Private Access Control for Function Secret Sharing

Sacha Servan-Schreiber[1], Simon Beyzerov[2,3], Eli Yablon[2,4], and Hyojae Park[2,4]

[1] MIT CSAIL
[2] MIT PRIMES
[3] Massachusetts Academy of Math and Science at WPI
[4] Sharon High School

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 $f(x)$ on an input x, without learning information about f.

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:

- anonymous authentication in a multi-party setting,
- access control in private databases, and
- 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× 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× when amortized over databases with 500,000 or more items.

Keywords: Function secret sharing, verifiable FSS, access control, authentication, anonymous communication, private databases, zero-knowledge multi-verifier proofs

Table of Contents

1 Introduction

Function secret sharing (FSS) [9, 11] is at the core of many privacy-preserving systems, including private databases [21, 22, 50], private telemetry [8], privacy-preserving machine learning [37, 42], distributed oblivious RAM (ORAM) [26], anonymous communication [19, 27, 38, 49], and efficient multi-party computation [12]. 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 [27, 38, 49]. For example, in applications of FSS that involve privately reading from—or writing to—a database [8, 26, 27, 38, 49], 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 all 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 [27, 38, 49], highlighting the utility of formally studying this paradigm.

For example, FSS is often used for private information retrieval (PIR) [9, 17, 30, 50]. 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 $[f_i(\mathcal{DB})]$, 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 successfuly share f_i with the servers. More specifically, in applications involving e-commerce [33], web queries [50], and media consumption [32], 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 [27, 38, 49] and private telemetry [8, 18], access control is crucial to prevent attacks by malicious users sending invalid writes (e.g., by overwriting mailboxes of honest users [27, 38, 49]). 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 [8, 11, 23]). Preventing malicious evaluators from violating privacy, without relying on strong assumptions or inefficient solutions, can be difficult [7, 11, 23].

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 [7]). 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 [9] 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 party can evaluate their secret share of f on a public input x to obtain a secret share of $f(x)$.
- **Privacy.** No evaluator gains any information on f given a secret share of $[f]$.
- **Efficiency.** FSS requires the size of the secret shares to be small (sublinear in the size of the truth table for f).

Boyle et al. [9, 11] 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 [9]. The main primitive behind their results is an FSS family for distributed point functions (DPFs) [9, 11, 30]. Subsequent work of Boyle et al. [11] 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 [9, 10, 25].

1.2 Prior work

Recent work on anonymous communication provides ad-hoc solutions to access control in the context of FSS. Express [27] (USENIX 2021), Spectrum [38] (NSDI 2022), and Sabre [49] (S&P 2022) 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 FSS, 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 FSS and currently remains specific to the two-party FSS construction for point functions [9, 11], (2) the use of "addresses" for access control requires a large output range and leads to a 5× computational overhead for the evaluators (and, more importantly, prevents optimization techniques for FSS [11]), 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 do not make assumptions on the underlying FSS scheme, making them compatible with FSS optimizations [11, 23].

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 FSS constructions [9, 11] and requires a round of interaction between the evaluators to enforce access control. The access control mechanism of Sabre is

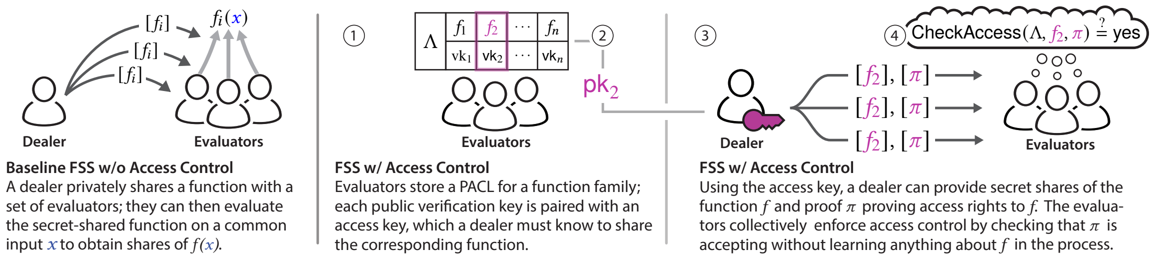

Fig. 1: Overview of FSS and our access control model. **Left:** In FSS *without* access control, a dealer distributes shares of a function f_i with the evaluators. ① A setting *with* access control where the evaluators have Λ and ② the dealer has an access key pk_2 for f_2. ③ the dealer (acting as the 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.

actually a special case of a generic approach to access control realized via zero-knowledge proofs over secret-shares, which we describe in Section 8. 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 9, 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:

Contribution I. 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 [27, 38, 49] and demand a stringent set of efficiency requirements that align closely with the goals of FSS.

Contribution II. 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.

Contribution III. For the special case of *verifiable* FSS [23], 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.

Contribution IV. 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. We also construct PACLs tailored to obfuscation-based FSS, using the recent result of Canetti et al. [15] (Eurocrypt'22). To the best of our knowledge, this is the first construction of *verifiable* obfuscation-based FSS and may be of independent interest. Unlike our generic construction of PACLs constructed from zero-knowledge proofs, our construction for obfuscation-based FSS does not require a random oracle.

Contribution V. 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 objects (in our case, functions) each associated with access and verification keys.

Definition 1 (Access Control Lists).
Let $\lambda \in \mathbb{N}$ be a security parameter, $\mathcal{F}\colon \{0,1\}^n \to \{0,1\}^*$ 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 1 is general and not specific to FSS (indeed, Definition 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 1 with *completeness* and *soundness* properties (with respect to an adversary). These are likewise deferred to the formalization of private ACLs in Definitions 3 and 4.

We now describe instantiations of CheckAccess from Definition 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 $(\mathsf{vk}_1, \dots, \mathsf{vk}_N)$ and access $(\mathsf{sk}_1, \dots, \mathsf{sk}_N)$ 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 $\Lambda_\lambda = (\mathsf{vk}_1, \dots, \mathsf{vk}_N)$
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 1 is the *inclusion* predicate that associates each function with $\ell \geq 1$ verification keys,

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

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

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

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

That is, $\mathsf{CheckAccess}$ 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 1 is a monotone boolean predicate (ANDs and ORs) over a list of $\mathsf{CheckAccess}$ 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 [45] consists of two (possibly randomized) algorithms $\mathsf{Share}_{(\mathbb{F},t,s)}$ and $\mathsf{Recover}$. 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 $\mathsf{Recover}$ to reveal the encoded value in the field $\mathbb{F}$, (2) no subset of fewer than $t-1$ 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 $\mathsf{Recover}$ 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 $\mathsf{ExpRecover}\colon \mathbb{G}^t \to \mathbb{G}$ to be the efficiently computable algorithm 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 $y \in \mathbb{G}$ [35].

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 (FSS [9]). *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:*

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

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 $D_I \approx_c \mathcal{S}(1^\lambda, I)$. That is, the distribution of any subset of $(t-1)$ FSS keys reveals no information on the function f to the subset of computationally bounded evaluators.
- **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. [9], we assume Decode is a linear function of the inputs and therefore let $\mathsf{Decode} := \mathsf{Recover}$. 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). 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 1) for the function family $\mathcal{F}$. The verifiers determine whether or not $\mathsf{CheckAccess}$ 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 Public-key PACL

A public-key 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 $\mathsf{CheckAccess}$ for the family of functions in the distributed setting. Audit and Verify only reveal the output of $\mathsf{CheckAccess}$ (yes or no). without revealing any other information to the verifiers. We leave public parameters as an implicit input to all algorithms.

Definition 3 (PACL: Syntax, Completeness, & Efficiency). *Let $\lambda \in \mathbb{N}$ be a security parameter, integer $N := 2^n$, and $\mathcal{F} : \left\{f_i : \{0,1\}^n \rightarrow \{0,1\}^* \mid 1 \leq i \leq N\right\}$ 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:*

- $\mathsf{KeyGen}(1^\lambda, f) \rightarrow (\mathsf{vk}, \mathsf{sk})$. *Takes as input a security parameter 1^λ and a function $f \in \mathcal{F}$. Outputs a new pair of verification and access keys $(\mathsf{vk}, \mathsf{sk})$.*
- $\mathsf{Prove}(f, \mathsf{sk}) \rightarrow ([\pi]_1, \ldots, [\pi]_s)$. *Takes as input a function $f \in \mathcal{F}$ and access key sk. Outputs a vector of s proof secret shares $([\pi]_1, \ldots, [\pi]_s)$.*
- $\mathsf{Audit}(\Lambda, [f]_i, [\pi]_i) \rightarrow \tau_i$. *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.*
- $\mathsf{Verify}(\mathcal{T} := \{\tau_i \mid i \in I\}) \rightarrow \mathsf{yes}/\mathsf{no}$. *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 $\mathsf{CheckAccess}$ be as defined in Definition 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 subsets $I \subseteq \{1, \ldots, s\}$ with $|I| \geq t$, for all security parameters λ, 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. Verify takes any subset of t audit tokens output by Audit.
- **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 (PACL), matches the ϵ of Definition 2 (FSS). This translates to a constant overhead in communication over sharing f itself via FSS (i.e., *without* any access control).

Definition 4 (PACL, Soundness & Privacy). *A PACL scheme (as defined in Definition 3) 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 $i \in \{1, \ldots, N\}$:	$T := T \cup \{j\}$
$\quad(\mathsf{vk}_i, \mathsf{sk}_i) \leftarrow \mathsf{KeyGen}(1^\lambda, f_i)$	**return** sk_j
$\Lambda := (\mathsf{vk}_1, \ldots, \mathsf{vk}_N),\ T = \{\}$	
$([f_\gamma], [\pi]) \leftarrow \mathcal{A}^{\textsc{GetKey}}(1^\lambda, \Lambda)$	
$f_\gamma \leftarrow \mathsf{Recover}([f_\gamma])$	
for $i \in I$:	
$\quad\tau_i \leftarrow \mathsf{Audit}(\Lambda, [f_\gamma]_i, [\pi_\gamma]_i)$	
return $\mathsf{Verify}(\{\tau_i \mid i \in I\}) = \mathsf{yes}$	
$\quad$ **and** $f_\gamma \in \mathcal{F}$ **and** $\gamma \notin T$	

Fig. 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 [27, 49], it is useful to relax the definition of soundness of Definition 4 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) [29].

Definition 5 (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 3.*

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

Fig. 3: 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 4, here Λ is private to the verifiers and is not given to $\mathcal{A}$.

3.3 Security against malicious verifiers

Privacy. Definition 3 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 4—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 servers adhere to the protocol (any malicious server 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 [27] and Spectrum [38]).

4 Group-based constructions

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 [9, 11]. 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 [30]. We focus on $m = 1$ for simplicity; 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 [9, 11], 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. [38]. In Section 4.1.2, we extend this technique to 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 $\mathsf{CheckAccess}$ 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 $\Lambda = (g^{\alpha_1}, \dots, g^{\alpha_N})$ and (2) the FSS key κ encoding a DPF can be used to privately retrieve the ith entry in any vector by first evaluating the DPF $[y_j] \leftarrow \mathsf{DPF.Eval}(\kappa, j)$ and then computing the inner-product "in the exponent" as: $g^{[\alpha_i]} := g^{\langle(\alpha_1,\dots,\alpha_N),([y_1],\dots,[y_N])\rangle}$. This allows the verifiers to *locally* obtain a (multiplicative) secret share $g^{[\alpha_i]}$. To verify knowledge of α_i, the prover distributes to the verifiers *additive* secret shares of $\alpha := \alpha_i$ (described in Prove). Each verifier computes $\tau_i := (g^{[\alpha_i]})g^{[\pi]}$ 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* $O(\lambda)$ *and where* $\mathsf{CheckAccess}$ *is instantiated as the match predicate of Section 2.1.*

Algorithm 1: DPF-PACL FOR MATCH PREDICATES

Public parameters: integers $2 \le t \le s$, function family $\mathcal{F} = \{f_i : \{0,1\}^n \to \mathbb{Z}_p \mid 1 \le i \le N\}$, 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: **parse** $\mathsf{sk} = \alpha$
2: $([\pi]_1, \dots, [\pi]_s) \leftarrow \mathsf{Share}_{(\mathbb{Z}_p,t,s)}(\alpha)$
3: **return** $([\pi]_1, \dots, [\pi]_s)$

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

– $\mathsf{Verify}(\tau_1, \dots, \tau_t)$:
1: $C \leftarrow \mathsf{ExpRecover}(\tau_1, \dots, \tau_t)$ `// See Remark 1 for definition.`
2: **return** $C \stackrel{?}{=} 1_{\mathbb{G}}$

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.

– $\mathsf{Precomputation}$: `// Compute correction terms`

```
parse Λ = (vk_1, ..., vk_N), vk_j = (vk_{j,1}, ..., vk_{j,ℓ})
for j ∈ {1, ..., N}, k ∈ {1, ..., ℓ}:
    vk_{j,k} = (g^{α_{j,k}}, g^{β_{j,k}})
    g^{w_{(j-1)ℓ+k}} := ∏_{l=1, l≠k}^{ℓ} g^{α_{j,l}}
```

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

– $\mathsf{KeyGen}(1^\lambda, f_i)$:

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

– $\mathsf{Prove}(f, \mathsf{sk})$:

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

– $\mathsf{Audit}(\Lambda, [f], [\pi])$:

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

– $\mathsf{Verify}(\tau_1, \ldots, \tau_t)$:

- **parse** $\tau_i = (\tau_i^{(0)}, \tau_i^{(1)})$
- **if** $\mathsf{Verify}'(\tau_1^{(0)}, \ldots, \tau_t^{(0)}) = \mathsf{no}$ **then return** no
- $C \leftarrow \mathsf{ExpRecover}(\tau_1^{(1)}, \ldots, \tau_t^{(1)})$ `// See Remark 1.`
- **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 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 4. One option is to use zero-knowledge proofs over secret shares [7, 18]. 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 (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 multiplicatively 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,

$$\left(\prod_{k=1}^{\ell} g^{[w_{i,k}]}\right) = \left(\prod_{k=1}^{\ell} g^{[\alpha_{i,k}]}\right)^{(\ell-1)},$$

which means that:

$$\left(\prod_{k=1}^{\ell} g^{[\alpha_{i,k}]}\right) \Big/ \left(\prod_{k=1}^{\ell} g^{[w_{i,k}]}\right)^{(\ell-1)} = g^0 = 1_{\mathbb{G}}.$$

Hence, the verifiers would then recover shares of $1_{\mathbb{G}}$ 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). However, we observe that we can use the "FSS tensoring" transformation described by Boyle et al. [11] 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. [11] 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 parties (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 [6] (e.g., when $\mathbb{G} = \mathbb{Z}_p^*$) then the construction satisfies the soundness property of PACLs as defined in Definition 4. When $\mathbb{G}$ is the *additive* group $\mathbb{Z}_p$, then we get a symmetric-key PACL satisfying the relaxed soundness property in Section 3.2.

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 $\alpha' := \sum_{i=1}^{q} \alpha_i$ 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 $\mathsf{CheckAccess}(\Lambda_i, f_i, \alpha_i) = \mathsf{yes}$ 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 and 4. We first prove a useful lemma which says that any adversary that wins the $\textsc{PkSoundness}_{\mathrm{PACL},\mathcal{A},I}(\lambda)$ 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 $\textsc{PkSoundness}_{\mathrm{PACL},\mathcal{A},I}(\lambda)$ 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 $\mathcal{F}_{\mathsf{DPF}}$, then, there exists an efficient $\mathcal{A}'$ that wins the $\textsc{PkSoundness}_{\mathrm{PACL},\mathcal{A},I}(\lambda)$ game with probability $\delta(\lambda)$ where $\mathcal{A}'$ outputs f_γ and π such that $f_\gamma \in \mathcal{F}_{\mathsf{DPF}}$.*

Proof. Deferred to Appendix B.1. ☐

Theorem 3. *Let p be a prime chosen in 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 and 4 with $\mathsf{CheckAccess}$ *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 $|I| \geq t$:

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

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 $\mathcal{A}^{\textsc{GetKey}}(1^\lambda, \Lambda)$. Respond to each $\textsc{GetKey}(j)$ 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 $\alpha_\gamma = -\pi$. 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^λ, index subset I, and $\{\tau_i^* \mid i \in I\}$, $\mathcal{S}$ proceeds as follows:

1: $J := \{1, \ldots, s\} \setminus I$.
2: $([0]_1, \ldots, [0]_s) \leftarrow_R \mathbb{Z}_p^s$.
3: $([\pi]_1, \ldots, [\pi]_s) \leftarrow_R \mathbb{Z}_p^s$.
4: $\tau_j := g^{[0]_j}$ for all $j \in J$.
5: 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, 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 $g^0 = 1_{\mathbb{G}}$. (Note that the audit tokens 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 $\mathcal{S}$ can thus only differ 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 $O(\lambda)$. $\square$

Theorem 4. *Let p be a prime chosen in 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 and 4 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.2. $\square$

5 Faster PACLs for DPFs from Verifiable DPFs

In this section, we introduce a concretely more efficient construction of DPF-PACL for the class of *verifiable* DPFs (VDPFs) [23] (also known as *extractable* DPFs [8]). A VDPF allows the evaluators to efficiently check if the DPF is well-formed (see Appendix A), 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., $\mathbb{Z}_p$) rather than in $\mathbb{G}$, then computing the audit token would be bottlenecked by operations over the field instead of (possibly expensive) group 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 an additive sharing of $[g^{\alpha_i}]$ (rather than a multiplicative sharing $g^{[\alpha_i]}$) 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.

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. 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 [7, 18] and can be thought of as a secret-shared analog of a Schnorr proof [43]. We define the formal requirements of SPoSS in Definition 6 and prove security of our construction in Appendix B.3. The proof size of our SPoSS construction is significantly smaller compared to generic approaches based on zero-knowledge proofs (see Section 9).

Definition 6 (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* (Prove, Audit, Verify) *with the following functionality. We leave $\mathbb{G}$ and g as implicit inputs.*

- $\mathsf{Prove}(x) \to [\pi]$. *Takes as input integer $x \in \mathbb{Z}_p$. Outputs proof shares $([\pi]_1, \ldots, [\pi]_s)$.*
- $\mathsf{Audit}([y], [\pi]) \to \tau$. *Takes as input a secret share $[y]$ and a secret share $[\pi]$. Outputs an audit token τ.*
- $\mathsf{Verify}(\mathcal{T} := \{\tau_i \mid i \in I\}) \to \mathsf{yes}/\mathsf{no}$. *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.*

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 $([\pi^*]_1, \ldots, [\pi^*_s])$ such that $\mathsf{Verify}(\tau_1, \ldots, \tau_t) = \mathsf{yes}$ (where each $\tau_i \leftarrow \mathsf{Audit}([y]_i, [\pi^*]_i)$) with probability $\delta(\lambda)$, then there exists an efficient knowledge extractor $\mathcal{E}$ and negligible function negl such that,

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

Algorithm 3: SCHNORR PROOF OVER SECRET SHARES

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

– $\mathsf{Prove}(x)$:

$([x]_A, [x]_B) \leftarrow \mathsf{Share}_{(\mathbb{Z}_p,2,2)}(x)$
$y_A := g^{[x]_A}$, $y_B := g^{[x]_B}$
$([a]_A, [b]_A, [c]_A, [a]_B, [b]_B, [c]_B) \leftarrow \mathsf{Beaver}_{(2,2)}(\mathbb{Z}_p)$
$a \leftarrow [a]_A + [a]_B$, $b \leftarrow [b]_A + [b]_B$ `// See optimization in Section 5.1.1`
$z_A, z_B \leftarrow_R \{0,1\}^\lambda$ `// random nonces`
$r_A \leftarrow H(z_A, [x]_A, a, [c]_A)$, $r_B \leftarrow H(z_B, [x]_B, b, [c]_B)$
$r \leftarrow r_A \oplus r_B$
$d \leftarrow rg^{[x]_A} - a$, $e \leftarrow g^{[x]_B} - b$
$\pi_A := (\text{A}, [x]_A, a, [c]_A, r, d, e, z_A)$
$\pi_B := (\text{B}, [x]_B, b, [c]_B, r, d, e, z_B)$
return (π_A, π_B)

– $\mathsf{Audit}([y], \pi)$:

parse $\pi := (\text{T}, [x], u, [c], r, d, e, z)$
$\hat{r} \leftarrow H(z, [x], u, [c])$
$\hat{y} \leftarrow g^{[x]}$
if $\text{T} = \text{A}$:
$f \leftarrow r\hat{y} - u$
$[v] \leftarrow (de/2) + eu$
$[w] \leftarrow [v] + [c] - r[y]$
if $\text{T} = \text{B}$:
$f \leftarrow \hat{y} - u$
$[v] \leftarrow (de/2) + du$
$[w] \leftarrow [v] + [c] - [y]$
$\tau := ([w], \hat{r}, r, f, d, e)$
return τ

– $\mathsf{Verify}(\{\tau_A, \tau_B\})$:

parse $\tau_A = ([w]_A, \hat{r}_A, r, \hat{d}, d, e)$.
parse $\tau_B = ([w]_B, \hat{r}_B, r, \hat{e}, d, e)$.
$\hat{r} \leftarrow \hat{r}_A \oplus \hat{r}_B$, $w \leftarrow [w]_A + [w]_B$
return $w = 0$ and $\hat{r} = r$ and $\hat{d} = d$ and $\hat{e} = e$

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 has to be 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 $g^{[x]}$ 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 $g^{[x]}$, 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 would require *interaction* between the verifiers. Instead, in Algorithm 3, we use a standard approach from zero-knowledge proofs over secret-shares [7, 18] and have the prover "assist" the verifiers in the computation. Specifically, the prover provides a Beaver multiplication triple [5], enabling the verifiers to compute the multiplication. (We provide an overview of Beaver multiplication in Appendix C for completeness.)

Preventing malicious provers. As observed in prior work [18], this enables 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 $\mathbb{Z}_p$ [18].

Removing interaction. Finally, in Algorithm 3, to avoid interaction between verifiers, we apply the Fiat-Shamir transform [28] 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. [7]. Given a random oracle H, the prover generates a proof using H to simulate the choice of r by the verifiers. As noted in [7], in the distributed setting, the resulting r can leak information about the shares. To prevent this, in Algorithm 3, we follow the blueprint of [7] 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 $[\hat{y}_i]$ (it is always zero). As a consequence, only the verifier holding the non-zero share needs to mask it when computing the Beaver multiplication (see Appendix C). 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 function 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 10), 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 in a security parameter $\lambda \in \mathbb{N}$ and let $\mathbb{G}$ be any 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 and 4 with* CheckAccess *as defined in Section 2.1 (match predicate).*

Algorithm 4: VDPF-PACL FOR MATCH PREDICATES

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

– $\mathsf{KeyGen}(1^\lambda, f_i)$: as in Algorithm 1.

– $\mathsf{Prove}(f, \mathsf{sk})$:

1: **parse** $f = P_i$ and $\mathsf{sk} = \alpha$
2: $([\pi]_1, \ldots, [\pi]_s) \leftarrow \mathsf{SPoSS.Prove}(\alpha)$
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: $I \leftarrow \{1, \ldots, |\mathcal{T}|\}$
2: **parse** $\mathcal{T} = \{(\tau_i, \rho_i) \mid i \in I\}$.
3: **return** $\mathsf{SPoSS.Verify}(\{\tau_i \mid i \in I\})$ and $\mathsf{VDPF.Verify}(\{\rho_i \mid i \in I\})$

Proof. We prove each property in turn.

Completeness. The inner-product computed over the keys results in parties holding secret shares of g^{α_i} (where i is the special index of the encoded point function). It then follows from the completeness of the SPoSS protocol (see Appendix B.3) that $\mathsf{VDPF.Verify}$ must pass following an honest audit procedure. The other half of the Verify conjunction follows from the completeness of VDPFs (see Appendix A).

Soundness. Assume, towards contradiction, that there exists an efficient prover $\mathcal{A}$ and non-negligible function δ such that for all $I \subset \{0, 1\}$:

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

By the soundness property of VDPFs (see Definition 10 in Appendix A), we can assume that f_γ (output by $\mathcal{A}$ in the $\textsc{PkSoundness}_{\text{PACL},\mathcal{A},I}(\lambda)$ game defined in Figure 2) is a point function with special index γ. This restricts $\mathcal{A}$ to outputting a well-formed VDPF, which implies that both verifiers obtain secret shares of g^{α_γ} from Λ when computing the inner product in Audit of Algorithm 4. We construct an efficient extractor $\mathcal{B}$ that solves the discrete-logarithm problem as follows.

1: On input $y := g^x$, generate a planted instance of Λ with y in a random index (as in the proof of Theorem 3).
2: Run $\mathcal{A}$ on inputs $(y, 1^\lambda, \Lambda)$ and obtains as output $([f_i], [\pi])$.
3: Simulate the SPoSS knowledge extractor on input $[\pi]$ (see Appendix B.3) and obtain x.
4: Output x.

If $\mathcal{A}$ succeeds, then $\mathcal{B}$ recovers the discrete logarithm of g^x via the SPoSS knowledge extractor, contradicting the assumption that the discrete logarithm is computationally intractable in $\mathbb{G}$. By contrapositive, soundness of Algorithm 4 follows.

Privacy. As with the proof of the DPF-PACL privacy, we construct an efficient simulator $\mathcal{S}$ for the view of verifier $b \in \{0, 1\}$. We use the efficient simulators $\mathcal{S}_{\mathsf{VDPF}}$ and $\mathcal{S}_{\mathsf{SPoSS}}$ to generate the view of the VDPF (Definition 10 in Appendix A) output and SPoSS proof (Definition 6), respectively. $\mathcal{S}$ proceeds as follows:

1: On inputs 1^λ and b, set $([\pi]_i, \tau_i, \tau_{i-1})$ as the output of $\mathcal{S}_{\mathsf{SPoSS}}(1^\lambda, \{1 \oplus b\})$.

2: Set ρ_i as the output of $\mathcal{S}_{\mathsf{VDPF}}(1^\lambda, \{1 \oplus b\})$.
3: Output (π_i, τ_i, ρ_i).

The distribution output by $\mathcal{S}$ matches the distribution of verifier $b \in \{0,1\}$ because: (1) the proof shares and audit tokens $([\pi]_i, \tau_i)$ are output by the SPoSS simulator $\mathcal{S}_{\mathsf{SPoSS}}$, which guarantees statistical indistinguishability of the view to that of Verifier b and (2) the VDPF simulator $\mathcal{S}_{\mathsf{VDPF}}$ guarantees computational indistinguishability of the VDPF verification token ρ_i. The output of $\mathcal{S}$ can thus only differ on the recovered public key share. However, an efficient distinguisher for a VDPF key contradicts the privacy property of VDPFs (Definition 10 in Appendix A). Privacy of Algorithm 4 follows.

Efficiency. Each proof share $[\pi]_i$ and audit token τ_i consists of a constant number of elements in $\mathbb{Z}_p$ and thus is of size $O(\lambda)$ (see Definition 6 and Appendix B.3). By definition of VDPFs, the size of ρ_i satisfies our efficiency constraint. (We note that under the VDPF formulation presented by de Castro and Polychroniadou [23], the efficiency of the verification procedure is not explicitly defined but can be seen as an implicit requirement.) □

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. [9, 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 [9], for instance, fully-homomorphic encryption [25].

PACLs for range functions and decision trees. Boyle et al. [9, 11] describe how to apply linear combinations of DPFs to derive FSS for range functions (and more generally decision trees [11]). 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 [9, 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 $\mathcal{F}$ [9]. Our DPF-PACL construction applies to this class of FSS directly as a result. Following similar transformations, Boyle et al. [9] 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 naïve approach. As observed by Boyle et al. [9], 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 8) 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 PACLs for $i\mathcal{O}$-based FSS

In this section, we describe how to construct PACLs for FSS based on indistinguishability obfuscation which is known to yield FSS for all functions in $P/poly$. Along the way, we also show how to construct *verifiable* FSS from indistinguishability obfuscation, which may be of independent interest. Compared to Section 8, which describes how to construct PACLs for *any* FSS scheme (including obfuscation-based FSS), our explicit construction for obfuscation-based FSS has two properties which are possibly of theoretical interest. First, unlike our generic construction, we do not make use of a random oracle and therefore weaken the necessary assumptions required. Second, to the best of our knowledge, we are the first to construct *verifiable* obfuscation-based FSS, which requires the use of the recent result of Canetti et al. [15] (Eurocrypt'22) for checking the "well formedness" of obfuscated programs.

7.1 Preliminaries

Indistinguishability obfuscation ($i\mathcal{O}$) [4] satisfies the property that the obfuscation of two "functionally equivalent" circuits C_0 and C_1 are computationally indistinguishable. Verifiable indistinguishability obfuscation ($vi\mathcal{O}$) [3, 15] additionally enables verifying some predicate ϕ over the obfuscated circuit (e.g., checking that the circuit encodes a specific function or "program" without revealing the hardcoded program secrets). We use $\equiv$ to denote "functional equivalence" between two circuits.

Definition 7 ($i\mathcal{O}$ [4]). *An efficient uniform algorithm* Obfs *is said to be an indistinguishability obfuscator for a class of circuits* $\{\mathcal{C}_\lambda\}_{\lambda\in\mathbb{N}}$ *if the following properties hold:*

- **Functionality.** There exists a negligible function negl such that for all $\lambda \in \mathbb{N}$, for all $C \in \mathcal{C}_\lambda$, and for every input x to C:

$$\Pr[\widetilde{C} \leftarrow \mathsf{Obfs}(C); \widetilde{C}(x) \not\equiv C(x)] \leq \mathsf{negl}(|C|),$$

  where the probability is over the randomness of Obfs.
- **Polynomial slowdown.** For all $\lambda \in \mathbb{N}$ and for all $C \in \mathcal{C}_\lambda$, it holds that $|\mathsf{Obfs}(C)| \leq \mathsf{poly}(\lambda, |C|)$. That is, the obfuscated circuit is at most polynomially larger than the circuit being obfuscated.
- **Indistinguishability.** For all efficient distinguishers $\mathcal{D}$, there exists a negligible function negl such that for all $\lambda \in \mathbb{N}$, for all pairs of circuits $C_0, C_1 \in \mathcal{C}_\lambda$ where $C_0(x) \equiv C_1(x)$,

$$\left|\Pr\left[\widetilde{C}_0 \leftarrow \mathsf{Obfs}(C_0) : \mathcal{D}(\widetilde{C}_0)\right] - \Pr\left[\widetilde{C}_1 \leftarrow \mathsf{Obfs}(C_1) : \mathcal{D}(\widetilde{C}_1)\right]\right| \leq \mathsf{negl}(\lambda),$$

  where the probability is over the randomness of Obfs and $\mathcal{D}$.

Probabilistic $i\mathcal{O}$ ($pi\mathcal{O}$) [14] enables obfuscation of probabilistic (randomized) circuits by compiling them into computationally indistinguishable deterministic circuits (subject to the program being run on only one input).

Verifiable $i\mathcal{O}$ ($vi\mathcal{O}$) provides a method for verifying the output correctness of an obfuscated probabilistic circuit [3, 15]. More formally, $vi\mathcal{O}$ consists of two algorithms $(\mathsf{Obf}, \mathsf{Verify})$ and a predicate ϕ, and guarantees that if $vi\mathcal{O}.\mathsf{Verify}$ accepts a string Π (purportedly generated by $vi\mathcal{O}.\mathsf{Obfs}$) and returns a valid $\widetilde{C} \neq \perp$, then there exists a ϕ-satisfying circuit that, when obfuscated, is functionally equivalent to $\widetilde{C}$.

Definition 8 (Verifiable Indistinguishability Obfuscation). *Let* $(\mathsf{Obf}, \mathsf{Verify})$ *be efficiently computable uniform algorithms where* Obfs *is an indistinguishability obfuscator for a circuit class* $\{\mathcal{C}_\lambda\}_{\lambda\in\mathbb{N}}$ *(as per Definition 7). Let* ϕ *be an efficiently computable predicate over circuits.* Verify *has the following syntax.*

- $\mathsf{Verify}(1^\lambda, \Pi_\lambda, \phi) \to \widetilde{C}$ *or* $\perp$. *Takes as input a security parameter, a string* Π_λ *purportedly generated by* $\mathsf{Obfs}(1^\lambda, C, \phi)$*, and predicate* ϕ*. Outputs* $\widetilde{C}$ *if* $\phi(C) = 1$ *and* $\perp$ *otherwise.*

The algorithms $(\mathsf{Obf}, \mathsf{Verify})$ *must satisfy:*

- **Correctness.** For all $\lambda \in \mathbb{N}$, for all circuit $C \in \mathcal{C}_\lambda$ such that $\phi(C) = 1$, and for all inputs x to C:

$$\Pr\left[\widetilde{C} \leftarrow \mathsf{Verify}(1^\lambda, \mathsf{Obf}(1^\lambda, C, \phi), \phi) : \widetilde{C}(x) \equiv C(x)\right] = 1.$$

- **Verifiability.** For every ensemble of polynomial-length strings $\{\Pi_\lambda\}_{\lambda\in\mathbb{N}}$, there exists a negligible function negl such that

$$\Pr\left[\begin{array}{l}\widetilde{C} \leftarrow \mathsf{Verify}(1^\lambda, \Pi_\lambda, \phi) : \\ \widetilde{C} \neq \bot \wedge \Big(\nexists (C \in \mathcal{C}_\lambda, r) : \phi(C) = 1 \wedge \widetilde{C} \equiv \mathsf{Obfs}(C; r)\Big)\end{array}\right] \leq \mathsf{negl}(\lambda),$$

where the probability is over Obfs.

- **Indistinguishability.** For all efficient distinguishers $\mathcal{D}$, there exists a negligible function negl such that for all $\lambda \in \mathbb{N}$, for all pairs of circuits $C_0, C_1 \in \mathcal{C}_\lambda$ where $C_0 \equiv C_1$ and $\phi(C_0) = \phi(C_1) = 1$,

$$\left|\Pr\Big[\widetilde{C}_0 \leftarrow \mathsf{Obfs}(C_0); \mathcal{D}(\widetilde{C}_0)\Big] - \Pr\Big[\widetilde{C}_1 \leftarrow \mathsf{Obfs}(C_1); \mathcal{D}(\widetilde{C}_1)\Big]\right| \leq \mathsf{negl}(\lambda),$$

where the probability is over Obfs and $\mathcal{D}$.

Theorem 6 (Verifiable Indistinguishability Obfuscation [3, 15]). *Assume the existence of sub-exponentially secure indistinguishability obfuscation and sub-exponentially secure one-way functions. Then there exists a vi$\mathcal{O}$ scheme for all polynomial-sized circuits (the class P/poly), satisfying Definition 8.*

Verifiable encryption. We require any IND-CCA [6] (Indistinguishability under Chosen Ciphertext Attacks) secure public key encryption scheme $\mathcal{E} := (\mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec})$ where $\mathcal{E}.\mathsf{Dec}$ outputs $\bot$ if provided an invalid ciphertext (a ciphertext that was not encrypted under a valid public key). We denote by $\mathcal{E}.\mathsf{Enc}(\mathsf{pk}, m; r)$ an encryption of a message m using randomness r.

Vector commitment scheme. A vector commitment scheme VC consists of efficient algorithms $(\mathsf{Commit}, \mathsf{Verify})$ and allows for committing to a vector of elements and later opening the commitment to a particular element index such that Verify accepts if the opening is valid. Catalano and Fiore [16] formalize the notion of vector commitments, which include standard constructions such as Merkle trees [36].

7.2 $i\mathcal{O}$-based FSS

Boyle et al. [9] provide a construction of *P/poly*-FSS using probabilistic indistinguishability obfuscation ($pi\mathcal{O}$), which is implied by the sub-exponential hardness of indistinguishability obfuscation ($i\mathcal{O}$) and one-way functions [14].

FSS from $pi\mathcal{O}$. The high-level idea is the following. The dealer generates a randomized program that has *hardcoded* in it the function f (described as a circuit) and an encryption public key for each party. The output of the program consists of *encryptions* of the secret shares of $f(x)$ such that each party can decrypt its own share. On input x, the $pi\mathcal{O}$ program first evaluates $f(x)$, then generates shares of $f(x)$ using the randomness hardcoded into the program, and outputs encryptions of the shares using the public keys of the parties. We focus on the two-party setting for simplicity as the scheme trivially generalizes to more parties. Boyle et al. [9] prove the following theorem.

Theorem 7 ([9]). *Assume the existence of sub-exponentially secure indistinguishability obfuscation and sub-exponentially secure one-way functions. Then there exists an FSS scheme for functions in P/poly.*

A failed approach for constructing $pi\mathcal{O}$-FSS PACLs. One (flawed) idea is to let the dealer hardcode the access sk into the obfuscated program and run $\mathsf{CheckAccess}$ inside the program, outputting $\bot$ if the check fails. Unfortunately, this approach is neither sound nor private.

The first problem is that a malicious prover can generate a bad program $\mathcal{P}^*$ that does not compute $\mathsf{CheckAccess}$ (or computes $\mathsf{CheckAccess}(\cdot,\cdot,\cdot) = 1$ on all inputs).

Second, even if each party could somehow check the structure of $\mathcal{P}$ to ensure that $\mathsf{CheckAccess}$ is evaluated correctly, nothing stops the prover from encrypting the outputs under the wrong public keys (the public keys are hardcoded into the program and thus cannot be checked by the party). A malicious prover could substitute pk^* for a party's true public key pk, which would result in an invalid decryption of the party's secret share (in essence allowing the malicious prover to share a random function, bypassing access control).

The third problem is that a malicious party can *exploit* the access control step to learn $f_i \in \mathcal{F}$ as follows. Suppose that Λ consists of verification keys $(\mathsf{vk}_1, \ldots, \mathsf{vk}_N)$ and the program contains the hardcoded access key $\mathsf{sk} = \mathsf{sk}_i$. A party can evaluate $\mathcal{P}$ with $\Lambda' := (\mathsf{vk}_1, \ldots, \mathsf{vk}_j^* \ldots, \mathsf{vk}_N)$ by replacing vk_j with a bogus verification key vk_j^*. The access control will succeed as long as $i \neq j$, which allows the party to guess-and-check each $j \in \{1, \ldots, N\}$ and recover f_i in polynomial time.

Fixing the problems. First, to prevent a prover from sharing a malicious program $\mathcal{P}^*$, our PACL construction will have the parties check the code of the provided program using *verifiable pi$\mathcal{O}$* (Definition 8) to ensure it is evaluating the correct functionality (without needing to know the hardcoded secret inputs and randomness).

Second, the parties have to ensure that they all have the same instance of the program, which we will do with the help of a collision-resistant hash function (CRHF).

Third, to prevent invalid encryptions, each party has to check to make sure they obtain an *authenticated* encryption of a function share (decryption will fail if the program does not output a valid encryption under their public key).

These three checks force the dealer to provide a valid *pi$\mathcal{O}$* program that outputs valid secret shares of the function and makes the resulting PACL meet the soundness requirement of Definition 4.

Finally, to prevent malicious verifiers from orchestrating guess-and-check attacks, the verifiers *commit* to the verification keys using any vector commitment scheme [16]. However, we observe that we do not necessarily require efficient openings (as typically required in vector commitment schemes) since the "opening" of the commitment can simply be the full vector of keys itself. As such, the verifiers can commit to the access control list by simply hashing the entire list using a CRHF. The program then internally checks the commitment opening, outputting $\perp$ if the commitment opening is invalid.

We formalize the above ideas in Section 7.3.

7.3 Construction

Theorem 8 (*pi$\mathcal{O}$*-PACL). *Assume the existence of sub-exponentially secure indistinguishability obfuscation and sub-exponentially secure collision-resistant hash functions (CRHFs). Then there exists a PACL for pi$\mathcal{O}$-FSS.*

Program 1 presents the construction of Boyle et al. [9] for *pi$\mathcal{O}$*-FSS. We highlight the changes that are necessary to enforce access control via our PACL for *pi$\mathcal{O}$*-FSS presented in Algorithm 5.

Program 1: $\mathrm{FSS}_{f,\mathsf{pk}_A,\mathsf{pk}_B,\mathsf{sk},\mathsf{c}}$ WITH ACCESS CONTROL

Hardcoded: $f \in \mathcal{F}$, public keys $\mathsf{pk}_A, \mathsf{pk}_B$, access key sk, and commitment c.
Input: $x \in \{0,1\}^n$, access control list Λ.
Randomness: R, r_A, r_B.

Encrypt R under pk_A, as $\hat{y}_A \leftarrow \mathcal{E}.\mathsf{Enc}(\mathsf{pk}_A, R; r_A)$.
Encrypt $f(x) - R$ under pk_B, as $\hat{y}_B \leftarrow \mathcal{E}.\mathsf{Enc}(\mathsf{pk}_B, f(x) - R; r_B)$.
if $\mathsf{Comm.Verify}(\Lambda, \mathsf{c}) = \mathsf{no}$ **then** output $\perp$.
if $\mathsf{CheckAccess}(\Lambda, f, \mathsf{sk}) = \mathsf{no}$ **then** output $\perp$.
else output $(\hat{y}_A, \hat{y}_B)$.

***pi$\mathcal{O}$*-PACL.** The dealer (prover) distributes secret shares of f by sending an instance of Program 1 to both parties. Each party inputs x and access control list Λ and runs Program 1. Our *pi$\mathcal{O}$*-PACL scheme is defined as follows. Let H be a CRHF sampled by the parties and made publicly available along with encryption public keys of each party and a commitment c to Λ generated according to $\mathsf{Comm.Commit}$. We let ϕ be the predicate that checks if C is an instance of Program 1 under some randomness R, r_A and r_B (in the general case, with more than two parties, ϕ must also check that the randomness used to generate additive secret shares cancels out when added).

Algorithm 5: PACLs for $pi\mathcal{O}$-based FSS

Public parameters: function family $\mathcal{F} = \{f_i : \{0,1\}^n \to \mathbb{Z}_p \mid 1 \le i \le N\}$, integers $2 \le t \le s$, and collision-resistant hash function H sampled by the evaluators.

– $\mathsf{KeyGen}(1^\lambda, f_i)$:

1: Sample $\mathsf{sk}_i \leftarrow_R \{0,1\}^\lambda$.
2: Compute $\mathsf{vk}_i := H(\mathsf{sk}_i)$.
3: Output $(\mathsf{sk}_i, \mathsf{vk}_i)$.

– $\mathsf{Prove}(f_i, \mathsf{sk})$:

1: Compute an obfuscated instance of Program 1 with the highlighted changes, denoted by the string $\mathit{\Pi}$, using $vi\mathcal{O}.\mathsf{Obfs}$. Program 1 is given hardcoded inputs f_i, access key sk, commitment c to the access control list $\mathit{\Lambda}$, and the public keys of both parties.
2: Output $\mathit{\Pi}$.

– $\mathsf{Audit}(\mathit{\Lambda}, [f], [\pi])$:

1: **parse** $[f] = \mathit{\Pi}$ and $[\pi] = b$.
2: If $vi\mathcal{O}.\mathsf{Verify}(1^\lambda, \mathit{\Pi}, \phi) = \perp$ (i.e., $\mathit{\Pi}$ does not encode an instance of Program 1), then output $\tau := \perp$.
3: $\widetilde{C} \leftarrow vi\mathcal{O}.\mathsf{Verify}(1^\lambda, \mathit{\Pi}, \phi)$.
4: If $\widetilde{C}(x, \mathit{\Lambda}) = \perp$, then output $\tau := \perp$.
5: If $\mathcal{E}.\mathsf{Dec}(\mathsf{sk}_b, \hat{y_b}) = \perp$, then output $\tau := \perp$.
6: If no check fails up to this point, then return $\tau := H(\mathit{\Pi})$.

– $\mathsf{Verify}(\mathcal{T})$:

1: **parse** $\mathcal{T} = \{\tau_1, \tau_2\}$.
2: Output yes if and only if the following two conditions hold:

- $\tau_1 = \tau_2$. `// All verifiers received the same program.`
- $\tau_1 \neq \perp \wedge \tau_2 \neq \perp$ `// No check failed for either verifier.`

Proposition 1. *Algorithm 5 instantiated with Program 1 satisfies the completeness, soundness, privacy, and efficiency guarantees of Definition 4.*

Proof (sketch). We give a proof sketch for each property in turn.

Completeness. Completeness follows by inspection of Algorithm 5 and Program 1.

Soundness. Suppose, towards contradiction, that a prover could bypass the access control of Algorithm 5. Then, the prover is either able to (1) generate a program $\mathit{\Pi}$ that does not compute $\mathsf{CheckAccess}$ correctly but still satisfies the predicate ϕ of $pi\mathcal{O}$, (2) send different program encodings $\mathit{\Pi} \neq \mathit{\Pi}'$ to the parties such that $H(\mathit{\Pi}) = H(\mathit{\Pi}')$, or (3) make the function secret shares output by Program 1 be invalid.

A prover bypassing access control via (1) contradicts the verifiability property of $vi\mathcal{O}$ in Definition 8. A prover bypassing access control via (2) contradicts the collision-resistance of H because it is able to generate two different program strings that hash to the same output. A prover bypassing access control via (3) contradicts the verifiable encryption property of IND-CCA-secure encryption by generating a ciphertext that correctly decrypts under secret key sk_b (for $b \in \{1,2\}$) but encrypted under some $\mathsf{pk}^* \neq \mathsf{pk}_b$ (note that by the verifiability property of $pi\mathcal{O}$, the output y_b is guaranteed to be a valid encryption under *some* public key).

In sum, soundness holds because the hardcoded inputs and randomness R, r_A, and r_B are the same for all parties, $\mathsf{CheckAccess}$ is evaluated correctly in the encoded program, and the correct encryption public keys are used to generate each output.

Privacy. Suppose, towards contradiction, that a Algorithm 5 instantiated with Program 1 does not satisfy the privacy property of Definition 4. Then, either Program 1 or the audit procedure reveals information about the hardcoded function f. By the privacy property of $vi\mathcal{O}$, the encoding of Program 1 reveals no information on f to computationally bounded verifiers; but it does not eliminate the possibility of the program itself leaking information when evaluated. Examining the highlighted changes, the only possibility of Program 1 leaking information when evaluated is when the output is $\perp$ (otherwise, the $pi\mathcal{O}$-based FSS scheme of Boyle et al. [9] is not private). Note that the only verifier-provided inputs are x and Λ, of which only Λ has the potential to make Program 1 output $\perp$. However, by the binding property of the commitment scheme, it is computationally infeasible to input $\Lambda' \neq \Lambda$ such that $\mathsf{Comm.Verify}(\Lambda', \mathsf{c}) = \mathsf{yes}$. As such, no information is revealed from the $pi\mathcal{O}$ encoding or the evaluation of the program.

Now, examining Algorithm 5, note that the audit token exchanged between verifiers is simply the hash of the provided program. Because the same program encoding is given to all verifiers (when the prover is honest), no new information is gained from the audit procedure.

This proves that if Algorithm 5 instantiated with Program 1 is not private, then either (1) $vi\mathcal{O}$ is not private or (2) the commitment scheme is not binding; a contradiction.

Efficiency. The efficiency of Program 1 is guaranteed by Definitions 7 and 8 as it is independent of access control list Λ. The size of the audit token τ is $O(\lambda)$.

□

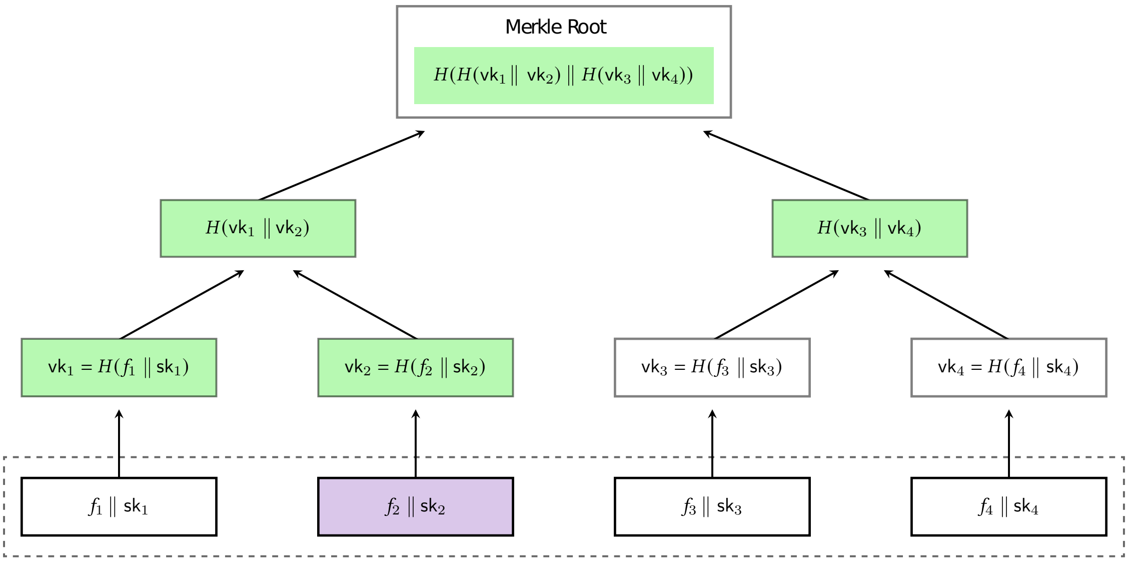

Fig. 4: Example of the Merkle tree description when proving $\mathsf{CheckAccess}(\Lambda, f_2, \mathsf{sk}_2)$. The SNIP consists of secret-shared inputs to $\mathsf{Merkle.Verify}$ which takes as input the path from the root to the leaf (nodes highlighted in green) where the leaf node is the access key for the function (highlighted in purple). Nodes at the leaf layer consist of access keys and are not known to the verifiers. Verification keys are of the form $\mathsf{vk}_i := H(f_i \| \mathsf{sk}_i)$ such that Λ consists of all nodes in the second-to-last layer.

8 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 $P/poly$ [9]). Our approach relies on secret-shared non-interactive proofs (SNIPs) [18] and Fiat-Shamir over SNIPs [7]. We describe these preliminaries in Section 8.1.

8.1 Preliminaries

SNIPs [18] (and their generalizations [7]) 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. 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. We provide a formal definition of SNIPs in Definition 9. We frame the definition to follow the syntax of Definition 3 by abstracting the verifier interaction using algorithms Audit and Verify.

Definition 9 (Secret-shared Non-interactive Proof (SNIP) [18]). *Let $\lambda \in \mathbb{N}$ be a computational security parameter, $\mathbb{F}$ be a finite field, and t and s be integers such that $2 \leq t \leq s$. Let C be any arithmetic circuit (defined over $\mathbb{F}$) where for an $x \in \mathbb{F}$, it holds that $C(x) = 1$. A SNIP is a zero-knowledge proof system instantiated between a prover and s verifiers holding t-out-of-s secret shares of x, where the prover convinces the verifiers that $C(x) = 1$, in zero knowledge. In the random oracle model, a SNIP proof system consists of three (possibly randomized) algorithms* $(\mathsf{Prove}, \mathsf{Audit}, \mathsf{Verify})$*:*

- $\mathsf{Prove}(x, C) \rightarrow ([\pi]_1, ..., [\pi]_s)$. *Takes as input the input x and the arithmetic circuit C. Outputs a vector of t-out-of-s proof secret shares.*
- $\mathsf{Audit}([x]_i, [\pi]_i) \rightarrow (\tau_i)$. *Takes as input a t-out-of-s secret share of x and the corresponding proof share. Outputs a verification string τ_i.*
- $\mathsf{Verify}(\mathcal{T} := \{\tau_i \mid i \in I\}) \rightarrow \mathsf{yes}/\mathsf{no}$. *Takes as input any subset of t or more verification strings indexed by the set $I \subseteq \{1, \ldots, s\}$. Outputs* yes *if it holds that $C(x) = 1$.*

A SNIP must satisfy the correctness, soundness, zero-knowledge*, and* efficiency *properties of a multi-verifier zero knowledge proof system [7, 51].*

- **Correctness.** For all arithmetic circuits C and for all x such that $C(x) = 1$, then for all subsets $I \subseteq \{1, \ldots, s\}$ such that $|I| \geq t$,

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

  where the probability is taken over the randomness of Share and Prove.
- **Soundness.** There exists a negligible function negl such that for all efficient $\mathcal{A}$, subsets $I \subseteq \{1, \ldots, s\}$ where $|I| \geq t$, and x such that $C(x) \neq 1$,

$$\Pr\left[\begin{array}{l}([\pi^*]_1, \ldots, [\pi^*]_s) \leftarrow \mathcal{A}(1^\lambda, C); \\ \{\tau_i \leftarrow \mathsf{Audit}([x]_i, [\pi^*]_i) \mid i \in I\} : \\ \mathsf{Verify}(\{\tau_i \mid i \in I\}) = \mathsf{yes}\end{array}\right] \leq \mathsf{negl}(\lambda).$$

  where the probability is taken over the randomness of $\mathcal{A}$.
- **Zero-knowledge.** For all subsets $I \subseteq \{1, \ldots, s\}$ such that $|I| < t$, define $J\{1, \ldots, s\} \setminus I$ and let $\mathcal{D}_{I,J}$ be the distribution over $\{[(\pi]_i, \tau_i^* \mid i \in I\} \cup \{\tau_j \mid j \in J\}$ where each π_i is sampled according to Prove, each τ_i^* is sampled arbitrarily, and each τ_j is sampled according to Audit. A SNIP 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, no subset of fewer than t (possibly malicious) verifiers gain any information about x (in the information theoretic sense).
- **Efficiency.** The size of each proof share $[\pi]_i$ is bounded by the number of multiplication gates in C and size of τ_i is constant (in the security parameter).

Fiat-Shamir for SNIPs. The Fiat-Shamir transform [28] 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 [7, 51]. 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).

As observed in [7], the main challenge in the distributed setting is that the shares of each verifier must be kept secret from the other verifiers. If Fiat-Shamir were to be applied directly over the proof share given to each verifier, then it could leak information about the proof shares of other verifiers (and possibly leak information about x). The high-level idea to get around this problem is to have the prover *randomize* each share with a random nonce, which it distributes to the verifiers along with the proof shares. Using the nonce, the verifiers are able to verify consistency of the challenge used to check the proof without learning any information on the secret shares held by the other verifiers. (See [7, 6.2.3] for details.)

How Fiat-Shamir is applied to SNIPs. When verifying a SNIP proof, the verifiers first jointly sample a random value r and proceed to perform a randomized polynomial identity test using the DeMillo-Lipton-Schwartz–Zippel lemma [24, 44]. The Fiat-Shamir transform can be used to let the prover sample the "challenge" randomness r on behalf of the verifiers using a random oracle. The verifiers then simply check the consistency of the proof (i.e., verify the polynomial identity test), which only requires exchanging one message.

Remark 3. Note that we implicitly apply the above template of SNIPs with Fiat-Shamir in our SPoSS construction of Section 5.1 to minimize interaction between verifiers down to one message.

8.2 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 naïve 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 [16, 36] (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 $\mathsf{VC.Verify}(\mathsf{c}, \mathsf{vk}_i, e_i) = 1$. See Figure 4 for an example where we instantiate the vector commitment using a Merkle tree [36]. 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 $\mathsf{FSS.Gen}$.

Proposition 2. *Algorithm 6 satisfies the completeness, soundness, privacy, and efficiency guarantees of Definition 4.*

Proof (sketch). We give a high-level proof sketch for each property in turn.

Completeness. Completeness follows from the correctness of vector commitment scheme and completeness of SNIPs (Definition 9). If the FSS shares of f_i are generated correctly (output by $\mathsf{FSS.Gen}$, then π_0 is accepted by the completeness of SNIPs. Similarly, for π_1 and π_2, where completeness is easy to see. As a result, Verify outputs yes.

Soundness. Soundness follows from the binding property of the vector commitment scheme and the soundness guarantee of SNIPs. Specifically, π_0 proves that $[f_i]$ is the output of $\mathsf{FSS.Gen}$, π_1 proves that the prover knows a pre-image of the form $f_i \| \mathsf{sk}$ for the verification key $\tilde{\mathsf{vk}}$, and π_2 proves that $\tilde{\mathsf{vk}}$ is in the key list Λ (i.e., a valid verification key). Note that the same f_i (which the verifiers hold secret share of) is used by the verifiers when checking validity of π_0, π_1, and π_2, which binds the proofs together.

Privacy. Privacy follows directly from the privacy guarantee of SNIPs since the only information revealed is (1) the proof shares $[\pi_0]$, $[\pi_1]$, and $[\pi_2]$ and (2) the SNIP verification strings.

Efficiency. The size of $[\pi_0]$ is proportional to the number of multiplication gates in $\mathsf{FSS.Gen}$. The size of $[\pi_1]$ is proportional to the number of multiplication gates in $\mathsf{VC.Verify}$ (bounded by $O(\log |\Lambda|)$ when instantiated using a Merkle tree [36]). The audit token consists of the SNIP verification strings, which are of size $O(\lambda)$. $\square$

Algorithm 6: Generic PACL with SNIPs

[t] **Public parameters**: CRHF H, vector commitment c to the list of verification keys Λ, and commitment openings $e_1, \dots, e_N$ to each verification key in Λ.

– $\mathsf{KeyGen}(1^\lambda, f_i)$

1: Sample $\mathsf{sk}_i \leftarrow_R \{0,1\}^\lambda$.
2: Compute $\mathsf{vk}_i := H(f_i \| \mathsf{sk})$.
3: Output $(\mathsf{sk}_i, \mathsf{vk}_i)$.

– $\mathsf{Prove}(f_i, \mathsf{sk})$:

1: Compute $\tilde{\mathsf{vk}} := H(f_i \| \mathsf{sk})$ and $\tilde{e} := e_i$.
2: Compute SNIP proofs (using the Fiat-Shamir transform) for the following statements:
  2.1: $[\pi_0]$ for the statement "$\mathsf{FSS.Gen}(1^\lambda, f_i; r) = [f_i]$".
  2.2: $[\pi_1]$ for the statement "$\tilde{\mathsf{vk}} = H(f_i, \mathsf{sk})$".
  2.3: $[\pi_2]$ for the statement "$\mathsf{VC.Verify}(\mathsf{c}, \tilde{\mathsf{vk}}, \tilde{e}) = 1$".
3: Output proof shares $[\pi] := ([\pi_0], [\pi_1], [\pi_2], [\tilde{\mathsf{vk}}], [\tilde{e}])$.

– $\mathsf{Audit}(\Lambda, [f], [\pi])$:

1: Parse $[\pi] := ([\pi_0], [\pi_1], [\pi_2], [\tilde{\mathsf{vk}}], [\tilde{e}])$.
2: Compute the SNIP verification strings τ_0, τ_1, and τ_2 for $[\pi_0]$, $[\pi_1]$, and $[\pi_2]$, respectively.
3: Output audit token $\tau := (\tau_0, \tau_1, \tau_2)$.

– $\mathsf{Verify}(\mathcal{T})$:

1: Parse each $\tau_i \in \mathcal{T}$ as $\tau_i = (\tau_i^{(0)}, \tau_i^{(1)}, \tau_i^{(2)})$.
2: $\mathcal{T}^{(j)} := \left\{ \tau_i^{(j)} \mid i \in \{1, \dots, |\mathcal{T}|\} \right\}$, for all $j \in \{0,1,2\}$.
3: Output yes if and only if all three sets $\mathcal{T}^{(0)}$, $\mathcal{T}^{(1)}$, $\mathcal{T}^{(2)}$ of SNIP verification strings are accepting.

Remark 4 (Efficiency). The proof size for our generic FSS PACL construction is proportional to the size of the Gen circuit. However, FSS is only concerned with the *share size* and not the running time of Gen. As such, it is possible that Gen runs in time proportional to the truth table of the function being secret shared (e.g., see *programmable* DPFs [13]). In this case, the size of the SNIP proof—which has size proportional to the circuit size of Gen—can negate the efficiency of FSS scheme. An interesting direction for future work would be to make the proof size only depend on the size of the function secret shares and be independent of Gen. However, we note that in all known constructions of FSS for $P/poly$ [9, 25], Gen runs in sublinear time relative to the truth table.

9 Implementation and evaluation

In this section, we describe our implementation and evaluation of the (V)DPF-PACL constructions from Sections 4 and 5. (We do not implement our $i\mathcal{O}$-based PACLs and generic constructions as they are primarily of theoretical interest.) Our evaluation focuses on the state-of-the-art two-party FSS schemes [11, 23]. Multi-party FSS constructions are less efficient [9] or require heavier cryptographic assumptions, making them concretely slower [19, 38]. 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. 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 4) and as $\mathbb{Z}_{2^\lambda}$ for symmetric-key (sym) soundness (Definition 5). 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 [11, 23].

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.

9.1 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 [9, 11] 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 $\mathbb{Z}_p^*$).

9.2 Communication costs

Tables 1 and 2 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 [11]).

In Table 1, we compare communication costs to other approaches for DPF access control. Express [27] and Sabre [49] operate in the symmetric-key (sym) setting, satisfying the relaxed PACL soundness definition (Section 3.2). Newman et al. [38] construct an access control mechanism similar to Algorithm 1 satisfying the soundness definition of Definition 4. Using SPoSS for verifying discrete-logarithm knowledge results in over 2,400× less communication compared to a naïve approach (described in Appendix D) and a 1,000× smaller proof size compared to Sabre [49].

Table 1: Proof size (Prover → Verifier) and audit token size (Verifier ↔ Verifier) for (V)DPF-PACL with the match predicate for access control (Algorithm 1). *Estimated (see Appendix D).

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 (naïve)*	4.7 MB	816 B
Express [27] (sym)	2 kB	184 B
Spectrum [38]	32 B	64 B
Sabre [49] (sym)	$(40 + 120n)$ kB	16 B

Table 2: 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 [27, 38, 49] do not support inclusion predicates. *Estimated (see Appendix D).

(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 (naïve)*	$(4.7 \times 10^6 + s_\ell)$ B	816 B

Table 3: 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)	
		sym	pub	sym	pub
DPF	1.41 μs	1.42 μs	5.66 μs	33.10 μs	84.11 μs
DMPF	90.44 μs	90.46 μs	93.26 μs	122.22 μs	213.47 μs
	VFSS.Eval **Baseline**	VDPF-PACL (match predicate)		VDPF-PACL (inclusion predicate)	
		sym	pub	sym	pub
DPF	1.46 μs	1.47 μs	1.69 μs	33.66 μs	36.11 μs
DMPF	93.14 μs	93.19 μs	93.50 μs	125.52 μs	128.71 μs

9.3 Verification costs

We report the processing time in Figures 5 and 6. 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 5), 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 6), 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 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 5 and 6 in Table 3, where we amortize over 100,000 evaluations of the DPF and DMPF.

9.4 Applications of PACLs

Private databases with access control. Systems that use multi-server PIR (e.g., [19, 21, 22, 32, 34, 50]) can take advantage of (V)DPF-PACLs to restrict access to database items. (Gupta et al. [32] specifically leave open the problem of supporting *authenticated* media consumption through Popcorn.) Other systems such as Dory [21] use DPFs for private keyword queries in a remote database. For example, Wang et al. [50] use PIR (realized using DPFs) to build privacy-preserving restaurant, geolocation, and flight searches. Gupta et al. [32] use PIR for privacy-preserving movie streaming.

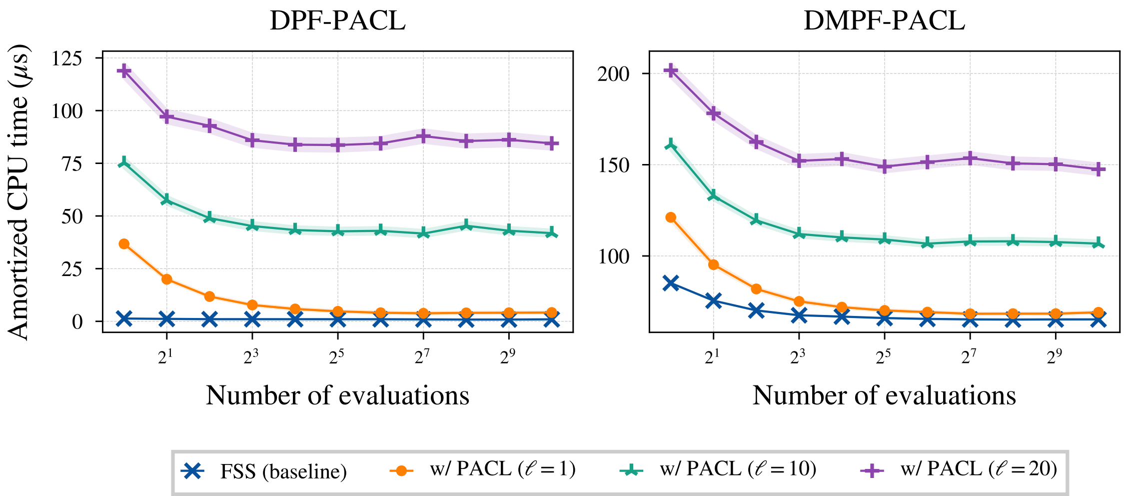

Fig. 5: Our DPF-PACL construction amortizes with more evaluations of the DPF but plateaus at approximately 32 evaluations. For DMPFs, which have a higher baseline processing time, the overhead is less significant thanks to aggregation, which amortizes the overhead of access control (Section 4.3).

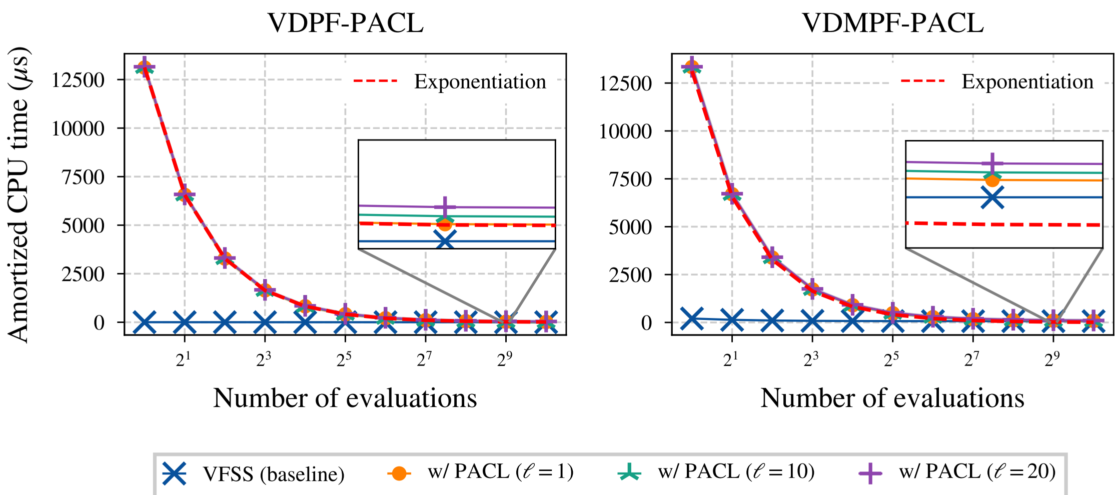

Fig. 6: Our VDPF-PACL construction is dominated by the exponentiation in $\mathbb{Z}_p^*$ (dashed lined) and has a higher *initial* overhead compared to the DPF-PACL construction. In contrast to DPF-PACLs, our VDPF-PACL construction amortizes almost entirely after 64 evaluations and has an asymptotically smaller overhead.

In Figure 7, 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 [41] describe a read-and-write private database, which can be realized using DPFs [11]. Applying VDPF-PACLs to this setting would result in similar overheads to the PIR setting.

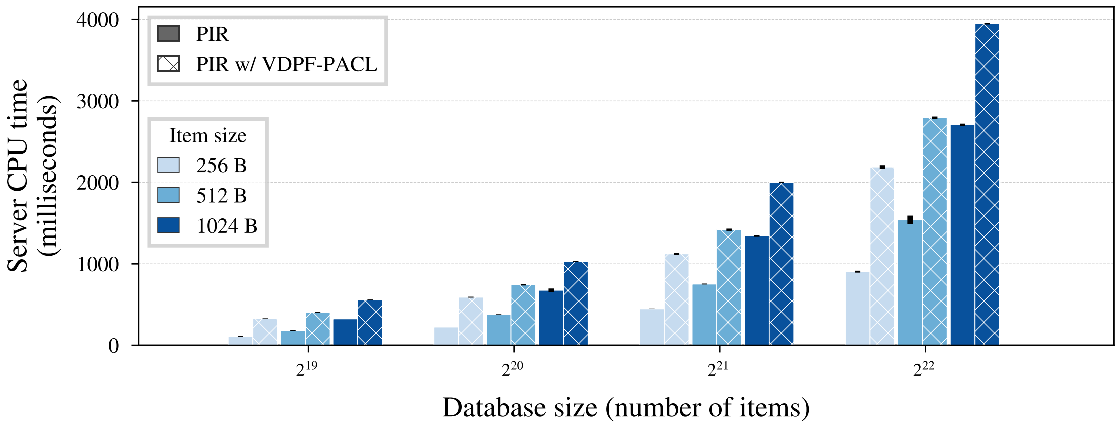

Fig. 7: Server-side CPU time for privately retrieving an item from a database through two-server PIR with and without VDPF-PACLs. Adding access control has a 1.5–3× impact on processing time. The access control overhead is automatically amortized over the number of items in the database. The relative 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 [31]) 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 [27], Sabre [49] and Spectrum [38] to improve their concrete performance. In Figure 8, we show that swapping their implicit access control mechanism with our VDPF-PACL construction improves performance by a factor of 50–70×. Sabre [49]'s computational overhead is on-par with baseline FSS (and DPF-PACLs satisfying symmetric-key soundness Section 3.2 and Table 1) but requires significantly larger proofs for access control purposes (approximately 3.5 MB; see Table 1). However, we note that Sabre [49] achieves other nice properties that are tailored to anonymous communication (see Section 1.2).

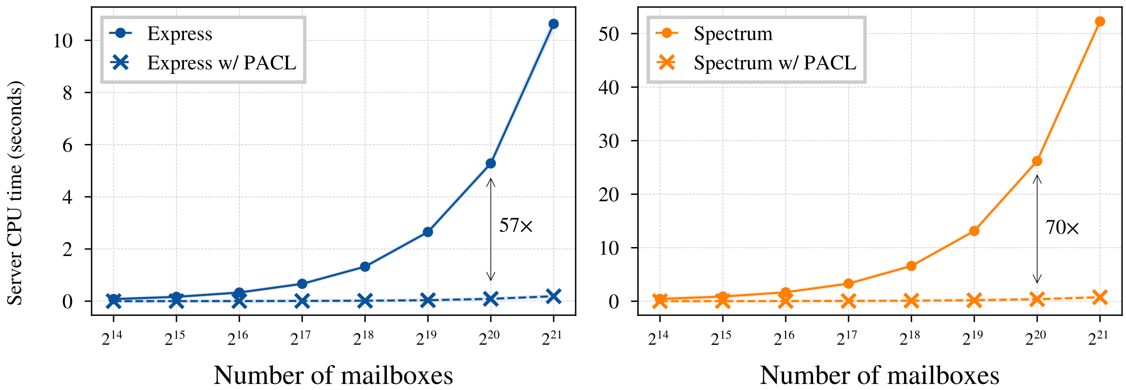

Fig. 8: Our VDPF-PACLs significantly improve performance of anonymous communication systems that require access control. These performance improvements come from two sources: (1) the ability to use optimized DPF constructions with our DPF-PACLs (e.g., the access control in Express requires concretely slower DPF constructions) and (2) the better amortization of VDPF-PACLs.

10 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 thank the organizers of the MIT PRIMES program for making it a such a fun and enriching experience.

Bibliography

[1] PACL source code. https://github.com/sachaservan/pacl, 2022.

[2] Abdelrahaman Aly, Aysajan Abidin, and Svetla Nikova. Practically efficient secure distributed exponentiation without bit-decomposition. In *International Conference on Financial Cryptography and Data Security*, pages 291–309. Springer, 2018.

[3] Saikrishna Badrinarayanan, Vipul Goyal, Aayush Jain, and Amit Sahai. Verifiable functional encryption. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 557–587. Springer, 2016.

[4] Boaz Barak, Oded Goldreich, Rusell Impagliazzo, Steven Rudich, Amit Sahai, Salil Vadhan, and Ke Yang. On the (im)possibility of obfuscating programs. In *Annual international cryptology conference*, pages 1–18. Springer, 2001.

[5] Donald Beaver. Efficient multiparty protocols using circuit randomization. In *Annual International Cryptology Conference*, pages 420–432. Springer, 1991.

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

[7] 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*, pages 67–97. Springer, 2019.

[8] 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)*, pages 762–776. IEEE, 2021.

[9] Elette Boyle, Niv Gilboa, and Yuval Ishai. Function secret sharing. In *Annual international conference on the theory and applications of cryptographic techniques*, pages 337–367. Springer, 2015.

[10] Elette Boyle, Niv Gilboa, and Yuval Ishai. Breaking the circuit size barrier for secure computation under DDH. In *Annual International Cryptology Conference*, pages 509–539. Springer, 2016.

[11] 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*, pages 1292–1303, 2016.

[12] 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*, pages 871–900. Springer, 2021.

[13] Elette Boyle, Niv Gilboa, Yuval Ishai, and Victor I Kolobov. Programmable distributed point functions. In *Annual International Cryptology Conference*, pages 121–151. Springer, 2022.

[14] Ran Canetti, Huijia Lin, Stefano Tessaro, and Vinod Vaikuntanathan. Obfuscation of probabilistic circuits and applications. In *Theory of Cryptography Conference*, pages 468–497. Springer, 2015.

[15] Ran Canetti, Suvradip Chakraborty, Dakshita Khurana, Nishant Kumar, Oxana Poburinnaya, and Manoj Prabhakaran. COA-secure obfuscation and applications. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 731–758. Springer, 2022.

[16] Dario Catalano and Dario Fiore. Vector commitments and their applications. In *International Workshop on Public Key Cryptography*, pages 55–72. Springer, 2013.

[17] Benny Chor, Oded Goldreich, Eyal Kushilevitz, and Madhu Sudan. Private information retrieval. In *Proceedings of IEEE 36th Annual Foundations of Computer Science*, pages 41–50. IEEE, 1995.

[18] 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)*, pages 259–282, 2017.

[19] 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*, pages 321–338. IEEE, 2015.

[20] Ivan Damgård, Matthias Fitzi, Eike Kiltz, Jesper Buus Nielsen, and Tomas Toft. Unconditionally secure constant-rounds multi-party computation for equality, comparison, bits and exponentiation. In *Theory of Cryptography Conference*, pages 285–304. Springer, 2006.

[21] 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)*, pages 1101–1119, 2020.
[22] Emma Dauterman, Mayank Rathee, Raluca Ada Popa, and Ion Stoica. Waldo: A private time-series database from function secret sharing. *Cryptology ePrint Archive*, 2021.
[23] 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*, pages 150–179. Springer, 2022.
[24] Richard A DeMillo and Richard J Lipton. A probabilistic remark on algebraic program testing. Technical report, Georgia Inst Tech Atlanta School of Information and Computer Science, 1977.
[25] Yevgeniy Dodis, Shai Halevi, Ron D Rothblum, and Daniel Wichs. Spooky encryption and its applications. In *Annual International Cryptology Conference*, pages 93–122. Springer, 2016.
[26] Jack Doerner and abhi shelat. Scaling ORAM for secure computation. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 523–535, 2017.
[27] 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)*, pages 1775–1792, 2021.
[28] 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*, pages 186–194. Springer, 1986.
[29] Craig Gentry, Philip MacKenzie, and Zulfikar Ramzan. A method for making password-based key exchange resilient to server compromise. In *Annual International Cryptology Conference*, pages 142–159. Springer, 2006.
[30] Niv Gilboa and Yuval Ishai. Distributed point functions and their applications. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 640–658. Springer, 2014.
[31] Matthew Green. Let's talk about PAKE, october 2018. https://blog.cryptographyengineering.com/2018/10/19/lets-talk-about-pake/, 2018. Accessed December 2022.
[32] 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)*, pages 91–107, 2016.
[33] Yizhou Huang and Ian Goldberg. Outsourced private information retrieval. In *Proceedings of the 12th ACM Workshop on Workshop on Privacy in the Electronic Society*, pages 119–130, 2013.
[34] Ari Juels. Targeted advertising... and privacy too. In *Cryptographers' Track at the RSA Conference*, pages 408–424. Springer, 2001.
[35] Jonathan Katz and Yehuda Lindell. *Introduction to modern cryptography*. CRC press, 2020.
[36] Ralph C Merkle. A digital signature based on a conventional encryption function. In *Conference on the theory and application of cryptographic techniques*, pages 369–378. Springer, 1987.
[37] 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)*, pages 2505–2522, 2020.
[38] 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.
[39] Chao Ning and Qiuliang Xu. Constant-rounds, linear multi-party computation for exponentiation and modulo reduction with perfect security. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 572–589. Springer, 2011.
[40] Takashi Nishide and Kazuo Ohta. Multiparty computation for interval, equality, and comparison without bit-decomposition protocol. In *International Workshop on Public Key Cryptography*, pages 343–360. Springer, 2007.
[41] Rafail Ostrovsky and Victor Shoup. Private information storage. In *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, pages 294–303, 1997.
[42] 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.
[43] Claus-Peter Schnorr. Efficient signature generation by smart cards. *Journal of cryptology*, 4(3): 161–174, 1991.

[44] Jacob T Schwartz. Fast probabilistic algorithms for verification of polynomial identities. *Journal of the ACM (JACM)*, 27(4):701–717, 1980.
[45] Adi Shamir. How to share a secret. *Communications of the ACM*, 22(11):612–613, 1979.
[46] Joseph H Silverman. *The arithmetic of elliptic curves*, volume 106. Springer, 2009.
[47] John T Tate. The arithmetic of elliptic curves. *Inventiones mathematicae*, 23(3):179–206, 1974.
[48] Tomas Toft. Constant-rounds, almost-linear bit-decomposition of secret shared values. In *Cryptographers' Track at the RSA Conference*, pages 357–371. Springer, 2009.
[49] 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.
[50] 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)*, pages 299–313, 2017.
[51] Kang Yang and Xiao Wang. Non-interactive zero-knowledge proofs to multiple verifiers. *Cryptology ePrint Archive*, 2022.

A Verifiable DPFs

de Castro and Polychroniadou [23] present definitions for $(2,2)$-VDPF constructions. We generalize their definitions to any (t,s)-VDPF scheme.

Definition 10 (VDPF [23]). *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 function $P_{i,m}$ such that $P_{i,m}(i) = m$.*

– $\mathsf{Eval}(\kappa, X \subseteq \{0,1\}^n) \to ([\boldsymbol{v}], \rho)$. *Takes as input an evaluation key κ and a subset of values in the domain. Outputs a secret share of a vector $\boldsymbol{v}$, where the jth coordinate of $\boldsymbol{v}$ corresponds to a share of $P_{i,m}(j)$ for $j \in X$, and a verification string ρ.*

– $\mathsf{Verify}(\{\rho_i \mid i \in I\}) \to \mathsf{yes}/\mathsf{no}$. *Takes as input any subset of t or more verification strings indexed by the set $I \subseteq \{1, \ldots, s\}$. Outputs* yes *if and only if $\{[\boldsymbol{v}]_i \mid i \in I\}$, as output by* Eval *using $\{\kappa_i \mid i \in I\}$, 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). 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 $j \in \{0,1\}^n$, for all $m \in \mathbb{F}$, and all subsets $I \subseteq \{1, \ldots, s\}$, such that $|I| \geq t$, there exists an efficient algorithm Decode such that

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

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_i sampled according to $(\_, \rho_i) \leftarrow \mathsf{Eval}(\kappa_i^*, X^*)$ for $i \in I$, $|I| \geq t$, it holds that

$$\Pr[\mathsf{Verify}(\{\rho_i \mid i \in I\}) = \mathsf{yes}] \geq 1 - \mathsf{negl}(\lambda),$$

if and only if the correctness property of FSS is satisfied, where the probability is taken over the adversary's choice of randomness.

– **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 $P_{i,m}$ encoded in the set of keys $(\kappa_1, \ldots, \kappa_s)$.

B Deferred proofs

B.1 Proof of Lemma 1

The proof hinges on the aggregation property of our construction (Section 4.3). Consider an $\mathcal{A}$ that outputs $\hat{f}_\gamma$ and $\hat{\pi}$ where $\hat{f}$ is *not* a point function (and also not the trivial identity function $f(x) = 1_{\mathbb{G}}$ 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 $\textsc{PkSoundness}_{\text{PACL},\mathcal{A},I}(\lambda)$ 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 $\mathcal{A}$.

B.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 $\textsc{PkSoundness}_{\text{PACL},\mathcal{A},I}(\lambda)$ 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 $|I| \geq t$,

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

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**.

Obtain output $([\hat{f}_\gamma], [\hat{\pi}])$ from $\mathcal{A}$.
Recover $\hat{f}_\gamma$ and $(\hat{f}_\omega, \hat{\alpha}, \hat{\beta})$ from $[\hat{f}_\gamma]$ and $[\hat{\pi}]$, respectively.
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^λ, index subset I, and $\{\tau_i^* \mid i \in I\}$, $\mathcal{S}$ proceeds as follows:

parse $\tau_i^* = (\hat{\tau}_i^{(0)}, \hat{\tau}_i^{(1)})$ for all $i \in I$.
$J := \{1, \ldots, s\} \setminus I$.
$([z]_1, \ldots, [z]_s) \leftarrow_R \mathbb{Z}_p^s$.
$([z']_1, \ldots, [z']_s) \leftarrow_R \mathbb{Z}_p^s$.
$(\kappa'_1, \ldots, \kappa'_s) \leftarrow \mathsf{DPF.Gen}(1^\lambda, P_1)$.
$\mathsf{view} \leftarrow \mathcal{S}'(1^\lambda, I, \left\{\hat{\tau}_i^{(0)} \mid i \in I\right\})$.
parse $\mathsf{view} = \left\{\left([\pi']_i, \hat{\tau}_i^{(0)}\right) \mid i \in I\right\} \cup \left\{\tau_j^{(0)} \mid j \in J\right\}$.
$[\pi]_i := ([z']_i, [\pi']_i, \kappa'_i)$ and $\tau_j^{(1)} := g^{[z]_j}$ for all $i \in I$.
$\tau_j := (\tau_j^{(0)}, \tau_j^{(1)})$ for all $j \in J$.
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) $[z]$ is uniformly random and therefore distributed identically to $[\alpha]$ 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), and (4) the audit tokens are (computationally-hiding) multiplicative secret shares of $(1_\mathbb{G}, 1_\mathbb{G}) \in \mathbb{G} \times \mathbb{G}$ in the real view and uniformly random multiplicative shares from $(\mathbb{G} \times \mathbb{G})$ in the output of $\mathcal{S}$. An efficient distinguisher for (4) would also contradict the privacy property of FSS.

Efficiency. Using the "FSS tensoring" [11] 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 $\{1, \ldots, \ell\}$.

B.3 Security of SPoSS

Proposition 3. *The SPoSS construction in Algorithm 3 satisfies the correctness, argument-of-knowledge, and zero-knowledge properties (Definition 6) required of a secret-shared non-interactive proof system, in the random oracle model [7].*

Proof of Proposition 3. 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:

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

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

2: 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 [18], 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 in the natural way). On input $(1^\lambda, \{i\}, \{\tau_i^*\})$, $\mathcal{S}$ proceeds as follows:

1: $j \in \{\text{A}, \text{B}\} \setminus \{i\}$.

2: $[w]_j, [c]_i, u, d, e \leftarrow_R \mathbb{Z}_p$, $r, r_j, z_i \leftarrow \{0,1\}^\lambda$.

3: $[\pi]_i := ([x]_i, u, [c]_i, r, d, e, z_i)$.

4: **if** $j = \text{A}$ **then** : $\tau_j \leftarrow ([w]_j, r_j, r, d, d, e)$.
else : $\tau_j \leftarrow ([w]_i, r, r, e, d, e)$.

5: Output $\{([\pi]_i, \tau_i^*), \tau_j\}$.

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 [7]. 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 identical to the view of verifier i in Algorithm 3, which concludes the proof of zero-knowledge.

C Beaver's Protocol

We provide a brief overview of Beaver's [5] approach to multiplication of secret shares as adapted by Corrigan-Gibbs and Boneh [18]. We focus on the setting with two parties; see [18] 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 [5].

D Naïve SPoSS using SNIPs

Here, we estimate the overhead of naïvely 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) or even tailored techniques (e.g., [2, 20, 39, 40, 48]), requires computing a bit decomposition to convert the secret-shared exponent into binary secret-shares. A bit decomposition circuit requires $O(\lambda \log \lambda)$ multiplication gates, where $\lambda := \lceil \log p \rceil$ [20, 40, 48]. Using the explicit multiplication complexity listed by Toft [48, Table 2] for their bit decomposition protocol, we get that the total number of multiplications to compute the circuit is $M = 31 \cdot \lambda \log(\lambda) + 71 \cdot \lambda + 30 \cdot \sqrt{\lambda}$. When $\mathbb{G} = \mathbb{Z}_p^*$ and $\log p \approx 3072$, then we have:

$$M = 32 \cdot 3072 \cdot \log(3072) + 71 \cdot 3072 + 30 \cdot \sqrt{3072} \approx 1.4 \times 10^6.$$

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). This results in $(3 \cdot 3072) + 2 \cdot (3072) \cdot M$ bits $= (9216) + (6144) \cdot (1.4 \times 10^6)$ bits ≈ 1 GB of communication per verifier.

However, we can be more clever and have the prover 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. We now describe the arithmetic circuit computing an exponentiation with a bit-decomposed exponent and repeated squaring. To the best of our knowledge, this is the most optimal generic approach to verifying a Schnorr proof over secret shares.

First, the verifiers 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:

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

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 [46, 47]. The advantage of working with $\mathbb{Z}_p^*$ is that we only require *one* multiplication in $\mathbb{Z}_p$ to apply the group operation.