers, we achieve optimal communication (up to constant factors) and ensure function privacy against malicious verifiers deviating from protocol. (Our definition will also implicitly eliminate all solutions that involve the prover in the verification process.)

3.1. Defining PACLs

A PACL scheme consists of four algorithms: KeyGen, Prove, Audit, and Verify, parameterized by a function family $\mathcal{F}$, and integers $2 \leq t \leq s$. Prove is used by the prover to generate an access control proof for a function f_i. The other algorithms are used by the verifiers to enforce access control. The Audit and Verify algorithms, combined, instantiate CheckAccess for the family of functions in the distributed setting. Audit and Verify only reveal the output of CheckAccess (yes or no). without revealing any other information to the verifiers. We leave public parameters as an implicit input to all algorithms.

Definition 3.1 (PACL: Syntax, Completeness, & Efficiency)**.** Let $\lambda \in \mathbb{N}$ be a security parameter, integer $N := 2^n$, and $\mathcal{F} : \{f_i : \{0,1\}^n \to \{0,1\}^* \mid 1 \leq i \leq N\}$ be a family of functions. Fix integers $2 \leq t \leq s$ and let $(\mathsf{Gen}, \mathsf{Eval})$ instantiate a (t, s)-FSS scheme for $\mathcal{F}$. A (t, s)-PACL scheme consists of efficient algorithms KeyGen, Prove, Audit, and Verify defined as follows:

- $\mathsf{KeyGen}(1^\lambda, f) \to (\mathsf{vk}, \mathsf{sk})$. Takes as input a security parameter 1^λ and a function $f \in \mathcal{F}$. Outputs a new pair of verification and access keys $(\mathsf{vk}, \mathsf{sk})$.
- $\mathsf{Prove}(f, \mathsf{sk}) \to ([\pi]_1, \ldots, [\pi]_s)$. Takes as input a function $f \in \mathcal{F}$ and access key sk. Outputs a vector of s proof secret shares $([\pi]_1, \ldots, [\pi]_s)$.
- $\mathsf{Audit}(\Lambda, [f]_i, [\pi]_i) \to \tau_i$. Takes as input access control list $\Lambda = (\mathsf{vk}_1, \ldots, \mathsf{vk}_N)$, function secret share $[f]_i$ of f sampled according to Gen, and proof share $[\pi]_i$. Outputs audit token τ_i.
- $\mathsf{Verify}(\mathcal{T} := \{\tau_i \mid i \in I\}) \to \mathsf{yes}/\mathsf{no}$. Takes as input a set of t or more audit tokens indexed by the set $I \subseteq \{1, \ldots, s\}$. Outputs yes or no.

The above functionality must satisfy:

- **Completeness.** Let CheckAccess be as defined in Definition 2.1. A (t, s)-PACL scheme with secret shares $([f]_1, \ldots, [f]_s)$ of $f \in \mathcal{F}$ sampled according to $\mathsf{Gen}(1^\lambda, f)$ is complete if for all security parameters λ, for all subsets $I \subseteq \{1, \ldots, s\}$ with $|I| \geq t$, and for all $\Lambda := (\mathsf{vk}_1, \ldots, \mathsf{vk}_N)$ where $\forall i, \mathsf{vk}_i$ is sampled according to KeyGen:

$$\Pr\left[\begin{array}{l} ([\pi]_1, \ldots, [\pi]_s) \leftarrow \mathsf{Prove}(f, \mathsf{sk}); \\ \{\tau_i \leftarrow \mathsf{Audit}(\Lambda, [f]_i, [\pi]_i) \mid i \in I\} : \\ \mathsf{Verify}(\{\tau_i \mid i \in I\}) = \mathsf{CheckAccess}(\Lambda, f, \mathsf{sk}) \end{array}\right] = 1,$$

  where the probability is taken over the randomness of KeyGen and Prove.

- **Efficiency.** The size of each proof share $[\pi]_i$ is most $O(\lambda N^\epsilon)$ for any $\epsilon < 1$ (possibly dependent on n). The size of each audit token τ_i is $O(\lambda)$.

Remark 2. We will primarily be interested in PACLs where ϵ, as defined in the efficiency property of Definition 3.1 (PACL), matches the ϵ of Definition 2.2 (FSS). This translates to a constant overhead in communication over sharing f itself via FSS (i.e., *without* any access control).

Definition 3.2 (PACL, Soundness & Privacy)**.** A PACL scheme (as defined in Definition 3.1) must satisfy the soundness and privacy properties, which are defined as follows.

- **Soundness.** There exists a negligible function negl and security parameter $\lambda \in \mathbb{N}$ such that for all efficient algorithms $\mathcal{A}$ and subsets $I \subseteq \{1, \ldots, s\}$ where $|I| \geq t$,

$$\Pr[\textsc{PkSoundness}_{\mathrm{PACL},\mathcal{A},I}(\lambda) = \mathsf{yes}] \leq \mathsf{negl}(\lambda),$$

  where $\textsc{PkSoundness}_{\mathrm{PACL},\mathcal{A},I}(\lambda)$ is defined in Figure 2.

Game $\textsc{PkSoundness}_{\mathrm{PACL},\mathcal{A},I}(\lambda)$	Oracle $\textsc{GetKey}(j)$
for $i \in \{1, \ldots, N\}$:	$T := T \cup \{j\}$
$(\mathsf{vk}_i, \mathsf{sk}_i) \leftarrow \mathsf{KeyGen}(1^\lambda, f_i)$	**return** sk_j
$\Lambda := (\mathsf{vk}_1, \ldots, \mathsf{vk}_N),\ T = \{\}$	
$([f_\gamma], [\pi_\gamma]) \leftarrow \mathcal{A}^{\textsc{GetKey}}(1^\lambda, \Lambda)$	
$f_\gamma \leftarrow \mathsf{Recover}([f_\gamma])$	
for $i \in I$:	
$\tau_i \leftarrow \mathsf{Audit}(\Lambda, [f_\gamma]_i, [\pi_\gamma]_i)$	
return $\mathsf{Verify}(\{\tau_i \mid i \in I\}) = \mathsf{yes}$ **and** $f_\gamma \in \mathcal{F}$ **and** $\gamma \notin T$	

Figure 2: PACL soundness game.

  In words, no efficient algorithm $\mathcal{A}$ can forge a proof π that verifies with non-negligible probability without knowledge of an access key for f_γ.

- **Privacy.** For all subsets $I \subset \{1, \ldots, s\}$ such that $|I| < t$, define $J := \{1, \ldots, s\} \setminus I$ and $D_{I,J}$ to be the distribution over $\{([\pi]_i, \tau_i^*) \mid i \in I\} \cup \{\tau_j \mid j \in J\}$ where each $[\pi]_i$ is sampled according to $\mathsf{Prove}(f, \mathsf{sk})$, each τ_i^* is sampled

arbitrarily, and $\tau_j \leftarrow \mathsf{Audit}(\Lambda, [f]_j, [\pi]_j)$ for all $j \in J$. A (t, s)-PACL is private if there exists an efficient simulator $\mathcal{S}$ such that: $D_{I,J} \approx_c \mathcal{S}(1^\lambda, I, \{\tau_i^* \mid i \in I\})$. That is, the distribution of proof shares and audit shares reveal nothing about f_i or the access key sk to a subset of at most $t-1$ computationally bounded (possibly malicious) verifiers.

3.2. Symmetric-key PACL

For some applications [19, 39], it is useful to relax the definition of soundness of Definition 3.2 and let the access control list Λ consist of the *secret* keys rather than *public* keys (see prior approaches in Section 1.2). In this regime, the soundness definition must exclude Λ from the inputs to the adversary $\mathcal{A}$. In practical terms, symmetric-key PACLs do not protect against snapshot attacks where an adversary might momentarily compromise a verifier and learn Λ (allowing it to subvert the access control at a later point in time) [21]. We provide a formal definition in Appendix A.

3.3. Security against malicious verifiers

Privacy. Definition 3.1 guarantees privacy against any subset of fewer than t malicious verifiers. Only one message (the audit token) is exchanged by the verifiers to check the proof. Thus, the audit token of each honest verifier is guaranteed to be computed *independently* of audit tokens output by malicious verifiers. As a consequence of this, the simulator $\mathcal{S}$—as defined in Definition 3.2—can simply ignore the audit tokens output by malicious verifiers (i.e., malicious verifiers have no influence over the output of the honest verifiers). This simplifies the analysis required in our security proofs (Section 4.4).

Soundness. In contrast, note that the *soundness* property of PACLs is only guaranteed if all verifiers follow the protocol. This is a natural consequence of the fact that FSS itself only guarantees integrity of the output if all evaluators adhere to the protocol (any malicious evaluator can incorrectly compute $[f_i(x)]$ to corrupt the final output). As such, access control is only well-defined when verifiers have a vetted interest in ensuring correctness of the function evaluation.

3.4. Key distribution

Key distribution is a challenging problem in many real-world systems. Systems using FSS and PACLs must handle distributing the verification and access keys to the users (dealers) and function evaluators (verifiers). This can be done through a variety of techniques. For example, a trusted setup can take place to generate and distribute the keys. Alternatively, anonymous communication channels can be used to register with the evaluators by providing a verification key for a particular function. Ultimately, the key distribution mechanism itself is orthogonal to the goals of PACLs as it depends significantly on the deployment setting (e.g., see Express [19] and Spectrum [31]).

4. PACLs for Distributed Point Functions

In this section, we describe our PACL constructions for the class of distributed point functions (DPFs). DPFs are the main primitive behind more complex FSS classes constructible from minimal assumptions [6, 8]. By focusing on DPFs, our PACL constructions become applicable to larger classes of FSS, which we explain further in Section 6.

Distributed Point Functions (DPFs). A *point function* P_i is a function that evaluates to 1 on input i and evaluates to 0 on all other inputs $j \neq i$. A *distributed* point function is an instance of FSS for the family of point functions. More generally, DPFs can be defined to output *any* value m at index i [22]. We focus on $m = 1$ for simplicity and note that our constructions generalize to arbitrary m.

4.1. PACLs for DPFs

Parameters. Let $\mathbb{G}$ be a group of order $p = p(\lambda)$ with generator g in which the discrete logarithm problem is assumed to be computationally intractable. We assume that the family of (distributed) point functions has range $\mathbb{Z}_p$. In the special case of two-party DPF constructions, which output in a binary field [6, 8], our constructions can be adapted by simply "interpreting" the binary secret share as an element of $\mathbb{Z}_p$, resulting in subtractive secret shares of either -1 or 1 at the special index, which the prover knows.

Overview. In Section 4.1.1, we construct a DPF-PACL for the match predicate of Section 2.1. Our construction can be seen as a generalization of the technique used by Newman et al. [31]. In Section 4.1.2, we extend this technique to a DPF-PACL for the inclusion predicate of Section 2.1.

4.1.1. DPF-PACL for match predicate.

In Algorithm 1, we present the construction for a DPF-PACL with CheckAccess instantiated for the match predicate described in Section 2.1. Loosely speaking, the idea behind the construction is to use the DPF to locally select shares of the ith verification key in Λ. Two facts make this possible: (1) all the verifiers have $\Lambda = (g^{\alpha_1}, \ldots, g^{\alpha_N})$ and (2) the FSS key κ_i encoding a DPF can be used to privately retrieve the kth entry in any vector by first evaluating the DPF $[y_j]_i \leftarrow \mathsf{DPF.Eval}(\kappa_i, j)$ and then computing the inner-product "in the exponent" as: $g^{[\alpha_k]} := g^{\langle(\alpha_1,\ldots,\alpha_N),([y_1]_i,\ldots,[y_N]_i)\rangle}$. This allows the verifiers to *locally* obtain a (multiplicative) secret share $g^{[\alpha_k]_i}$. To verify knowledge of α_k, the prover distributes to the verifiers *additive* secret shares of $\pi := -\alpha_k = \mathsf{sk}_k$ (described in Prove). Each verifier computes $\tau_i := (g^{[\alpha_k]_i})g^{[\pi]_i}$ using Audit and reveals τ_i to all other verifiers. All verifiers proceed to check that $\tau = g^0$ (described in Verify).

Theorem 1. *There exists a DPF-PACL for the FSS family* $\mathsf{DPF} : \{0,1\}^n \to \mathbb{Z}_p$ *with proof size* $O(\lambda)$ *and audit size* $O(\lambda)$*, where* CheckAccess *is instantiated as the match predicate of Section 2.1.*

Algorithm 1: DPF-PACL FOR MATCH PREDICATES

Public parameters: integers $2 \leq t \leq s$, function family $\mathcal{F} = \{f_i : \{0,1\}^n \to \mathbb{Z}_p \mid 1 \leq i \leq N\}$, and group $\mathbb{G} = (g, p)$.

- $\mathsf{KeyGen}(1^\lambda, f_i)$:
  1: $\alpha_i \leftarrow_R \mathbb{Z}_p$
  2: $\mathsf{vk}_i := g^{\alpha_i}$, $\mathsf{sk}_i := -\alpha_i$
  3: **return** $(\mathsf{vk}_i, \mathsf{sk}_i)$

- $\mathsf{Prove}(f, \mathsf{sk})$:
  1: $([\pi]_1, \ldots, [\pi]_s) \leftarrow \mathsf{Share}_{(\mathbb{Z}_p, t, s)}(\mathsf{sk})$
  2: **return** $([\pi]_1, \ldots, [\pi]_s)$

- $\mathsf{Audit}(\Lambda, [f]_i, [\pi]_i)$:
  1: **parse** $\Lambda = (g^{\alpha_1}, \ldots, g^{\alpha_N})$ and $[f]_i = \kappa_i$
  2: $[y_j]_i \leftarrow \mathsf{DPF.Eval}(\kappa, j)$, $\forall j \in \{1, \ldots, N\}$
  3: $A := \prod_{j=1}^{N} (g^{\alpha_j})^{[y_j]_i}$ `// Inner product in` $\mathbb{G}$.
  4: $\tau_i := A \cdot g^{[\pi]_i}$
  5: **return** τ_i

- $\mathsf{Verify}(\mathcal{T})$:
  1: **parse** $\mathcal{T} := \{\tau_1, \ldots, \tau_t\}$.
  2: $C \leftarrow \mathsf{ExpRecover}(\tau_1, \ldots, \tau_t)$ `// See Remark 1.`
  3: **return** $C \stackrel{?}{=} 1_{\mathbb{G}}$

4.1.2. DPF-PACL for inclusion predicates. We now describe how to instantiate a DPF-PACL with an inclusion predicate (Section 2.1). Each function is associated with ℓ access keys. As such, Λ consists of N verification keys, where each verification key consists of ℓ subkeys. For each $\mathsf{vk}_i \in \Lambda$, any of the ℓ subkeys can be used to prove access rights for the function f_i.

Theorem 2. *Let s_ℓ be the size of a DPF key for a point function with domain $\{1, \ldots, \ell\}$. There exists a DPF-PACL for the FSS family $\mathsf{DPF} : \{0,1\}^n \to \mathbb{Z}_p$ with proof size $O(\lambda + s_\ell)$ and audit size $O(\lambda)$, where* CheckAccess *is instantiated as the inclusion predicate of Section 2.1.*

Algorithm 2 presents our construction of DPF-PACL for inclusion predicates. At a high level, the verifiers "select" the ith row in the matrix Λ using f_i (similarly to Algorithm 1) by computing the inner product between the evaluation of f_i on its domain and the rows of the access control matrix. However, the challenge is then to have the verifiers obliviously select the jth *column* in the selected row. Because the resulting row is secret-shared, the verifiers cannot recursively select the column using another DPF, as it would require the vector to be known by all verifiers. Revealing the column does not work either as it would violate the privacy requirement of Definition 3.2. One option is to use zero-knowledge proofs over secret shares [4, 12]. However, we opt for a simpler and more efficient approach. First, the verifiers generate ℓ sums of verification keys for each row in the access control list

Algorithm 2: DPF-PACL FOR INCLUSION PREDICATES

Public parameters: integers $2 \leq t \leq s$, function family $\mathcal{F} = \{f_i : \{0,1\}^n \to \mathbb{Z}_p \mid 1 \leq i \leq N\}$, and group $\mathbb{G} = (g, p)$.

Let $(\mathsf{Prove}', \mathsf{Audit}', \mathsf{Verify}')$ be as in Algorithm 1.

- Precomputation: `// Compute correction terms`
  1: **parse** $\Lambda = (\mathsf{vk}_1, \ldots, \mathsf{vk}_N)$, $\mathsf{vk}_j = (\mathsf{vk}_{j,1}, \ldots, \mathsf{vk}_{j,\ell})$
  2: **for** $j \in \{1, \ldots, N\}$, $k \in \{1, \ldots, \ell\}$:
    2.1: $\mathsf{vk}_{j,k} = (g^{\alpha_{j,k}}, g^{\beta_{j,k}})$
    2.2: $g^{w_{(j-1)\ell+k}} := \prod_{l=1, l\neq k}^{\ell} g^{\alpha_{j,l}}$

- $\mathsf{KeyGen}(1^\lambda, f_i)$:
  1: $(\alpha_{i,1}, \ldots, \alpha_{i,\ell}) \leftarrow_R \mathbb{Z}_p^\ell$, $(\beta_{i,1}, \ldots, \beta_{i,\ell}) \leftarrow_R \mathbb{Z}_p^\ell$
  2: **for** $j \in \{1, \ldots, \ell\}$
    2.1: $\mathsf{vk}_{i,j} := (g^{\alpha_{i,j}}, g^{\beta_{i,j}})$, $\mathsf{sk}_{i,j} := (-\alpha_{i,j}, -\beta_{i,j}, j)$
  3: **return** $(\mathsf{vk}_{i,1}, \ldots, \mathsf{vk}_{i,\ell}), (\mathsf{sk}_{i,1}, \ldots, \mathsf{sk}_{i,\ell})$

- $\mathsf{Prove}(f, \mathsf{sk})$:
  1: **parse** $f = P_i$ and $\mathsf{sk} = (\alpha, \beta, \gamma)$
  2: $([\alpha]_1, \ldots, [\alpha]_s) \leftarrow \mathsf{Share}_{(\mathbb{Z}_p, t, s)}(\alpha)$
  3: $\omega := (i-1)\ell + \gamma$ `// γth key in row i`
  4: $(\kappa'_1, \ldots, \kappa'_s) \leftarrow \mathsf{DPF.Gen}(1^\lambda, P_\omega)$
  5: $([\tilde{\pi}]_1, \ldots, [\tilde{\pi}]_s) \leftarrow \mathsf{Prove}'([P_\omega, \beta)$
  6: $[\pi]_j := ([\alpha]_j, [\tilde{\pi}]_j, \kappa'_j)$ **for** $j \in \{1, \ldots, s\}$
  7: **return** $([\pi]_1, \ldots, [\pi]_s)$

- $\mathsf{Audit}(\Lambda, [f]_i, [\pi]_i)$:
  1: **parse** $\Lambda = (\mathsf{vk}_1, \mathsf{vk}_2 \ldots, \mathsf{vk}_N)$,
  $\mathsf{vk}_j = (\mathsf{vk}_{j,1}, \ldots, \mathsf{vk}_{j,\ell})$, $\mathsf{vk}_{j,k} = (g^{\alpha_{j,k}}, g^{\beta_{j,k}})$,
  $[f]_i = \kappa_i$, and $[\pi]_i = ([\alpha]_i, [\beta]_i, \kappa'_i)$
  2: $[y_j]_i \leftarrow \mathsf{DPF.Eval}(\kappa_i, j)$ **for** $j \in \{1, \ldots, N\}$
  3: $(A_1, \ldots, A_\ell) := \prod_{j=1}^{N} (g^{\alpha_{j,1}}, \ldots, g^{\alpha_{j,\ell}})^{[y_j]_i}$
  4: $\Lambda' := (g^{w_1}, \ldots, g^{w_{N\ell}})$, $\tau_i^{(0)} \leftarrow \mathsf{Audit}'(\Lambda', \kappa'_i, [\beta]_i)$
  5: $[c_j]_i \leftarrow \mathsf{DPF.Eval}(\kappa'_i, j)$, $\forall j \in \{1, \ldots, N\ell\}$
  6: $W := \prod_{j=1}^{N\ell} (g^{w_j})^{[c_j]_i}$ `// Correction term.`
  7: $A := \left(\prod_{j=1}^{\ell} A_j\right) \cdot (W)^{-1}$, $\tau_i^{(1)} := A \cdot g^{[\alpha]_i}$
  8: **return** $\tau_i := (\tau_i^{(0)}, \tau_i^{(1)})$

- $\mathsf{Verify}(\mathcal{T})$:
  1: **parse** $\mathcal{T} = \{\tau_1, \ldots, \tau_t\}$ and $\tau_i = (\tau_i^{(0)}, \tau_i^{(1)})$
  2: **if** $\mathsf{Verify}'\left(\left\{\tau_1^{(0)}, \ldots, \tau_t^{(0)}\right\}\right) = \mathsf{no}$ **then return** no
  3: $C \leftarrow \mathsf{ExpRecover}(\tau_1^{(1)}, \ldots, \tau_t^{(1)})$ `// See Remark 1.`
  4: **return** $C \stackrel{?}{=} 1_{\mathbb{G}}$

(resulting in a total of $N\ell$ terms). One of these sums can then be used as a "correction term" by the prover to select only the jth column in the row. To see how, consider a row $R_i = (g^{\alpha_{i,1}}, \ldots, g^{\alpha_{i,\ell}})$. Each of the ℓ correction terms $g^{w_{i,1}}, \ldots, g^{w_{i,\ell}}$ associated with the ith row is defined as: $g^{w_{i,j}} := \prod_{k=1, k\neq j}^{\ell} g^{\alpha_{i,k}}$. The jth entry in the multiplica-

tively secret-shared row $[R_i] := (g^{[\alpha_{i,1}]}, \ldots, g^{[\alpha_{i,\ell}]})$ can be recovered as: $g^{[\alpha_{i,j}]} := \left(\prod_{k=1}^{\ell} g^{[\alpha_{i,k}]}\right) / g^{[w_{i,j}]}$.

The prover can easily select the correction term $g^{w_{ij}}$ by generating a separate DPF for the point function P_ω and sending it to the verifiers. The verifiers use the DPF to select the ωth term in the list of correction terms. To see how, notice that we can "flatten" the correction terms into a list of size $N\ell$ elements and take the inner product to get a secret share of the ωth correction term, as in Section 4.1.1.

Unfortunately, while the prover can now select the correct key in the list, this idea also creates an avenue for an attack. A malicious prover can subvert access control entirely by selecting multiple correction terms to "annihilate" a row. The prover can send a distributed *multi-point* function (a point function that evaluates to 1 on multiple inputs) to select all ℓ correction terms of a row in Λ. Then, the verifiers would recover shares of g^0 for which the discrete logarithm is simply zero. To prevent this attack, we leverage the following insight: each correction term is associated with a *unique* access key in Λ. As a consequence, we can instantiate a separate DPF-PACL to enforce access control over the vector of correction terms. Specifically, we generate an access key $\beta_{i,j}$ for $w_{i,j}$ and apply Algorithm 1 to enforce the access control over the set of correction terms. The access key is now a *tuple* $(\alpha_{i,j}, \beta_{i,j})$, and verification consists of checking access control for *two* DPFs: the implicit DPF (P_i) and the DPF selecting the correction term.

4.2. Optimizations and extensions

We briefly highlight some optimizations and extensions that can be applied to Algorithms 1 and 2.

Reducing communication and computation. We present Algorithm 2 with a separate DPF for the selection of the correction term. This would result in an additive overhead of $O(\lambda(N\ell)^\epsilon)$ in communication (ϵ as defined in Definition 2.2). However, we observe that we can use the "FSS tensoring" transformation described by Boyle et al. [8] to capitalize on the common "backbone" of the underlying DPF being authenticated and reduce the communication overhead from $O(\lambda(N\ell)^\epsilon)$ down to $O(\lambda\ell^\epsilon)$. Specifically, the prover can use $P_i(i)$ (the non-zero output of the DPF) as a mask for κ' (the key for the DPF selecting the correction term). In this way, κ' only needs a range of ℓ (rather than $N\ell$) leading to the reduced proof size. In the interest of space, we point the reader to Boyle et al. [8] for a full description of the FSS tensoring technique.

Sparse domain auditing. The constructions presented in Algorithms 1 and 2 require $O(N)$ work per verifier to compute Audit. However, in practice, the evaluators (i.e., verifiers) might only evaluate f on a *sparse* subset of the domain rather than the entire domain of the function. In this case, the verifiers only need to compute Audit on the matching subset of the domain on which they evaluate f. Taking this to its extreme, if the verifiers only evaluate f on a constant number of inputs, then this optimization leads to Audit running in $O(1)$ time. Furthermore, the ACL Λ need only contain $O(1)$ keys. More generally, for a subset $S \subseteq \{1, \ldots, N\}$ of the DPF domain, we need $|S|$ keys in Λ and evaluate Audit on the $|S|$ inputs, making the verifier work $O(|S|)$. Given this optimization, the overhead of PACLs is essentially constant relative to the evaluation of the function itself. A downside, however, is that the prover may need to know S (or a subset thereof) when computing Prove. More specifically, we can view this optimization as enforcing access control on a smaller function f' that *coincides* with f on all inputs in the subset S. That is, $f'(x) = f(x)$ for all $x \in S$ but it may be the case that $f'(x') \neq f(x')$ for all $x' \notin S$, which naturally requires the prover to know f'.

Public-key vs. symmetric-key DPF-PACL. When $\mathbb{G}$ is chosen to be a group in which the discrete logarithm problem is assumed to be computationally intractable [3] (e.g., when $\mathbb{G} = \mathbb{Z}_p^*$) then the construction satisfies the soundness property of PACLs as defined in Definition 3.2. When $\mathbb{G}$ is the *additive* group $\mathbb{Z}_p$, then we get a symmetric-key PACL satisfying the relaxed soundness property in Appendix A.

4.3. Aggregating PACLs

A nice property of our DPF-PACL constructions (Sections 4.1.1 and 4.1.2) is the ability to *aggregate* proofs across different DPFs and access control lists. Concretely, our constructions satisfy the following two aggregation properties. At a high level, for any integer q that is polynomial in the security parameter λ and family of point functions $\mathcal{F}$:

1) Let Λ be an ACL for the family $\mathcal{F}$ and let $f_1, \ldots, f_q \in \mathcal{F}$ have associated access keys $\alpha_1, \ldots, \alpha_q \in \Lambda$, then $\alpha' := \sum_{i=1}^{q} \mathsf{sk}_i$ is an access key for $f'(x) := \sum_{i=1}^{q} f_i(x)$. This aggregation property allows the verifiers to simultaneously enforce access control on q distinct functions in the family for the computational and bandwidth overhead of verifying a single function in the family.
2) Our constructions permit aggregating proofs from multiple *separate* ACLs to simultaneously enforce access control on a vector of functions $(f_1, \ldots, f_q) \in \mathcal{F}^q$. For a vector of ACLs $(\Lambda_1, \ldots, \Lambda_q)$, the ACL $\Lambda' := \bigodot_{i=1}^{q} \Lambda_i$ (where $\odot$ denotes the group operation applied component-wise) is an ACL for the vector $(f_1, \ldots, f_q)$ such that $\mathsf{CheckAccess}(\Lambda_i, f_i, \mathsf{sk}_i) = \mathsf{yes}$ for all i. This aggregation property allows for batched verification of q functions, each associated with a *separate* ACL: the verifiers first compute $g^{[\alpha^{(1)}]}, \ldots, g^{[\alpha^{(q)}]}$ individually for each function using the corresponding ACL. Then $g^{[\alpha]} := \bigodot_{i=1}^{q} g^{[\alpha^{(i)}]}$ can be verified using Λ'. While the computational overhead of this aggregation property remains proportional to verifying each function individually, it permits compact proofs and audits.

4.4. Security analysis

In this section, we prove security of Algorithms 1 and 2 with respect to Definitions 3.1 and 3.2. We first prove

a useful lemma which says that any adversary that wins the $\text{PKSOUNDNESS}_{\text{PACL},\mathcal{A},I}(\lambda)$ game in our DPF-PACL constructions with some function (not necessarily a DPF), must also implicitly output a valid DPF and access key.

Lemma 1. *If there exists an efficient $\mathcal{A}$ that wins the $\text{PKSOUNDNESS}_{\text{PACL},\mathcal{A},I}(\lambda)$ game for our DPF-PACL constructions (Algorithms 1 and 2) with non-negligible probability $\delta(\lambda)$ for some function $\hat{f}_\gamma$ and proof $\hat{\pi}$ where $\hat{f}_\gamma$ is not sampled from $\mathcal{F}_{\text{DPF}}$, then, there exists an efficient $\mathcal{A}'$ that wins the $\text{PKSOUNDNESS}_{\text{PACL},\mathcal{A},I}(\lambda)$ game with probability $\delta(\lambda)$, where $\mathcal{A}'$ outputs f_γ and π such that $f_\gamma \in \mathcal{F}_{\text{DPF}}$.*

Proof. Deferred to Appendix A. □

Theorem 3. *Let p be a prime chosen with respect to a security parameter λ and let $\mathbb{G}$ be any group of order p in which the discrete logarithm problem is assumed to be computationally intractable. Algorithm 1 (DPF-PACL for match predicates) satisfies the completeness, efficiency, soundness, and privacy properties of Definitions 3.1 and 3.2 with* CheckAccess *as defined in Section 2.1 (match predicate).*

Proof. We prove each property in turn.

Completeness. Let i be the special index of the encoded point function. Consider the exponent of the recovered audit: $\log_g(C) = \log_g(A \cdot g^{-\alpha}) = (\sum_{j=1}^{N} \alpha_j y_j) - \alpha$. We have: $(\sum_{j=1}^{N} \alpha_j y_j) = \alpha_i$ if $i \neq 0$ and 0 otherwise. As such, $\log_g(C) = \alpha_i - \alpha$. By construction, $\alpha := \alpha_i$, so it follows that $\log_g(C) = 0$. Therefore, $C = g^0 = 1_{\mathbb{G}}$, as required.

Soundness. Assume, towards contradiction, that there exists an efficient prover $\mathcal{A}$ and non-negligible function δ such that for all $I \subseteq \{1, \ldots, s\}$ where $|I| \geq t$:

$$\Pr[\text{PKSOUNDNESS}_{\text{PACL},\mathcal{A},I}(\lambda) = \mathsf{yes}] \geq \delta(\lambda).$$

By Lemma 1, we can assume that f_γ (output by $\mathcal{A}$ in Figure 2) is a point function with special index γ. Construct an efficient algorithm $\mathcal{B}$ that solves the discrete logarithm problem as follows. On input $y := g^x$, sample random $\gamma' \leftarrow_R \{1, \ldots, N\}$ and $(\alpha_1, \ldots, \alpha_N) \leftarrow_R \mathbb{Z}_p^N$. Set $\Lambda := (g^{\alpha_1}, \ldots g^{\alpha_N})$ but replace $g^{\alpha_{\gamma'}}$ with y. Let $T := \{\}$. Run $\mathcal{A}^{\text{GETKEY}}(1^\lambda, \Lambda)$. Respond to each $\text{GETKEY}(j)$ query with α_j (unless $j = \gamma'$, in which case abort) adding j to T. Obtain f_γ and π from $\mathcal{A}$. If $\gamma \neq \gamma'$ output fail. Else, output $-\pi$. The list Λ constructed by $\mathcal{B}$ matches the distribution of KeyGen because $y := g^x$ is a random element of $\mathbb{G}$. If $\mathcal{A}$ succeeds, then Verify outputs yes, which means that $C = 1_{\mathbb{G}}$ and so it holds that $x = -\pi$. The probability that $\gamma = \gamma'$ is $\frac{1}{N}$ and so $\mathcal{B}$ succeeds with probability at least $\frac{1}{N}\delta(\lambda)$, which remains non-negligible. As such, $\mathcal{B}$ successfully recovers the discrete logarithm in $\mathbb{G}$ contradicting the assumption that the discrete logarithm is computationally intractable in $\mathbb{G}$.

Privacy. We construct an efficient simulator $\mathcal{S}$ for the view of any subset of $t-1$ (possibly malicious) verifiers. On input $(1^\lambda, I, \{\tau_i^* \mid i \in I\})$, $\mathcal{S}$ proceeds as follows:

1: $J := \{1, \ldots, s\} \setminus I$.
2: $([0]_1, \ldots, [0]_s) \leftarrow \mathsf{Share}_{(\mathbb{Z}_p,t,s)}(0)$.
3: $([\pi]_1, \ldots, [\pi]_s) \leftarrow \mathsf{Share}_{(\mathbb{Z}_p,t,s)}(0)$.
4: $\tau_k := g^{[0]_k}$ for all $k \in I \cup J$.
5: Output $\{([\pi]_i, \tau_i) \mid i \in I\} \cup \{\tau_j \mid j \in J\}$.

First, note that the τ_i^*'s are independent of the honest verifier outputs (see Section 3.3) and therefore do not influence the simulator. The distribution output by $\mathcal{S}$ matches the distribution of any subset $I \subset \{1, \ldots, s\}$, where $|I| < t$, because in the real view, (1) the proof shares $[\pi]$ are output by Share which guarantees that any subset of fewer than t shares is information-theoretically hiding and (2) the audit tokens are (computationally-hiding) multiplicative secret shares of $g^0 = 1_{\mathbb{G}}$. The audit tokens in the real view are not information-theoretically hiding because they are computed using the output of the DPF, which consists of computationally-hiding secret shares. The output of $\mathcal{S}$ thus only differs on (2). However, if there is an efficient distinguisher for (2), then the FSS scheme is not private, a contradiction.

Efficiency. Each proof share $[\pi]_i$ is an element of $\mathbb{G}$ and thus is of size $O(\lambda)$. Each audit token is also of size $O(\lambda)$. □

Theorem 4. *Let p be a prime chosen with respect to a security parameter λ and let $\mathbb{G}$ be any group of order p in which the discrete logarithm problem is assumed to be computationally intractable. Algorithm 2 (DPF-PACL for Inclusion Predicates) satisfies the completeness, efficiency, soundness, and privacy properties of Definitions 3.1 and 3.2 for the inclusion predicate of Section 2.1.*

Proof. The proof follows a similar structure to the proof of Theorem 3 but involves more tedious calculations. We defer the proof to Appendix B. □

5. Faster PACLs from Verifiable DPFs

In this section, we introduce a concretely more efficient construction of DPF-PACL for the class of *verifiable* DPFs (VDPFs) [16] (also known as *extractable* DPFs [5]). A VDPF allows the evaluators to efficiently check if the DPF is well-formed (see Appendix B), which we will capitalize on to construct more efficient PACLs.

The primary source of inefficiency in Algorithms 1 and 2 is due to the group operations required in computing the PACL audit. If, instead, the verifiers could "select" the public key over a field (e.g., $\mathbb{F}_p$) rather than in $\mathbb{G}$, then computing the audit token would be bottle-necked by operations over the field instead of operations in $\mathbb{G}$.

There are two technical challenges with this approach. First, if the audit is not computed in $\mathbb{G}$, the verifiers end up with additive shares of $[g^{\alpha_i}]$ (rather than multiplicative shares $g^{[\alpha_i]}$), which does not lend itself to the efficient verification procedure of Algorithms 1 and 2. To overcome this problem, we introduce a building block we call a *Schnorr Proof over Secret Shares* (SPoSS; Section 5.1), which allows a prover to efficiently prove to a set of verifiers that it knows the discrete logarithm of an additively secret-shared element.

Algorithm 3: SCHNORR PROOF OVER SECRET SHARES

Public parameters: Group $\mathbb{Z}_p^* = (g, p)$ and random oracle H.

- $\mathsf{Prove}(x)$:
  1: $([x]_A, [x]_B) \leftarrow \mathsf{Share}_{(\mathbb{Z}_p,2,2)}(x)$
  2: $y_A := g^{[x]_A}$, $y_B := g^{[x]_B}$
  3: $([a]_A, [b]_A, [c]_A, [a]_B, [b]_B, [c]_B) \leftarrow \mathsf{Beaver}_{(2,2)}(\mathbb{Z}_p)$
  4: $a \leftarrow [a]_A + [a]_B$, $b \leftarrow [b]_A + [b]_B$
  5: $z_A, z_B \leftarrow_R \{0,1\}^\lambda$ // random nonces
  6: $r_A \leftarrow H(z_A, [x]_A, a, [c]_A)$, $r_B \leftarrow H(z_B, [x]_B, b, [c]_B)$
  7: $r \leftarrow r_A + r_B$
  8: $d \leftarrow r g^{[x]_A} - a$, $e \leftarrow g^{[x]_B} - b$
  9: $[\pi]_A := (\text{A}, [x]_A, a, [c]_A, r, d, e, z_A)$
  10: $[\pi]_B := (\text{B}, [x]_B, b, [c]_B, r, d, e, z_B)$
  11: **return** $([\pi]_A, [\pi]_B)$

- $\mathsf{Audit}([y]_i, [\pi]_i)$:
  1: **parse** $[\pi] := (i, [x]_i, u, [c]_i, r, d, e, z)$
  2: $\hat{r}_i \leftarrow H(z, [x]_i, u, [c]_i)$
  3: $\hat{y} \leftarrow g^{[x]_i}$
  4: **if** $i = \text{A}$: / **if** $i = \text{B}$:
  4.1: $f \leftarrow r\hat{y} - u$ / $f \leftarrow \hat{y} - u$
  4.2: $[v]_i \leftarrow (de/2) + eu$ / $[v]_i \leftarrow (de/2) + du$
  4.3: $[w]_i \leftarrow [v]_i + [c]_i - r[y]_i$ / $[w]_i \leftarrow [v]_i + [c]_i - [y]_i$
  5: $\tau_i := ([w]_i, \hat{r}, r, f, d, e)$
  6: **return** τ

- $\mathsf{Verify}(\{\tau_A, \tau_B\})$:
  1: **parse** $\tau_A = ([w]_A, \hat{r}_A, r, \hat{d}, d, e)$
  2: **parse** $\tau_B = ([w]_B, \hat{r}_B, r, \hat{e}, d, e)$
  3: $\hat{r} \leftarrow \hat{r}_A + \hat{r}_B$, $w \leftarrow [w]_A + [w]_B$
  4: **return** $w = 0$ and $\hat{r} = r$ and $\hat{d} = d$ and $\hat{e} = e$

The second challenge is that, in the proof of security, the knowledge extractor (see Section 4.4) would *not* have the guarantee that the resulting additive secret shares encode a verification key from Λ (it could be any linear combination of group elements). This rather subtle problem is a barrier to proving soundness when taking this approach with (non-verifiable) DPFs. To overcome this, we restrict our focus to VDPFs, which ensures that the verifiers always obtain a valid group element from Λ. We then prove security similarly to the proof of Theorem 3.

5.1. Schnorr Proof over Secret Shares (SPoSS)

SPoSS is a non-interactive proof system instantiated in the random oracle model between a prover and a set of two or more verifiers. The verifiers hold additive secret shares of a group element $y := g^x$. The prover provides a zero-knowledge proof-of-knowledge of x (i.e., the discrete logarithm of y base g). SPoSS is a concrete instantiation of a general zero-knowledge proof system over secret shares [4, 12] and can be thought of as a secret-shared analog of a Schnorr proof [34]. We define the formal requirements of SPoSS in Definition 5.1 and prove security of our construction in Appendix C. The proof size of our SPoSS construction is significantly smaller compared to generic approaches based on zero-knowledge proofs (see Section 8).

Definition 5.1 (SPoSS)**.** Let $\lambda \in \mathbb{N}$ be a security parameter and let $\mathbb{G}$ be a cyclic group of order $p = p(\lambda)$ with generator g. A non-interactive zero-knowledge proof of discrete-logarithm knowledge over a (t, s)-secret-shared element y, consists of efficient (possibly randomized) algorithms $(\mathsf{Prove}, \mathsf{Audit}, \mathsf{Verify})$ with the following functionality. We leave $\mathbb{G}$ and g as implicit inputs.

- $\mathsf{Prove}(x) \to ([\pi]_1, \ldots, [\pi]_s)$. Takes as input integer $x \in \mathbb{Z}_p$. Outputs proof shares $([\pi]_1, \ldots, [\pi]_s)$.
- $\mathsf{Audit}([y]_i, [\pi]_i) \to \tau_i$. Takes as input a secret share $[y]_i$ and a secret share $[\pi]_i$. Outputs an audit token τ_i.
- $\mathsf{Verify}(\mathcal{T} := \{\tau_i \mid i \in I\}) \to \mathsf{yes}/\mathsf{no}$. Takes as input any subset of t or more audit tokens indexed by the set $I \subseteq \{1, \ldots, s\}$. Outputs yes if and only if π is a valid proof of discrete logarithm knowledge with respect to $y \in \mathbb{G}$.

The functionality must satisfy the following properties.

Completeness. For all $x \in \mathbb{Z}_p$ and $y := g^x$, and all subsets $I \subseteq \{1, \ldots, s\}$ such that $|I| \geq t$,

$$\Pr\left[\begin{array}{l} ([y]_1, \ldots, [y]_s) \leftarrow \mathsf{Share}_{(\mathbb{Z}_p,t,s)}(y); \\ ([\pi]_1, \ldots, [\pi]_s) \leftarrow \mathsf{Prove}(x); \\ \{\tau_i \leftarrow \mathsf{Audit}([y]_i, [\pi]_i) \mid i \in I\} : \\ \mathsf{Verify}(\{\tau_i \mid i \in I\}) = \mathsf{yes} \end{array}\right] = 1,$$

where the probability is over the randomness of Prove.

Argument-of-knowledge. If there exists an efficient (possibly malicious) prover $\mathcal{P}^*$ such that for all group elements y, $\mathcal{P}^*$ produces $[\pi^*]$ such that $\mathsf{Verify}(\{\tau_i \mid i \in I\}) = \mathsf{yes}$ (where $\tau_i \leftarrow \mathsf{Audit}([y]_i, [\pi^*]_i)$, for all $i \in I$) with probability $\delta(\lambda)$, then there exists an efficient knowledge extractor $\mathcal{E}$ and negligible function negl such that,

$$\Pr\left[\ x \leftarrow \mathcal{E}^{\mathcal{P}^*}(y) \ \ : y = g^x\right] \geq \delta(\lambda) - \mathsf{negl}(\lambda),$$

where the probability is over the randomness of $\mathcal{P}^*$. In words, $\mathcal{E}$ recovers the discrete logarithm x from valid proofs output by $\mathcal{P}^*$. SPoSS is an *argument* (rather than a proof) of knowledge because the prover is computationally bounded in the random oracle model.

Zero-knowledge. For all subsets $I \subset \{1, \ldots, s\}$ such that $|I| < t$, define $J := \{1, \ldots, s\} \setminus I$ and $\mathcal{D}_{I,J}$ to be the distribution over $\{([\pi]_i, \tau_i^*) \mid i \in I\} \cup \{\tau_j \mid j \in J\}$ where $[\pi]_i$ is sampled according to $\mathsf{Prove}(x)$, τ_i^* is sampled arbitrarily, and $\tau_i \leftarrow \mathsf{Audit}([y]_i, [\pi]_i)$ for all $j \in J$. SPoSS is zero-knowledge if there exists an efficient simulator $\mathcal{S}$ such that $\mathcal{D}_{I,J} \approx \mathcal{S}(1^\lambda, I, \{\tau_i^* \mid i \in I\})$. That is, the view induced by the proof shares and audit tokens reveals no information about x or y to any subset of fewer than $t - 1$ computationally bounded (possibly malicious) verifiers.

SPoSS: Main idea. The main idea behind SPoSS is to leverage the additive and multiplicative homomorphism of secret shares over $\mathbb{Z}_{p-1}$ and $\mathbb{Z}_p^*$, respectively. Our construction assumes $\mathbb{G} = \mathbb{Z}_p^*$. However, our approach generalizes to any group where the group operation can be described as an arithmetic circuit over a ring. Notice that, given share $[x]_i$ in $\mathbb{Z}_{p-1}$, each verifier can obtain a *multiplicative* share of x by computing $g^{[x]_i}$. At a high level, the SPoSS verification procedure goes as follows. Each verifier holds additive secret-shares of y and x (secret shared over $\mathbb{Z}_p$ and $\mathbb{Z}_{p-1}$, respectively). First, each verifier computes $g^{[x]_i}$ to obtain a multiplicative secret share of x. Notice that $g^{[x]_i}$ is defined over the field $\mathbb{Z}_p$ and that the group operation of $\mathbb{Z}_p^*$ is multiplication over $\mathbb{Z}_p$. The verifiers then compute the group operation (multiplication in $\mathbb{Z}_p$) over the additive shares using a prover-assisted computation. Notice that as a result of this computation, the verifiers hold additive secret shares $[g^x]$. Third, the verifiers compute $[w] := [y] - [g^x]$ and swap their shares of $[w]$ to check if $w = 0$.

5.1.1. Protocol overview. We describe SPoSS in Algorithm 3. For clarity, we describe the protocol with two verifiers but note that all our techniques extend to a many-verifier setting. In Algorithm 3, the verifiers first derive additive shares of $g^{[x]_i}$ (for $i \in \{A, B\}$), which we denote by $[g^{[x]_i}]$. With two verifiers, this is done by simply letting Verifier A set $[g^{[x]_A}]_A := g^{[x]_A}$ and verifier B set $[g^{[x]_A}]_B := 0$ (observe that $[g^{[x]_A}]_A + [g^{[x]_A}]_B = g^{[x]_A}$, as required). Verifier B proceeds to do the same with $g^{[x]_B}$. If it were possible to compute the product $[g^{[x]_A} \cdot g^{[x]_B}]$ non-interactively over the additive secret shares, then the verifiers could locally obtain $[g^x]$. Unfortunately, doing so requires *interaction* between the verifiers. Instead, in Algorithm 3, we use a standard approach from zero-knowledge proofs over secret-shares [4, 12] and have the prover "assist" the verifiers in the computation. Specifically, the prover provides a Beaver multiplication triple [2], enabling the verifiers to compute the multiplication. (We provide an overview of Beaver multiplication in Appendix D for completeness.)

Preventing malicious provers. As observed in prior work [12], prover-assisted multiplication can allow the prover to cheat by introducing a linear term in the output of the multiplication, which would result in the verifiers computing $[\hat{y}] := [g^{[x]_A} \cdot g^{[x]_B} + \Delta]$, for some Δ. To defend against this attack, in Algorithm 3, the verifiers instead check that $[r(g^{[x]_A} \cdot g^{[x]_B})] - [r(y)] = [0]$ where r is a random scalar chosen by the verifiers. As long as the prover does not choose r, the proof is guaranteed to fail for any $\Delta \neq 0$ with probability $1 - \frac{1}{p}$, when instantiated over $\mathbb{Z}_p$ [12].

Removing interaction. Finally, in Algorithm 3, to avoid interaction between verifiers, we apply the Fiat-Shamir transform [20] and let the *prover* (instead of the verifiers) choose r using a random oracle H. This makes SPoSS mesh with our PACL definition (which only allows for one message exchanged between verifiers). Concretely, we use the distributed analog of Fiat-Shamir described in the full version of Boneh et al. [4]. Given a random oracle H, the prover generates a proof using H to simulate the choice of r by the verifiers. As noted in [4], in the distributed setting, the resulting r can leak information about the shares. To prevent this, in Algorithm 3, we follow the blueprint of [4] and generate random nonces z_A and z_B, that are independent of the proof shares and serve to "mask" the inputs to H.

Reducing proof size. We observe that because each verifier sets all but their own additive share of $\hat{y}_i := g^{[x]_i}$ to zero, the ith verifier knows the value of all other verifiers' "secret" share of $[\hat{y}_i]$ (they are always zero). As a consequence, only the verifier holding the non-zero share needs to mask it when computing the Beaver multiplication (see Appendix D). This corresponds to revealing a (from the Beaver triple) to Verifier A and b to Verifier B, where the Beaver triple is of the form $([a], [b], [ab])$. Because a and b are random, they still serve as a mask when computing the Beaver multiplication. We apply this optimization in Algorithm 3.

5.2. VDPF-PACL using SPoSS

In this section, we describe how SPoSS can be used to construct a VDPF-PACL. We focus on constructing a VDPF-PACL for the match predicate since extending the construction to inclusion predicates can be achieved by following the blueprint of Algorithm 2. We describe our construction in Algorithm 4. The main idea is that, following private selection of the verification keys (as in Algorithm 1 but now over $\mathbb{Z}_p$) with a VDPF (Definition B.1), each verifier holds an (additive) secret-share of $y := g^{\alpha_i}$ (in contrast to Algorithm 1, where the verifiers hold multiplicative secret shares of y). To prove knowledge of α_i, the prover provides a SPoSS proof to the verifiers for the secret-shared group element y. The verifiers then proceed to verify the SPoSS proof and accept if it passes.

5.3. Security analysis

Theorem 5. *Let p be a prime chosen with respect to a security parameter $\lambda \in \mathbb{N}$ and let $\mathbb{G}$ be a group of order p in which the discrete logarithm problem is assumed to be computationally intractable. Algorithm 4 (VDPF-PACL for match predicates) satisfies the completeness, efficiency, soundness, and privacy properties of Definitions 3.1 and 3.2, with* CheckAccess *as defined in Section 2.1 (match predicate).*

Proof. Deferred to the full version of the paper [35]. □

6. PACLs for FSS from DPF-PACLs

We now describe a set of PACL constructions for classes of FSS derived from DPFs. These transformations are taken from Boyle et al. [6, Section 3.2] and form the class of functions that can be efficiently secret-shared using lightweight cryptographic assumptions. More expressive classes of FSS are believed to require heavier tools [6], for instance, fully-homomorphic encryption [17].

Algorithm 4: VDPF-PACL FOR MATCH PREDICATES

Public parameters: integers $2 \leq t \leq s$, function family $\mathcal{F} = \{f_i : \{0,1\}^n \to \mathbb{Z}_p \mid 1 \leq i \leq N\}$, and group $\mathbb{Z}_p^* = (g, p)$.

- $\mathsf{KeyGen}(1^\lambda, f_i)$: as in Algorithm 1.
- $\mathsf{Prove}(f, \mathsf{sk})$:
  1: **parse** $f = P_i$
  2: $([\pi]_1, \ldots, [\pi]_s) \leftarrow \mathsf{SPoSS.Prove}(\mathsf{sk})$
  3: **return** $([\pi]_1, \ldots, [\pi]_s)$
- $\mathsf{Audit}(\Lambda, [f], [\pi])$:
  1: **parse** $\Lambda = (g^{\alpha_1}, \ldots, g^{\alpha_N})$ and $[f] = \kappa$
  2: $([y_j], \rho) \leftarrow \mathsf{VDPF.Eval}(\kappa, j), \ \forall j \in \{1, \ldots, N\}$
  3: $A := \sum_{j=1}^{N} (g^{\alpha_j})[y_j]_i$
  4: $\tilde{\tau} \leftarrow \mathsf{SPoSS.Audit}(A, [\pi])$
  5: **return** $\tau := (\tilde{\tau}, \rho)$
- $\mathsf{Verify}(\mathcal{T})$:
  1: **parse** $\mathcal{T} = \{(\tau_1, \rho_1), \ldots, (\tau_t, \rho_t)\}$.
  2: **return** $\mathsf{SPoSS.Verify}(\{\tau_1, \ldots, \tau_t\})$ and $\mathsf{VDPF.Verify}(\{\rho_i, \ldots, \rho_t\})$

PACLs for range functions and decision trees. Boyle et al. [6, 8] describe how to apply linear combinations of DPFs to derive FSS for range functions (and more generally decision trees [8]). Range functions and decision trees can be viewed as special cases of distributed multi-point functions (DMPF), which evaluate to a non-zero value on multiple inputs. In turn, DMPFs can be viewed as an *aggregation* of DPFs (this also follows from the linear-composition of FSS [6, Section 3.2]). By the aggregation property of DPF-PACLs (Section 4.3), we immediately obtain PACLs for DMPFs and, as a result, PACLs for range functions and decision trees.

PACLs for small function classes. FSS for all functions with a small domain $|\mathcal{F}|$ can be obtained via a DPF that "selects" the function $f_i \in \mathcal{F}$ in the canonically ordered function family $\mathcal{F}$ [6]. Our DPF-PACL construction applies to this class of FSS directly as a result. Following similar transformations, Boyle et al. [6] obtain FSS for data matching and NC^0 functions, which we briefly describe next.

PACLs for data matching functions. Data-matching functions are parameterized by a set $S \subseteq \{1, \ldots, N\}$ of $\ell \in O(1)$ elements and a value $v \in \{0,1\}^n$ such that $f_{S,v}(x) = 1$ if $x_i = v_i, \forall i \in S$. FSS for this class of functions can be realized using a DPF with a range large enough to describe all $\binom{n}{\ell} 2^\ell$ possible values of $f_{S,v}$ (hence the requirement that ℓ is constant). As a consequence, our DPF-PACL can be applied directly to this family of FSS by associating each access key with the corresponding canonically ordered function.

PACLs for NC^0 functions. The class NC^0 captures all functions that can be represented by constant-depth boolean circuits $C : \{0,1\}^u \to \{0,1\}^v$ with fan-in 2 (two inputs per gate). We can trivially consider a DPF-PACL where Λ corresponds to all possible such circuits, of which there are $v^{O(u^{2^d})}$ in total. However, this is a naive approach. As observed by Boyle et al. [6], it is possible to leverage the bit-wise *parallel* structure of NC^0 circuits and DPFs to realize efficient FSS for NC^0 functions. Specifically, any circuit $C \in \mathsf{NC}^0$ can be decomposed into v 1-bit-output, depth-d sub-circuits. For u-bit inputs, each such sub-circuit has only $O(u^{2^d})$ possibilities. For each sub-circuit, we can generate a DPF for the ith canonical ordering of all $O(u^{2^d})$ possible circuits. Repeating this for all v sub-circuits yields an FSS scheme consisting of v DPF keys (one for each sub-circuit). Using the aggregation property of our DPF-PACL construction described in Section 4.3, it is possible to enforce access control over the v DPFs simultaneously. However, it becomes necessary to enforce access control over the unique *combination* of sub-circuits since each DPF-PACL operates independently of the *gloabal* circuit C. To achieve this, we can apply a "generic PACL" (Section 7) over the combination of sub-circuits (in conjunction with DPF-PACLs for each sub-circuits) using a zero-knowledge proof over secret-shared data.

7. Generic PACLs from zero-knowledge proofs

In this section, we describe how to construct PACLs for any FSS class (formally, FSS for all functions in $P/poly$ [6]). Our approach relies on secret-shared non-interactive proofs (SNIPs) [12] and Fiat-Shamir over SNIPs [4]. We describe these in Section 7.1.

7.1. Preliminaries

SNIPs [12] (and their generalizations [4]) can be used to prove that any (public) arithmetic circuit C evaluates to 1 on a secret-shared input x provided that the following two conditions are met: (1) the circuit C is known to the verifiers and (2) the prover knows the input x and C. SNIPs guarantee that the verifiers (who hold secret shares of x) do not learn any information except that $C(x) = 1$. The efficiency of SNIPs is measured by the size of a SNIP proof and the interaction between verifiers (note that SNIPs are non-interactive for the prover). The size of a SNIP is proportional to the number of multiplication gates in the circuit and can be verified in one round of interaction.

Fiat-Shamir for SNIPs. The Fiat-Shamir transform [20] is a standard technique used to eliminate interaction in zero-knowledge proofs. At a high level, Fiat-Shamir allows the prover to generate its own challenges with the help of a random oracle by "simulating" the randomness chosen by the verifiers in the interactive proof system. With SNIPs, however, the situation is slightly different because SNIPs are already non-interactive for the prover. Instead, Fiat-Shamir can be applied to SNIPs to reduce the interaction required when *verifying* the proof [4]. Specifically, the verification

of the SNIP with Fiat-Shamir requires only one message exchanged between verifiers (instead of one round of interaction consisting of two sequential messages).

7.2. Construction

At a high level, we construct PACLs for $P/poly$-FSS as follows. As in our group-based constructions, the dealer first distributes shares of the function f using the FSS scheme and shares of the access key sk to all verifiers. In addition, the prover distributes a SNIP proof showing that (1) $[f]$ corresponds to *some* valid output of $\mathsf{FSS.Gen}$ and (2) the access key sk is such that $\mathsf{CheckAccess}(\Lambda, f, \mathsf{sk}) = 1$. (Without loss of generality, we assume that both $\mathsf{FSS.Gen}$ and $\mathsf{CheckAccess}$ are described as arithmetic circuits over a field $\mathbb{F}$.) The verifiers then check the SNIP using their shares of f and sk.

While this approach to generic PACLs is conceptually simple, there is an efficiency challenge that needs to be addressed. By definition, $\mathsf{CheckAccess}$ takes the entire access control list Λ, which would result in a large (linear in $|\Lambda|$) proof size, violating the efficiency property of PACLs. Specifically, the naive approach requires the prover to incorporate the entire access control list into the SNIP when proving that $\mathsf{CheckAccess}(\Lambda, f, \mathsf{sk}) = 1$, even if sk only depends on *one* verification key in Λ. We overcome this efficiency problem by using any vector commitment scheme [10, 29] (described by algorithms $\mathsf{VC.Commit}$ and $\mathsf{VC.Verify}$). We instead define $\mathsf{vk}_i := H(f_i \| \mathsf{sk}_i)$, where H is a collision-resistant hash function (CRHF) sampled by the verifiers and sk_i is the access key for function f_i. The verifiers compute a commitment to all verification keys using $\mathsf{VC.Commit}$, publishing the resulting commitment c and all the openings $e_1, \ldots, e_N$ such that $\mathsf{VC.Verify}(\mathsf{c}, \mathsf{vk}_i, e_i) = 1$. The prover then sends $[\tilde{\mathsf{vk}}]$, $[\tilde{e}]$, and three SNIP proofs: (1) a proof that $\mathsf{FSS.Gen}(1^\lambda, f_i; r) = [f_i]$, where r describes the random coins of $\mathsf{FSS.Gen}$, (2) a proof that $\tilde{\mathsf{vk}} = H(f_i \| \mathsf{sk}_i)$, and (3) a proof that $\mathsf{Verify}(\mathsf{c}, \tilde{\mathsf{vk}}, \tilde{e}) = 1$. The verifiers check the three SNIP proofs using their shares $[f_i]$ and $[\mathsf{sk}_i]$. If the proofs are accepting, then the verifiers are convinced that the dealer knows the access key sk_i associated with f_i and that their shares of f_i were output according to $\mathsf{FSS.Gen}$. The size of the SNIP proofs is bounded by $O(s + \log|\Lambda|)$ rather than $O(|\Lambda|)$, where s is the number of multiplication gates in $\mathsf{FSS.Gen}$. See the full version of the paper [35] for details.

8. Implementation and evaluation

In this section, we describe our implementation and evaluation of the (V)DPF-PACL constructions from Sections 4 and 5. Our evaluation focuses on the state-of-the-art two-party FSS schemes [8, 16]. Multi-party FSS constructions are less efficient [6] or require heavier cryptographic assumptions, making them concretely slower [13, 31]. Because we are interested in evaluating the *overhead* of PACLs, evaluating our constructions in a two-party setting results in *worst-case overheads* relative to baseline FSS evaluations.

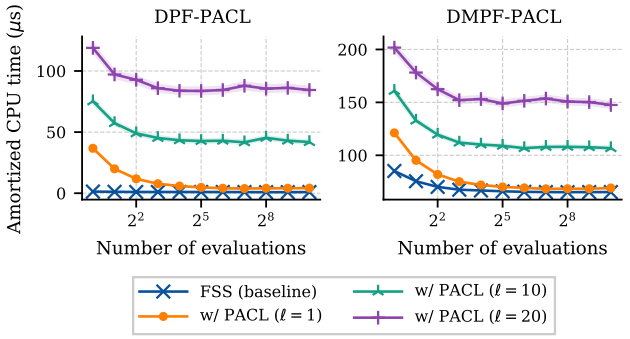


Figure 3: Our DPF-PACL construction amortizes with more evaluations of the DPF but plateaus at approximately 32 evaluations. For DMPFs, which have a higher baseline processing time, the overhead is less significant thanks to aggregation, which amortizes the overhead of access control (Section 4.3).

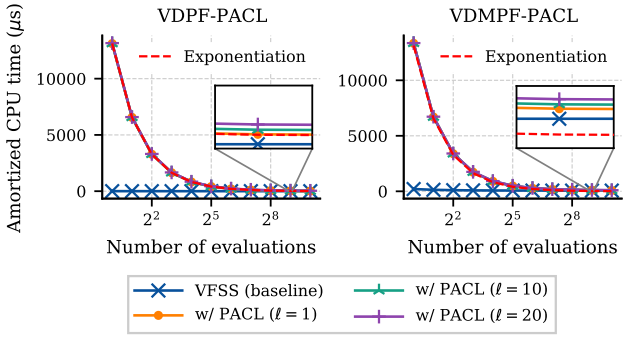


Figure 4: Our VDPF-PACL construction is dominated by the exponentiation in $\mathbb{Z}_p^*$ (dashed lined) and has a higher *initial* overhead compared to the DPF-PACL construction. In contrast to DPF-PACLs, our VDPF-PACL construction amortizes almost entirely after 64 evaluations and has an asymptotically smaller overhead.

We evaluate our implementation for applications of FSS including private information retrieval, distributed anonymous authentication, and anonymous communication protocols.

Implementation. We implement PACLs in Go v1.16. and C in approximately 4,500 lines of code. Our implementation is open-source [1]. We instantiate $\mathbb{G}$ as the P-256 elliptic curve group (part of the `crypto/elliptic` package in Go) in our DPF-PACL construction for public-key (pub) soundness (Definition 3.2) and as $\mathbb{Z}_{2^\lambda}$ for symmetric-key (sym) soundness (Definition A.1). For our public-key VDPF-PACL construction, we instantiate $\mathbb{G}$ as $\mathbb{Z}_p^*$ with a 3072-bit safe prime p as specified in RFC3526.

Environment. We use Amazon Elastic Cloud Compute (EC2) for our experiments. We run experiments on a `c4.4xlarge` (16 vCPUs; 32 GB RAM) Amazon Linux general purpose virtual machine. We use `AES-NI`-enabled CPUs for fast PRG evaluations, as well as other (V)DPF-specific optimizations [8, 16]. **All experiments are performed on a single core.**

TABLE 1: Overhead of introducing public-key (pub; Section 3) and symmetric key (sym; Section 3.2) PACLs to DPF and DMPF classes of FSS. As the FSS class becomes more complex (e.g., FSS for DMPFs such as inequality and range functions) the overhead of enforcing access control diminishes. We set the (V)DPF domain to $\{0,1\}^{32}$. All benchmarks are amortized over 100,000 evaluations of the FSS.

	FSS.Eval **Baseline**	DPF-PACL (match predicate)		DPF-PACL (inclusion predicate)		VFSS.Eval **Baseline**	VDPF-PACL (match predicate)		VDPF-PACL (inclusion predicate)	
		sym	pub	sym	pub		sym	pub	sym	pub
DPF	1.41 μs	1.42 μs	5.66 μs	33.10 μs	84.11 μs	1.46 μs	1.47 μs	1.69 μs	33.66 μs	36.11 μs
DMPF	90.44 μs	90.46 μs	93.26 μs	122.22 μs	213.47 μs	93.14 μs	93.19 μs	93.50 μs	125.52 μs	128.71 μs

Methodology. We run each experiment between 10 and 1,000 times (depending on the experiment) and report the average over the trials. 95% confidence interval is occasionally invisible.

Optimizations. In a setting with two verifiers, we observe that Verify simply has each verifier checking that each secret share corresponds to a share of zero. If the parties convert their shares to *subtractive* (rather than additive) secret shares, then this check becomes an equality check (both parties have the *same* subtractive share if it's a share of zero). Therefore, to avoid sending all secret shares, the verifiers can send succinct hashes of their audit shares to reduce communication.

8.1. Prover costs

The proving costs for our PACL constructions are minimal. The prover only needs to generate (V)DPF keys (for inclusion predicates) in our (V)DPF-PACL constructions. The complexity of (V)DPF.Gen is linear in the (V)DPF domain size n [6, 8] and remains below 20 ms for practical values of n (i.e., $n \leq 128$). In our VDPF-PACL construction the prover also has to compute the SPoSS proof. We benchmark SPoSS.Prove at 30 ms of CPU time (bottlenecked by exponentiation in $\mathbb{Z}_p^*$).

8.2. Communication costs

Tables 2 and 3 report the concrete communication overheads on the prover and the verifiers. To remain independent of the underlying DPF construction, we let s_ℓ denote the size of a DPF with range ℓ (e.g., $s_\ell = \log(\ell) \cdot (\lambda + 2)$ bits [8]).

In Table 2, we compare communication costs to other approaches for DPF access control. Express [19] and Sabre [39] operate in the symmetric-key (sym) setting, satisfying the relaxed PACL soundness definition (Appendix A). Newman et al. [31] construct an access control mechanism similar to Algorithm 1 satisfying the soundness definition of Definition 3.2. Using SPoSS for verifying discrete-logarithm knowledge results in over 2,400× less communication compared to a naive approach (described in Appendix E) and a 1,000× smaller proof size compared to Sabre [39].

8.3. Verification costs

We report the processing time in Figures 3 and 4. Introducing PACLs results in a concrete processing overhead relative to evaluating f_i (here, f_i is either a DPF or DMPF), especially when the number of evaluations of the function is small (e.g., less than 64). However, as the number of evaluations increases, the *amortized* cost of access control decreases (the overhead of the group exponentiation in Verify is amortized over the evaluations of f_i). FSS itself is typically only of interest in settings where the function is evaluted on a large number of inputs (otherwise it is more efficient to just secret share $[f(x)]$ rather than $[f]$). As such, it is more reasonable to consider the amortized overhead that access control introduces. For our DPF-PACL construction (reported in Figure 3), the amortized overhead plateaus at approximately 5× the baseline cost of evaluating f_i with around 2^8 evaluations. This is primarily due to the linear number of group (elliptic curve) exponentiations required in the Audit procedure. In contrast, for our VDPF-PACL construction (reported in Figure 4), which requires only a constant number of group exponentiations in $\mathbb{Z}_p^*$, we observe a larger initial overhead but far better amortized overhead. The larger initial overhead is entirely due to the single exponentiation in $\mathbb{Z}_p^*$ (which we benchmark at approximately 13 ms). All our constructions have a lower overhead as

TABLE 2: Proof size (Prover → Verifier) and audit token size (Verifier ↔ Verifier) for (V)DPF-PACL with the match predicate for access control (Algorithm 1). *Naive SPoSS (see Appendix E).

Match predicate	Prover → Verifier	Verifier ↔ Verifier
DPF-PACL	32 B	64 B
VDPF-PACL (SPoSS)	1952 B	816 B
(V)DPF-PACL (sym)	16 B	16 B
VDPF-PACL (naive) *	4.7 MB	816 B
Express [19] (sym)	2 kB	184 B
Spectrum [31]	32 B	64 B
Sabre [39] (sym)	$(40 + 120n)$ kB	16 B

TABLE 3: Proof size (Prover → Verifier) and audit token size (Verifier ↔ Verifier) for (V)DPF-PACL with inclusion predicate for access control (Algorithm 2) and ℓ access keys per function in the FSS family. We denote by s_ℓ the size (in B) of a (V)DPF key with a range of $\{1, \ldots, \ell\}$. Prior techniques [19, 31, 39] do not support inclusion predicates. *Naive SPoSS (see Appendix E).

(inclusion predicate)	Prover → Verifier	Verifier ↔ Verifier
DPF-PACL	$(16 + s_\ell)$ B	32 B
VDPF-PACL (SPoSS)	$(1952 + s_\ell)$ B	816 B
(V)DPF-PACL (sym)	$(16 + s_\ell)$ B	16 B
VDPF-PACL (naive)*	$(4.7 \times 10^6 + s_\ell)$ B	816 B

the complexity of the FSS increases (e.g., when applying PACLs to DMPFs) thanks to the aggregation properties described in Section 4.3. To better understand the asymptotic amortization of our constructions, we report the tail values of Figures 3 and 4 in Table 1, where we amortize over 100,000 evaluations of the DPF and DMPF.

8.4. Applications of PACLs

Private databases with access control. Systems that use multi-server PIR (e.g., [13–15, 24, 27, 40]) can take advantage of (V)DPF-PACLs to restrict access to database items. (Gupta et al. [24] specifically leave open the problem of supporting *authenticated* media consumption through Popcorn.) Other systems such as Dory [14] use DPFs for private keyword queries in a remote database. For example, Wang et al. [40] use PIR (realized using DPFs) to build privacy-preserving restaurant, geolocation, and flight searches. Gupta et al. [24] use PIR for privacy-preserving movie streaming. In Figure 5, we report the overhead of introducing access control in PIR via VDPF-PACLs. As the items in the database become larger, the overhead of introducing access control diminishes. Part of the overhead from introducing PACLs to PIR is due to switching from operations in a binary field (xors) to operations in $\mathbb{Z}_p$, which are concretely slower. Ostrovsky and Shoup [32] describe a read-and-write private database, which can be realized using DPFs [8]. Applying VDPF-PACLs to this setting would result in similar overheads to the PIR setting.

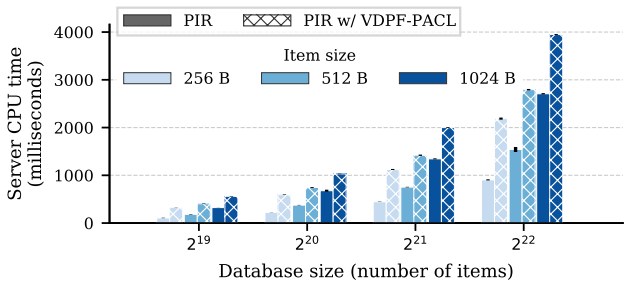


Figure 5: Server-side CPU time for privately retrieving an item from a database through two-server PIR with and without VDPF-PACLs. Adding access control has a 1.5–3× impact on processing time. The access control overhead is automatically amortized over the number of items in the database. The relative overhead also diminishes as the items in the database become larger.

Anonymous authentication. We identify a potentially interesting application of PACLs for the purpose of anonymous user authentication (for instance, password-based authentication [23]) in a distributed setting. Denote each user pseudonym by i and let sk_i be the corresponding key (or password). We can set Λ to be the set of valid account keys and let f_i be the point function with special index i. Applying our VDPF-PACL construction over $[f_i]$, the verifiers can learn if user i has a valid account (and knows the key sk_i associated with vk_i) *without* learning which account was used to authenticate. In Table 4, we benchmark the processing time required to authenticate a user as a function of the number of accounts in the database.

TABLE 4: Evaluation of VDPF-PACLs applied to anonymous user authentication with varying number of accounts (evaluation points).

Anonymous authentication with VDPF-PACLs				
Number of accounts:	250K	500K	1M	2M
Authentication time:	103 ms	192 ms	381 ms	757 ms

Faster anonymous communication. Our VDPF-PACL construction can be applied out-of-the-box to the anonymous communication systems Express [19], Sabre [39] and Spectrum [31] to improve their concrete performance. In Figure 6, we show that swapping their implicit access control mechanism with our VDPF-PACL construction improves performance by a factor of 50–70×. Sabre [39]'s computational overhead is on-par with baseline FSS (and DPF-PACLs satisfying symmetric-key soundness Appendix A and Table 2) but requires significantly larger proofs for access control purposes (approximately 3.5 MB; see Table 2). However, we note that Sabre [39] achieves other nice properties that are tailored to anonymous communication (see Section 1.2).

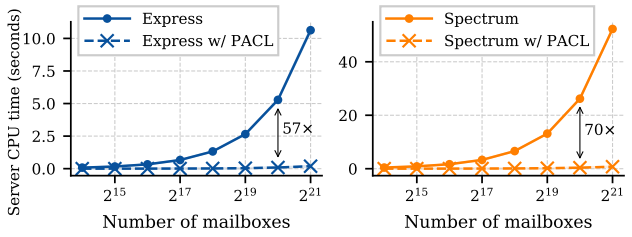


Figure 6: Our VDPF-PACLs significantly improve performance of anonymous communication systems that require access control. These performance improvements come from two sources: (1) the ability to use optimized DPF constructions with our DPF-PACLs (e.g., the access control in Express requires concretely slower DPF constructions) and (2) the better amortization of VDPF-PACLs.

9. Conclusion

We modeled and formalized the notion of private access control for FSS. Our constructions can be applied to a variety of FSS applications and improve the performance of ad-hoc methods found in prior work. We also present a generic theoretical construction that has exciting potential for future work. Finally, we evaluate our constructions and showcase their performance on several concrete use cases, ranging from anonymous authentication and communication to access control in private databases. Our evaluation shows that introducing access control results in minimal overheads relative to baseline FSS, and amortizes well asymptotically when the function is evaluated on many inputs.

Acknowledgements

We would like to thank Srini Devadas for his generous support and for reading several drafts of this paper. We would also like to thank Anish Athalye, Leo de Castro, Kyle Hogan, Simon Langowski, Manon Revel, and Mayuri Sridhar for helpful discussions, comments, and suggestions. We thank the USENIX 2022 reviewers for providing us with exceptionally thorough feedback on an earlier version of this work and the S&P 2023 reviewers and our shepherd for their incredibly detailed comments that helped us to significantly improve the presentation of the paper. Finally, we would like to thank the organizers of the MIT PRIMES program for making it a such a fun and enriching experience.

References

[1] Source code. https://github.com/sachaservan/pacl.

[2] Donald Beaver. Efficient multiparty protocols using circuit randomization. In *Annual International Cryptology Conference*. Springer, 1991.

[3] Dan Boneh and Victor Shoup. A graduate course in applied cryptography. *Draft 0.5*, 2020.

[4] Dan Boneh, Elette Boyle, Henry Corrigan-Gibbs, Niv Gilboa, and Yuval Ishai. Zero-knowledge proofs on secret-shared data via fully linear PCPs. In *Annual International Cryptology Conference*. Springer, 2019.

[5] Dan Boneh, Elette Boyle, Henry Corrigan-Gibbs, Niv Gilboa, and Yuval Ishai. Lightweight techniques for private heavy hitters. In *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2021.

[6] Elette Boyle, Niv Gilboa, and Yuval Ishai. Function secret sharing. In *Annual international conference on the theory and applications of cryptographic techniques*. Springer, 2015.

[7] Elette Boyle, Niv Gilboa, and Yuval Ishai. Breaking the circuit size barrier for secure computation under DDH. In *Annual International Cryptology Conference*. Springer, 2016.

[8] Elette Boyle, Niv Gilboa, and Yuval Ishai. Function secret sharing: Improvements and extensions. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016.

[9] Elette Boyle, Nishanth Chandran, Niv Gilboa, Divya Gupta, Yuval Ishai, Nishant Kumar, and Mayank Rathee. Function secret sharing for mixed-mode and fixed-point secure computation. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2021.

[10] Dario Catalano and Dario Fiore. Vector commitments and their applications. In *International Workshop on Public Key Cryptography*. Springer, 2013.

[11] Benny Chor, Oded Goldreich, Eyal Kushilevitz, and Madhu Sudan. Private information retrieval. In *Proceedings of IEEE 36th Annual Foundations of Computer Science*. IEEE, 1995.

[12] Henry Corrigan-Gibbs and Dan Boneh. Prio: Private, robust, and scalable computation of aggregate statistics. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, 2017.

[13] Henry Corrigan-Gibbs, Dan Boneh, and David Mazières. Riposte: An anonymous messaging system handling millions of users. In *2015 IEEE Symposium on Security and Privacy*. IEEE, 2015.

[14] Emma Dauterman, Eric Feng, Ellen Luo, Raluca Ada Popa, and Ion Stoica. DORY: An encrypted search system with distributed trust. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, 2020.

[15] Emma Dauterman, Mayank Rathee, Raluca Ada Popa, and Ion Stoica. Waldo: A private time-series database from function secret sharing. In *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2022.

[16] Leo de Castro and Anitgoni Polychroniadou. Lightweight, maliciously secure verifiable function secret sharing. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2022.

[17] Yevgeniy Dodis, Shai Halevi, Ron D Rothblum, and Daniel Wichs. Spooky encryption and its applications. In *Annual International Cryptology Conference*. Springer, 2016.

[18] Jack Doerner and abhi shelat. Scaling ORAM for secure computation. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017.

[19] Saba Eskandarian, Henry Corrigan-Gibbs, Matei Zaharia, and Dan Boneh. Express: Lowering the cost of metadata-hiding communication with cryptographic privacy. In *30th USENIX Security Symposium (USENIX Security 21)*, 2021.

[20] Amos Fiat and Adi Shamir. How to prove yourself: Practical solutions to identification and signature problems. In *Conference on the theory and application of cryptographic techniques*. Springer, 1986.

[21] Craig Gentry, Philip MacKenzie, and Zulfikar Ramzan. A method for making password-based key exchange resilient to server compromise. In *Annual International Cryptology Conference*. Springer, 2006.

[22] Niv Gilboa and Yuval Ishai. Distributed point functions and their applications. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2014.

[23] Matthew Green. Let's talk about PAKE, october 2018. https://blog.cryptographyengineering.com/2018/10/19/lets-talk-about-pake/, 2018. Accessed December 2021.

[24] Trinabh Gupta, Natacha Crooks, Whitney Mulhern, Srinath Setty, Lorenzo Alvisi, and Michael Walfish. Scalable and private media consumption with popcorn. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, 2016.

[25] Ryan Henry, Femi Olumofin, and Ian Goldberg. Practical PIR for electronic commerce. In *Proceedings of the 18th ACM conference on Computer and communications security*, 2011.

[26] Yizhou Huang and Ian Goldberg. Outsourced private information retrieval. In *Proceedings of the 12th ACM Workshop on Workshop on Privacy in the Electronic Society*, 2013.
[27] Ari Juels. Targeted advertising... and privacy too. In *Cryptographers' Track at the RSA Conference*. Springer, 2001.
[28] Jonathan Katz and Yehuda Lindell. *Introduction to modern cryptography*. CRC press, 2020.
[29] Ralph C Merkle. A digital signature based on a conventional encryption function. In *Conference on the theory and application of cryptographic techniques*. Springer, 1987.
[30] Pratyush Mishra, Ryan Lehmkuhl, Akshayaram Srinivasan, Wenting Zheng, and Raluca Ada Popa. Delphi: A cryptographic inference service for neural networks. In *29th USENIX Security Symposium (USENIX Security 20)*, 2020.
[31] Zachary Newman, Sacha Servan-Schreiber, and Srinivas Devadas. Spectrum: High-bandwidth anonymous broadcast. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. USENIX Association, April 2022.
[32] Rafail Ostrovsky and Victor Shoup. Private information storage. In *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, 1997.
[33] Théo Ryffel, Pierre Tholoniat, David Pointcheval, and Francis R. Bach. Ariann: Low-interaction privacy-preserving deep learning via function secret sharing. *Proc. Priv. Enhancing Technol.*, 2022(1):291–316, 2022.
[34] Claus-Peter Schnorr. Efficient signature generation by smart cards. *Journal of cryptology*, 4(3), 1991.
[35] Sacha Servan-Schreiber, Simon Beyzerov, Eli Yablon, and Hyojae Park. Private access control for function secret sharing. Cryptology ePrint Archive, Paper 2022/1707, 2022.
[36] Adi Shamir. How to share a secret. *Communications of the ACM*, 22(11), 1979.
[37] Joseph H Silverman. *The arithmetic of elliptic curves*, volume 106. Springer, 2009.
[38] John T Tate. The arithmetic of elliptic curves. *Inventiones mathematicae*, 23(3), 1974.
[39] A. Vadapalli, K. Storrier, and R. Henry. Sabre: Sender-anonymous messaging with fast audits. In *2022 2022 IEEE Symposium on Security and Privacy (SP) (SP)*, pages 1326–1343, Los Alamitos, CA, USA, may 2022. IEEE Computer Society.
[40] Frank Wang, Catherine Yun, Shafi Goldwasser, Vinod Vaikuntanathan, and Matei Zaharia. Splinter: Practical private queries on public data. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, 2017.

Appendix

Appendix A. Symmeric-key PACL soundness

Definition A.1 (PACL: Symmetric-key soundness)**.** There exists a negligible function negl and security parameter $\lambda \in \mathbb{N}$ such that for all efficient algorithms $\mathcal{A}$ and subsets $I \subseteq \{1, \ldots, s\}$ where $|I| \geq t$,

$$\Pr[\textsc{SkSoundness}_{\mathrm{PACL},\mathcal{A},I}(\lambda) = \mathsf{yes}] \leq \mathsf{negl}(\lambda),$$

where $\textsc{SkSoundness}_{\mathrm{PACL},\mathcal{A},I}(\lambda)$ is defined in Figure 7.

Game $\textsc{SkSoundness}_{\mathrm{PACL},\mathcal{A},I}(\lambda)$	Oracle $\textsc{GetKey}(j)$
for $i \in \{1, \ldots, N\}$:	$T := T \cup \{j\}$
$\quad(\_, \mathsf{sk}_i) \leftarrow \mathsf{KeyGen}(1^\lambda, f_i)$	**return** sk_j
$\Lambda := (\mathsf{sk}_1, \ldots, \mathsf{sk}_N),\ T = \{\}$	
$([f_\gamma], [\pi_\gamma]) \leftarrow \mathcal{A}^{\textsc{GetKey}}(1^\lambda)$	
$f_\gamma \leftarrow \mathsf{Recover}([f_\gamma])$	
foreach $i \in I$:	
$\quad\tau_i \leftarrow \mathsf{Audit}(\Lambda, [f_\gamma]_i, [\pi_\gamma]_i)$	
return $\mathsf{Verify}(\{\tau_i \mid i \in I\}) = \mathsf{yes}$	
$\quad$ **and** $f_\gamma \in \mathcal{F}$ **and** $\gamma \notin T$	

Figure 7: Symmetric-key PACL access soundness game.

In words, no efficient algorithm $\mathcal{A}$, without knowledge of the access key, can forge a proof π that verifies with non-negligible probability. Unlike Definition 3.2, here Λ is private to the verifiers and is not given to $\mathcal{A}$.

Appendix B. Verifiable DPFs

de Castro and Polychroniadou [16] present definitions for $(2, 2)$-VDPF constructions. We generalize their definitions to any (t, s)-VDPF scheme.

Definition B.1 (VDPF [16])**.** Let $\lambda \in \mathbb{N}$ be a security parameter and $\mathbb{F}$ be any finite field. Fix a domain $\{0,1\}^n$. A VDPF consists of three (possibly randomized) algorithms $(\mathsf{Gen}, \mathsf{Eval}, \mathsf{Verify})$:

- $\mathsf{Gen}(1^\lambda, i \in \{0,1\}^n, m \in \mathbb{F}) \to (\kappa_1, \ldots, \kappa_s)$. Takes as input a security parameter, an index $i \in \{0,1\}^n$, and message m. Outputs a set of evaluation keys encoding point function $P_{i,m}$ such that $P_{i,m}(i) = m$.
- $\mathsf{Eval}(\kappa_j, X \subseteq \{0,1\}^n) \to ([\boldsymbol{v}]_j, \rho_j)$. Takes as input an evaluation key κ_j and a subset of values in the domain. Outputs a secret share of a vector $\boldsymbol{v}$, where the kth coordinate of $\boldsymbol{v}$ corresponds to a share of $P_{i,m}(k)$ for $k \in X$, and a verification string ρ_j.
- $\mathsf{Verify}(\{\rho_j \mid j \in J\}) \to \mathsf{yes}/\mathsf{no}$. Takes as input any subset of t or more verification strings indexed by the set

$J \subseteq \{1, \ldots, s\}$. Outputs yes if and only if $\{[\boldsymbol{v}]_j \mid j \in J\}$, as output by Eval using $\{\kappa_j \mid j \in J\}$, encodes a point function on the evaluated set of inputs X.

A (t, s)-VDPF must satisfy the *correctness*, *privacy*, and *efficiency* properties of FSS (Definition 2.2). Additionally, a VDPF must guarantee *soundness*. Informally, soundness requires that Verify outputs no for any set of evaluation keys that do not encode a point function on the evaluated points.

- **Correctness.** A (t, s)-VDPF is correct if for all $k \in \{0,1\}^n$, for all $m \in \mathbb{F}$, and all subsets $J \subseteq \{1, \ldots, s\}$, such that $|J| \geq t$, there exists an efficient algorithm Decode such that

$$\Pr\left[\begin{array}{l}(\kappa_1, \ldots, \kappa_s) \leftarrow \mathsf{Gen}(1^\lambda, i, m): \\ \mathsf{Decode}(\{\mathsf{Eval}(\kappa_j, \{k\}) \mid j \in J\}) = P_{i,m}(k)\end{array}\right] = 1,$$

  where the probability is over Gen.

- **Soundness.** A (t, s)-VDPF is sound if for all (possibly maliciously generated) keys $(\kappa_1^*, \ldots, \kappa_s^*)$, adversarially chosen inputs $X^* \subseteq \{0,1\}^n$, and ρ_j sampled according to $(\_, \rho_j) \leftarrow \mathsf{Eval}(\kappa_j^*, X^*)$ for $j \in J$, $|J| \geq t$, it holds that if the correctness property is not satisfied then:

$$\Pr[\mathsf{Verify}(\{\rho_j \mid j \in J\}) = \mathsf{yes}] \leq \mathsf{negl}(\lambda),$$

  where the probability is taken over the adversary's randomness. Otherwise, $\mathsf{Verify}(\{\rho_j \mid j \in J\}) = \mathsf{yes}$ with probability 1.

- **Privacy.** For all subsets $I \subset \{1, \ldots, s\}$ such that $|I| < t$, define $J := \{1, \ldots, s\} \setminus I$ and $\mathcal{D}_{I,J}$ to be the distribution over $\{(\kappa_i, \rho_i^*) \mid i \in I\} \cup \{\rho_i \mid j \in J\}$ where κ_i is sampled according to Gen, each ρ_i^* is sampled arbitrarily, and each ρ_i is sampled according to Eval. A (t, s)-VDPF is private if there exists an efficient simulator $\mathcal{S}$ such that $\mathcal{D}_{I,J} \approx_c \mathcal{S}(1^\lambda, I, \{\rho_i^* \mid i \in I\})$. That is, all subsets of fewer than s evaluation keys and the entire set of verification strings (of which $t-1$ might be maliciously generated), reveal no information on the point function encoded in the set of keys $(\kappa_1, \ldots, \kappa_s)$.

Appendix C. Deferred proofs

C.1. Proof of Lemma 1

The proof hinges on the aggregation property of our construction (Section 4.3). Consider an efficient $\mathcal{A}$ that outputs $\hat{f}_\gamma$ and $\hat{\pi}$ where $\hat{f}$ is *not* a point function (and also not the trivial identity function $f(x) = 0$ for all x). Then, it holds that $\hat{f}_\gamma = \sum_{j \in S} a_j f_j$, where $S \subseteq \{0, 1, \ldots, N\}$, each f_j is a point function, and a_j are arbitrary scalars in $\mathbb{Z}_p \setminus \{0\}$. Construct an adversary $\mathcal{A}'$ that breaks the $\textsc{PkSoundness}_{\mathrm{PACL},\mathcal{A},I}(\lambda)$ game with $f_\gamma = P_\gamma$ (a point function) as follows. First, run $\mathcal{A}$ to get function $\hat{f}_\gamma$. Then compute $f_\gamma := \hat{f}_\gamma - \sum_{j \in S, j \neq \gamma} a_j f_j$ and $\pi := \hat{\pi} - \sum_{j \in S, j \neq \gamma} a_j \alpha_j$ (recall that $\mathcal{A}'$ is allowed to query for all α_j provided $j \neq \gamma$). Finally, output f_γ and π. It must hold that $\gamma \in S$ (if this were not the case then $\mathcal{A}$ does not succeed as it queried all the necessary access keys sk_j for $j \in S$). By the aggregation properties of our construction (described in Section 4.3), it follows that f_γ is a point function and π is a valid access proof for f_γ. Thus, $\mathcal{A}'$ succeeds with the same probability as $\mathcal{A}$.

C.2. Proof of Theorem 4 (security of Algorithm 2)

Completeness. Consider $C \in \mathbb{G}$ as computed in Verify:

$$C := \left(\prod_{j=1}^{\ell} A_j\right) \cdot \left(\prod_{j=1}^{N,\ell} g^{-c_j \cdot w_j}\right) \cdot g^\alpha.$$

Examining "the exponent," we get that:

$$\log_g(C) = \sum_{j=1}^{N} \sum_{k=1}^{\ell} \alpha_{j,k} \cdot y_j - \left(\sum_{j=1}^{N\ell} w_j \cdot c_j\right) + \alpha.$$

If f_i is a DPF instance, then $y_j = 1$ only for $j = i$. Thus,

$$\log_g(C) = \sum_{k=1}^{\ell} \alpha_{i,k} - \left(\sum_{j=1}^{N\ell} w_j \cdot c_j\right) + \alpha.$$

Further, if $(\kappa'_1, \ldots, \kappa'_s)$ encode a DPF for the $(i-1)\ell + \gamma = \omega$th index, then all $c_j = 0$ for $j \neq \omega$. Therefore,

$$\log_g(C) = \sum_{k=1}^{\ell} \alpha_{i,k} - w_\omega + \alpha.$$

However, by construction, $w_\omega = \sum_{k=1, k \neq \gamma}^{\ell} \alpha_{i,k}$, and so we get that: $\log_g(C) = \alpha_{i,\gamma} + \alpha = 0$, by construction since $\alpha = -\alpha_{i,\gamma}$. Therefore, it holds that $C = g^0 = 1_{\mathbb{G}}$ and Verify outputs yes, making the construction complete.

Soundness. Assume, towards contradiction, that there exists an efficient $\mathcal{A}$ that wins the $\textsc{PkSoundness}_{\mathrm{PACL},\mathcal{A},I}(\lambda)$ game with non-negligible probability $\delta(\lambda)$. Then, $\mathcal{A}$ outputs secret shares of $\hat{f}_\gamma$ (corresponding to point function P_γ) encoded as keys $(\hat{\kappa}_1, \ldots, \hat{\kappa}_s)$ and proof shares $([\hat{\pi}]_1, \ldots, [\hat{\pi}]_s)$ such that for all $I \subseteq \{1, \ldots, s\}$ where $|I| \geq t$,

$$\Pr\left[\begin{array}{l}\tau_i \leftarrow \mathsf{Audit}(\Lambda, [\hat{f}_\gamma]_i, [\hat{\pi}]_i), \forall i \in I: \\ \mathsf{Verify}(\{\tau_i \mid i \in I\}) = \mathsf{yes}\end{array}\right] \geq \delta(\lambda).$$

Without loss of generality (by Lemma 1), we can assume that $\mathcal{A}$ outputs $\hat{f}_\gamma$ sampled from the family of point functions when considering Algorithm 2. For notational simplicity, we "expand" the N verification keys in Λ (each containing ℓ subkeys) so as to make Λ consist of $N\ell$ verification keys. We now construct an efficient algorithm $\mathcal{B}$ that solves the discrete logarithm problem as follows. On input $y := g^x$,

1: $(\alpha_{1,1}, \ldots, \alpha_{N,(\ell-1)}) \leftarrow_R \mathbb{Z}_p^{N \times (\ell-1)}$.
2: $\Lambda := (g^{\alpha_{1,1}}, \ldots, g^{\alpha_{1,\ell}}, \ldots, g^{\alpha_{N,1}}, \ldots, g^{\alpha_{N,(\ell-1)}}, y)$.
3: Run $\mathcal{A}^{\mathsf{GetKey}}(1^\lambda, \Lambda)$ and answer each $\mathsf{GetKey}(i)$ query with $(\alpha_{i,1}, \ldots, \alpha_{i,\ell})$ for all $i \neq N$. If $\mathcal{A}$ queries GetKey on input N, then **abort**.
4: Obtain output $([\hat{f}_\gamma], [\hat{\pi}])$ from $\mathcal{A}$.
5: Recover $\hat{f}_\gamma$ and $(\hat{f}_\omega, \hat{\alpha}, \hat{\beta})$ from $[\hat{f}_\gamma]$ and $[\hat{\pi}]$, respectively.
6: Output $\hat{\alpha}$.

By the aggregation properties of our DPF-PACL constructions and Lemma 1 and soundness of Algorithm 1 (Section 4.4), it holds that $\hat{\alpha}$ is the discrete logarithm of y whenever $\mathcal{A}$ succeeds in $\text{PKSOUNDNESS}_{\text{PACL},\mathcal{A},I}(\lambda)$ with a DPF f_N and using the ℓth key in Λ for f_N. The probability that $\mathcal{A}$ outputs f_N is $\frac{1}{N}$ and the probability $\mathcal{A}$ uses the ℓth key for row N is $\frac{1}{\ell}$. Thus, $\mathcal{B}$ succeeds with probability at least $\frac{1}{N\ell}\delta(\lambda)$ which remains non-negligible, contradicting the hardness of the discrete logarithm problem.

Privacy. We construct an efficient simulator $\mathcal{S}$ for the view of any subset of at most $t-1$ (possibly malicious) verifiers. Let $\mathcal{S}'$ be the simulator in the proof of Theorem 3. On input $(1^\lambda, I, \{\tau_i^* \mid i \in I\})$, $\mathcal{S}$ proceeds as follows:

$J := \{1, \ldots, s\} \setminus I$.
parse $\tau_i^* = (\hat{\tau}_i^{(0)}, \hat{\tau}_i^{(1)})$ for all $i \in I$.
$([0]_1, \ldots, [0]_s) \leftarrow \mathsf{Share}_{(\mathbb{Z}_p,t,s)}(0)$.
$([z']_1, \ldots, [z']_s) \leftarrow \mathsf{Share}_{(\mathbb{Z}_p,t,s)}(0)$.
$(\kappa'_1, \ldots, \kappa'_s) \leftarrow \mathsf{DPF.Gen}(1^\lambda, P_1)$.
$\mathsf{view} \leftarrow \mathcal{S}'(1^\lambda, I, \{\hat{\tau}_i^{(0)} \mid i \in I\})$.
parse $\mathsf{view} = \left\{\left([\pi']_i, \tau_i^{(0)}\right) \mid i \in I\right\} \cup \left\{\tau_j^{(0)} \mid j \in J\right\}$.
$[\pi]_i := ([z']_i, [\pi']_i, \kappa'_i)$.
$\tau_k^{(1)} := g^{[0]_k}$ for all $k \in I \cup J$.
$\tau_k := (\tau_k^{(0)}, \tau_k^{(1)})$ for all $k \in I \cup J$.
Output $\{([\pi]_i, \tau_i) \mid i \in I\} \cup \{\tau_j \mid j \in J\}$.

The distribution output by $\mathcal{S}$ matches the distribution of any subset $I \subset \{1, \ldots, s\}$, where $|I| < t$ because: (1) any subset of $\{[0]_1, \ldots, [0]_s\}$ of size $< t$ is uniformly distributed and therefore matches the distribution of any subset of $\{[\alpha]_1, \ldots, [\alpha]_s\}$ of size $< t$ in the real view, (2) $[\pi']_i$ is guaranteed to be computationally indistinguishable by the proof of Theorem 3, (3) the DPF key for point function P_ω (in the real view) is computationally indistinguishable to κ'_i corresponding to point function P_1 by the privacy of FSS (Definition 2.2), and (4) the audit tokens are (computationally-hiding) multiplicative secret shares of $(1_\mathbb{G}, 1_\mathbb{G})$ in the real view and uniformly random multiplicative secret shares in the output of $\mathcal{S}$. An efficient distinguisher for (4) would also contradict the privacy property of FSS.

Efficiency. Using the "FSS tensoring" [8] optimization (see Section 4.1.2), the size of each proof share is $O(\lambda + s_\ell)$ where s_ℓ is the size of a DPF key encoding a point function with range $\{1, \ldots, \ell\}$.

C.3. Security of SPoSS

Proposition 1. The SPoSS construction in Algorithm 3 satisfies the correctness, argument-of-knowledge, and zero-knowledge properties (Definition 5.1) required of a secret-shared non-interactive proof system [4].

Proof of Proposition 1. We prove each property in turn: *completeness*, *argument-of-knowledge*, and *zero-knowledge*.

Completeness. We show that if the prover is honest, then Verify outputs yes. In Algorithm 3, Verify outputs yes if and only if $w_A + w_B = 0$ and $\hat{r} = r$, $\hat{d} = d$, and $\hat{e} = e$. The equality of $\hat{r} = r$, $\hat{d} = d$, and $\hat{e} = e$ follows by inspection. To see why it holds that $w_A + w_B = 0$, observe that

$$\begin{aligned}
w_A + w_B &= v_A - ry_A + v_B - ry_B \\
&= v - ry \\
&= 2(\frac{de}{2}) + ea + db + (c_A + c_B) - ry \\
&= \underbrace{(r\hat{y}_A - a)}_{d}\underbrace{(\hat{y}_B - b)}_{e} + ea + db + c - ry \\
&= r\hat{y} - a(\hat{y}_B) + a(\hat{y}_B) - ry \\
&= r(\hat{y} - y) = 0 \text{ by assumption that } \hat{y} = g^x = y.
\end{aligned}$$

Argument-of-knowledge. We construct an efficient extractor $\mathcal{E}$ that recovers the discrete logarithm of y from a proof $[\pi]$ output by a possible malicious prover $\mathcal{P}^*$. $\mathcal{E}$ proceeds as follows:

Run $\mathcal{P}^*(y)$ to obtain as output (π_A, π_B) where

$$\pi_A = (\text{A}, [x]_A, a, [c]_A, r, d, e, z_A)$$
$$\pi_B = (\text{B}, [x]_B, b, [c]_B, r, d, e, z_B).$$

Output $x = [x]_A + [x]_B$.

If (π_A, π_B) is a valid SPoSS proof valid, then $w_A + w_B = 0$. In turn, we have that $r\hat{y} - (r\hat{y}_A)b - a(\hat{y}_B) + ab + ea + db - ry + c = 0$ for some randomness r (see completeness proof). The malicious prover $\mathcal{P}^*$ can choose arbitrary a, b, c. As such, we have that $c = ab + \Delta$ for some Δ [12], which yields $r\hat{y} - (r\hat{y}_A)b - a(\hat{y}_B) + 2ab + ea + db - ry + \Delta$ which reduces to $r(\hat{y} - y) + \Delta = 0$. Thus, either (1) the malicious prover obtained r from H such that $r(\hat{y} - y) = -\Delta$ (which happens with negligible probability given H is a random oracle) or (2) $\hat{y} = y$ and $\Delta = 0$ which implies that $(\hat{y} - y) = 0$ and therefore $g^{[x]_A + [x]_B} = \hat{y} = y$ and so $x = [x]_A + [x]_B$ is the discrete logarithm, as required.

Zero-knowledge. To prove that SPoSS is zero-knowledge, we construct an efficient simulator $\mathcal{S}$ that given $i \in \{\text{A}, \text{B}\}$ and τ_i^*, outputs a statistically indistinguishable view to that of verifier i (the simulator generalizes to the many-verifier case). On input $(1^\lambda, \{i\}, \{\tau_i^*\})$, $\mathcal{S}$ proceeds as follows:

$j \in \{\text{A}, \text{B}\} \setminus \{i\}$.
$([w]_A, [w]_B) \leftarrow \mathsf{Share}_{(\mathbb{F}_p,2,2)}(0)$.
$[x]_i \leftarrow_R \mathbb{Z}_{p-1}$.
$[c]_i, u, d, e \leftarrow_R \mathbb{F}_p$.
$\hat{r}_i, r, \leftarrow_R \mathbb{Z}_p \setminus \{0\}$.
$z_i \leftarrow_R \{0,1\}^\lambda$.
$[\pi]_i := ([x]_i, u, [c]_i, r, d, e, z_i)$.
$\tau_A \leftarrow ([w]_A, \hat{r}_i, r, d, d, e)$.
$\tau_B \leftarrow ([w]_B, r - \hat{r}_i, r, e, d, e)$.
Output $\{([\pi]_i, \tau_i)\} \cup \{\tau_j\}$.

Analysis. Consider the distribution of $[\pi]_i$ in the real view of the ith verifier. Observe that $[\pi]_i$ consists of (1) secret shares

$[x]_i$ and $[c]_i$, (2) masks z_i and u (see optimization described in Section 5.1.1), (3) Beaver multiplication openings d, e, and (4) the distributed Fiat-Shamir randomness r. All these values are generated by the prover. (1), (2) and (3) are uniformly distributed. (4) is uniformly distributed due to the mask z_i [4]. As such, the distribution of $[\pi]_i$, as output by $\mathcal{S}$, matches that of the real view of the ith verifier.

Now, consider τ_j, which consists of (1) a secret share $[w]_i$, (2) random oracle outputs $\hat{r}$ and r (which are identical when the prover is honest and distributed uniformly due to the mask z_j) and (3) Beaver multiplication openings f (computed by verifier j and d, e given by the prover). (1) and (3) are uniformly distributed due to $[c]_j$ being uniform share and the mask u_j. Moreover, f is either d or e and thus provides no new information. (2) reveals no new information because $\hat{r} = r$ and r is given to all verifiers. We conclude that the output distribution of $\mathcal{S}$ is distributed identically to the view of verifier i in Algorithm 3, which concludes the proof of zero-knowledge.

Appendix D. Beaver's Protocol

We provide a brief overview of Beaver's [2] approach to multiplication of secret shares as adapted by Corrigan-Gibbs and Boneh [12]. We focus on the setting with two parties; see [12] for a more general exposition. Given two parties holding additive shares $[x]$ and $[y]$, encoding field elements x and y, the parties must securely compute shares of $[xy]$, encoding the value $xy \in \mathbb{F}$. A Beaver triple consists of additive shares of $([a], [b], [c])$ such that a and b are random field elements and $c := ab \in \mathbb{F}$. If the parties are given shares of a Beaver triple, then the parties can compute a secret share encoding the product two secret shares x and y as follows. Each party locally computes:

$$[d] \leftarrow [x] - [a] \quad \text{and} \quad [e] \leftarrow [y] - [b],$$

and broadcasts its shares of d and e. The parties recover d and e and locally compute: $[xy] := d[b] + e[a] + [c] + \frac{de}{2}$. This works because

$$\begin{aligned} d[b] &= (x-a)[b] = [xb - ab], \\ e[a] &= (y-b)[a] = [ya - ab], \\ \frac{de}{2} &= \frac{xy - xb - ay + ab}{2} \end{aligned}$$

($\frac{de}{2}$ is a 2-out-of-2 "share" of $de = xy - xb - ay + ab$) and so we get that:

$$\begin{aligned} d[b] + e[a] + \frac{de}{2} + [c] = \\ &= [xb - ab] + [ya - ab] + [xy - xb - ay + ab] + [c] \\ &= [xb - ab + ya - ab + xy - xb - ay + ab + c] \\ &= [xy - ab + c] \\ &= [xy]. \end{aligned}$$

As such, Beaver's technique reduces the rounds of communication required to compute a multiplication over secret shares down to one round [2].

Appendix E. Naive SPoSS using SNIPs

Here, we estimate the overhead of naively applying a SNIP for verifying a Schnorr proof over secret shares. Verification in Schnorr requires computing an exponentiation in $\mathbb{G}$. This translates to an exponentiation with a secret-shared exponent in our case. Using the textbook approach to modular exponentiation (e.g., repeated squaring) requires one multiplication per bit of the group order. When $\mathbb{G} = \mathbb{Z}_p^*$ we have $\log p \approx 3072$ for security. We let $\lambda := \lceil \log p \rceil$. The SNIP proof requires sending one Beaver triple (3λ bits per verifier) and two elements of $\mathbb{Z}_p$ per multiplication gate in the circuit (the proof consists of a degree-$2M$ polynomial interpolating the multiplication gates). We now describe the arithmetic circuit computing an exponentiation. We note that the prover can secret-share the *bit decomposed* exponent as part of the proof. In this case, the verifiers only need to check that the secret shares encode a binary number and then apply the group operation (multiplication in $\mathbb{Z}_p$) λ times to computed the repeated squaring circuit. First, this requires the verifiers to check that each secret-shared bit a_i for $i \in \{1, \ldots, \lambda\}$ is either 0 or 1. The arithmetic circuit computing this check is defined as $C(a_i) = 1 + a_i^2 - a_i$. Therefore, the arithmetic circuit for checking the validity of the binary decomposition requires λ multiplication gates and makes the SNIP proof consist of 2λ elements of $\mathbb{Z}_p$. The total size, in bits, is therefore $2\lambda^2$.

Second, the verifiers much check the repeated squaring circuit, which requires computing the group operation (one multiplication in $\mathbb{Z}_p$) λ times. This makes the SNIP proof consist of 2λ elements of $\mathbb{Z}_p$. The total size of the repeated squaring proof, in bits, is therefore $2\lambda^2$ as well.

Combined, the total proof size is:

$$\begin{aligned} &\underbrace{(3 \cdot 3072)}_{\text{Beaver triple}} + \underbrace{2 \cdot (3072)^2}_{\text{binary check}} + \underbrace{2 \cdot (3072)^2}_{\text{repeated squaring}} \text{ bits} \\ &= (9216) + 2 \cdot (3072)^2 + 2 \cdot (3072)^2 \text{ bits} \\ &\approx 4.7 \text{ MB}. \end{aligned}$$

Note that using an elliptic curve instead of $\mathbb{Z}_p^*$ (which would allow us to work over a field of roughly order $p \approx 2^{256}$ instead of $p \approx 2^{3072}$) does not improve the situation. While the proof size for the validity of the binary decomposition (checking that $C(a_i) = 1$, for all i) would be roughly $12\times$ smaller, these savings are negated by the complexity of the elliptic curve group operation, which requires multiple field multiplications to compute the group operation [37, 38]. The advantage of working with $\mathbb{Z}_p^*$ is that we only require *one* multiplication in $\mathbb{Z}_p$ to apply the group operation.