LSS Homomorphisms and Applications to Secure Signatures, Proactive Secret Sharing and Input Certification

Diego Aranha, Anders Dalskov, Daniel Escudero, and Claudio Orlandi

Aarhus University

Abstract. In this paper we present the concept of linear secret-sharing homomorphisms, which are linear transformations between different secret-sharing schemes defined over vector spaces over a field $\mathbb{F}$ and allow for efficient multiparty conversion from one secret-sharing scheme to the other. This concept generalizes the observation from (Smart and Talibi, IMACC 2019) and (Dalskov et al., EPRINT 2019) that moving from a secret-sharing scheme over $\mathbb{F}$ to a secret sharing over an elliptic curve group $\mathbb{G}$ of order p can be done very efficiently with no communication by raising the generator of $\mathbb{G}$ to each share over $\mathbb{F}$. We then show how to securely evaluate arbitrary bilinear maps, which can be instantiated in particular with pairings over elliptic curves.
We illustrate the benefits of being able to efficiently perform secure computation over elliptic curves by providing several applications and use-cases. First, we show methods for securely encoding and *decoding* field elements into elliptic curve points, which enable applications that require computation back and forth between fields and elliptic curves. Then, we show how to use use the secure pairing evaluation to sign and verify Pointcheval-Sanders signatures (D. Pointcheval and O. Sanders, CT-RSA 2016) in MPC, which enable multiple applications in which some authenticity property is required on secret-shared data. We consider some of these applications in our work, namely Dynamic Proactive Secret Sharing, on which a shared secret is intended to be transferred from one set of parties to another, and Input Certification, on which the "validity" of the input provided by some party to some MPC protocol can be verified.

1 Introduction

A (t, n)-secure secret-sharing scheme allows a secret to be distributed into n shares in such a way that any set of at most t shares are independent of the secret, but any set of at least $t + 1$ shares together can completely reconstruct the secret. In a *linear* secret-sharing schemes (LSSS), shares of two secrets can be added together to obtain shares of the sum of the secrets. A popular example of a $(n - 1, n)$-secure LSSS is additive secret sharing, whereby a secret $s \in \mathbb{F}$ (here $\mathbb{F}$ denotes integers modulo a prime p) is secret-shared by sampling uniformly random $s_1, \ldots, s_n \in \mathbb{F}$ subject to $s_1 + \cdots + s_n \equiv s \bmod p$. Another popular

choice of a (t, n)-secure LSSS is Shamir secret sharing [35], that distributes a secret $s \in \mathbb{F}$ by sampling a random polynomial $f(x)$ over $\mathbb{F}$ of degree at most t such that $f(0) = s$, and setting the i-th share to be $s_i = f(i)$.

Linear secret-sharing schemes are information-theoretic in nature, that is, they do not rely on any computational assumption and therefore tend to be very efficient. Furthermore, they are widely used in multiple applications like distributed storage [21] or secure multiparty computation [12]. These schemes can be augmented with techniques from public-key cryptography, like elliptic-curve cryptography. As an example, consider (a variant of) Feldman's scheme for verifiable secret sharing[1] [18]: To distribute a secret $s \in \mathbb{F}$, the dealer samples a polynomial of degree at most t such that $f(0) = s$, say $f(x) = s + r_1 x + \cdots + r_t x^t$, and sets the i-th share to be $s_i = f(i)$. On top of this, the dealer publishes $s \cdot G, r_1 \cdot G, \ldots, r_t \cdot G$, where G is a generator of some elliptic-curve group $\mathbb{G}$ of order p for which the discrete-log problem is hard. With this, each party can detect if its share s_i is correct by computing $s_i \cdot G$ and checking that it equals $s \cdot G + i^1(r_1 G) + i^2(r_2 G) + \cdots + i^t(r_t G)$.

Similar approaches have also been used to instantiate polynomial commitments [26], or to securely compute ECDSA signatures [15,36]. The key idea behind these techniques is that the group $\mathbb{G}$, having order p, is homomorphic to $\mathbb{F}$ as an additive group. Given this, a linear secret sharing scheme over $\mathbb{F}$, that satisfies some kind of "homomorphism" mod p, turns out to be "compatible" with arithmetic over $\mathbb{G}$ as well. This interaction enables applications that exploit the power of primitives that require computational assumptions, like commitments or signatures, together with efficient distributed information-theoretic techniques of linear secret sharing.

In this work we formalize and generalize the above notion by using an adequate mathematical definition of LSSS, extending it to general vector spaces, of which elliptic curves are particular cases, and using linear transformations between these vector spaces to convert from one secret-shared representation to a different one. Furthermore, we extend this notion from linear transformation to bilinear maps, and show how generic multiplication triples over $\mathbb{F}$ can be used to securely compute these. We then show that these simple but powerful techniques can be used to improve the efficiency of some relevant problems. This was already shown in [15,36] to securely evaluate ECDSA signatures. Here we consider other applications: More efficient signatures, proactive secret sharing and input certification.

1.1 Our Contributions

Our contributions can be summarized as follows.

LSS homomorphisms and bilinear maps. We introduce the concept of *LSSS homomorphisms*, which generalize the folklore notion of "putting the share in the

[1] A verifiable secret sharing scheme is one in which parties can verify that the dealer shared the secret correctly

exponent". Then, with the mathematical foundations of LSSS homomorphisms, we show how generic multiplication triples can be used to securely evaluate *any* bilinear map, using *any* linear secret-sharing scheme. This is presented in Section 2. Furthermore, our experimental results in Section 8 show that the performance penalty per party for a pairing computation in MPC is only 30% compared to computing a pairing in a non-MPC context, without considering the cost of communication or preprocessing.

Secure computation over elliptic curves. Our treatment in Section 2 is completely general, applying to arbitrary vector spaces and also arbitrary linear secret-sharing schemes. Moreover, in Section 3 we instantiate our constructions with popular secret-sharing schemes—specifically additive, Shamir and replicated secret sharing—and we use pairings over elliptic-curves for the instantiation of bilinear maps and vector spaces. This allows us to perform arbitrary computations over elliptic curves which use group addition and pairings. However, since the most natural use-cases for elliptic curves include operating on strings that are *encoded* into elliptic curve points, to be decoded later on, we present in Section 4 how to securely encode secret shared data into secret-shared elliptic curve points, as well as how to decode securely.

Secure evaluation and verification of signatures. We show how to securely sign secret-shared data, and how to securely verify secret-shared signatures on secret-shared data, in Section 5. To this end, we make use of Pointcheval-Sanders (PS) signatures [33], which are a more efficient and compact drop-in replacement for the Camenisch-Lysyanskaya signatures widely used in anonymous credentials [10]. General-purpose MPC protocols can be used, in principle, to evaluate arbitrary digital signature schemes securely. However, this typically requires computing a hash function on the data to be signed, which is not particularly efficient to do in MPC. Our approach constitutes the first efficient protocol for signing and verifying in MPC, requiring only 1 round and a secure dot product for signing, and 1 round with a single secure scalar multiplication (shared scalar multiplied by a shared point) for verification. Finally, our experimental results from Section 8 show that generating keys and verifying signatures using MPC is at most twice as expensive in comparison to local computation. Signing multiple messages reduces this performance penalty significantly by exploiting features of the PS signature scheme.

Dynamic proactive secret-sharing. As an application of secure signatures, we consider in Section 6 the problem of dynamic proactive secret sharing, where a shared secret among multiple parties is transferred to a new set of parties while preserving the privacy of the secret. Dynamic proactive secret-sharing is very relevant primitive in the context of for example blockchains, where an initial set of parties hold a secret-shared key, and wish to transfer it to a different set of parties. Thus, this technique allows one to securely move control of an account from one committee to a different one (where these two committees can differ in both number of parties and the threshold). We propose a *novel*

and *simple* protocol for proactive secret sharing with abort, where our protocol for secure signatures is used to ensure that the secret is kept untouched during transfer. In a nutshell, our protocol works by first transmitting the message *and* a signature via a cheap but malleable proactivization method, followed by a signature verification at the receiving end. This approach improves the communication of the state of the art, CHURP [29], by a factor of 9, on top of being fundamentally simpler.

Input certification. Finally, we apply our protocol for secure signatures to the task of input certification in MPC. In this setting, some parties engage in an MPC protocol, but the input they provide, on top of being private, must be "cetified" by some trusted authority. Previous works considered either garbled circuits or zero knowledge proofs. In our work, we consider a much simpler solution in which the trusted authority provides each party with a signature on its input, which is presented at computation time, and verified securely in MPC. Our solution improves the state-of-the-art by a factor of at least 2 in terms of computation, on top of being conceptually much simpler.

We provide experimental results via a proof-of-concept implementation in Section 8.

1.2 Related Work

Previous works [15,36] make use of the folklore idea of "putting the shares in the exponent" to efficiently instantiate threshold ECDSA, among other things. They approach the problem from a more practical point of view, using certain specific protocols and focusing on the application at hand, whereas our work is more general, applying to *any* linear secret-sharing scheme and also any vector space homomorphism. Furthermore, these works did not consider the case of cryptographic pairings, as these are not needed in the ECDSA algorithm.

Multiple works have addressed the problem of proactive secret-sharing. It was originally proposed in [25,31], and several works have built on top of these techniques [24,34,5,4,29], including ours. Among these, the closest to our work is the state-of-the-art [29], which also makes use of pairing-friendly elliptic curves to ensure correctness of the transmitted message. However, a crucial difference is that in their work, a commitment scheme based on elliptic curves, coupled with the technique of "putting the share in the exponent" is used to ensure each player *individually* behaves correctly. Instead, in our work, we use elliptic curve computation on the secret rather than on the shares, which reduces the communication complexity, as shown in Section 6.

Finally, not many works have been devoted to the important task of input certification in MPC. For general functions, the only works we are aware of are [7,27,37,8]. Among these, only [8] tackles the problem from a more general perspective, having multiple parties and different protocols. However, their approach differs fundamentally from ours: In [8], the concept of signature schemes with privacy is introduced, which are signatures that allow for an interactive protocol for verification, in such a way that the privacy of the message is preserved.

Then, the authors present constructions of this type of signatures, and use them to solve the input certification problem. However, the techniques from [8] differ from ours at a fundamental level: Their protocols first compute a commitment of the MPC inputs, and then engage in an interactive protocol for verification to check the validity of these inputs. Furthermore, these techniques are presented separately for two MPC protocols: one from [17] and one from [16]. Instead, our results apply to *any* MPC protocol based on linear secret-sharing schemes, and moreover, is much simpler and efficient as no commitments, proofs of knowledge, nor special interactive verification is needed, beyond the MPC execution of a cheap verification protocol.

2 LSS Homomorphisms and Bilinear Maps

Let $\mathbb{F}$ be a prime field of order p. We use $a \in_R A$ to represent that a is sampled uniformly at random from the finite set A.

2.1 Linear Secret Sharing

In this section we define the notion of linear secret sharing that we will use throughout this paper. Most of the presentation here can be seen as a simplified verision of [13, Section 6.3], but it can also be regarded as a generalization since we consider arbitrary vector spaces.

Definition 1. *Let $\mathbb{F}$ be a field. A linear secret sharing scheme (LSSS) $\mathcal{S}$ over V for n players is defined by a matrix $M \in \mathbb{F}^{m\times(t+1)}$, where $m \geq n$, and a function* $\mathsf{label} : \{1,\ldots,m\} \to \{1,\ldots,n\}$. *We say M is the matrix for $\mathcal{S}$. We can apply* label *to the rows of M in a natural way, and we say that player $P_{\mathsf{label}(i)}$ owns the i-th row of M. For a subset A of the players, we let M_A be the matrix consisting of the rows owned by players in A.*

To secret-share a value $s \in V$, the dealer samples uniformly at random a vector $\boldsymbol{r}_s \in V^{t+1}$ such that its first entry is s, and sends to player P_i each row of $M \cdot \boldsymbol{r}_s$ owned by this player. We write $[\![s, \boldsymbol{r}_s]\!]$ for the vector of shares $M \cdot \boldsymbol{r}_s$, or simply $[\![s]\!]$ if the randomness vector $\boldsymbol{r}_s$ is not needed. Observe that the parties can obtain shares of $s_1 + s_2$ from shares of s_1 and shares of s_2 by locally adding their respective shares. We denote this by $[\![s_1 + s_2]\!] = [\![s_1]\!] + [\![s_2]\!]$.

The main properties of a secret sharing scheme are privacy and reconstruction, which are defined with respect to an access structure. In this work, and for the sake of simplicity, we consider only threshold access structures. That said, our results generalize without issue to more general access structures as well.

Definition 2. *An LSSS $\mathcal{S} = (M, \mathsf{label})$ is $(t, t+1)$-secure if the following holds:*

- *(Privacy) For all $s \in V$ and for every subset A of players with $|A| \leq t$, the distribution of $M\boldsymbol{r}_s$ is independent of s*
- *(Reconstruction) For every subset A of players with $|A| \leq t$ there is a reconstruction vector $\boldsymbol{e}_A \in \mathbb{F}^{m_A}$ such that $\boldsymbol{e}_A^{\mathsf{T}}(M_A\boldsymbol{r}_s) = s$ for all $s \in V$.*

2.2 LSS over Vector Spaces

Let V be a finite-dimensional $\mathbb{F}$-vector space, and let $\mathcal{S} = (M, \mathsf{label})$ be an LSSS over $\mathbb{F}$. Since V is isomorphic to $\mathbb{F}^k$ for some k, we can use the LSSS $\mathcal{S}$ to secret-share elements in V by simply sharing each one of its k components. This is formalized as follows.

Definition 3. *A linear secret-sharing scheme over a finite-dimensional $\mathbb{F}$-vector space V is simply an LSSS $\mathcal{S} = (M, \mathsf{label})$ over $\mathbb{F}$. To share a secret $v \in V$, the dealer samples uniformly at random a vector $\boldsymbol{r}_v \in V^{t+1}$ such that its first entry is v, and sends to player P_i each row of $M \cdot \boldsymbol{r}_v \in V^m$ owned by this player. Privacy properties are preserved. To reconstruct, a set of parties A with $|A| > t$ uses the reconstruction vector $\boldsymbol{e}_A$ as $\boldsymbol{e}_A^\intercal(M_A \boldsymbol{r}_v) = v$.*

As before, given $v \in V$ we use the notation $[\![v, \boldsymbol{r}_v]\!]_V$, or simply $[\![v]\!]_V$, to denote the vector in V^m of shares of v.

2.3 LSS Homomorphisms

Let U and V be two finite-dimensional $\mathbb{F}$-vector spaces, and let $\phi : V \to U$ be a vector-space homomorphism. According to the definition in Section 2.2, any given LSSS $\mathcal{S} = (M, \mathsf{label})$ over $\mathbb{F}$ can be seen as an LSSS over V or over U. However, the fact that there is a vector-space homomorphism from V to U implies that, for any $v \in V$, the parties can locally get $[\![\phi(v)]\!]_U$ from $[\![v]\!]_V$. We formalize this below.

Definition 4. *Let U and V be two finite-dimensional $\mathbb{F}$-vector spaces, and let $\phi : V \to U$ be a vector-space homomorphism. Let $\mathcal{S} = (M, \mathsf{label})$ be an LSSS over V. We say that the pair $(\mathcal{S}, \phi)$ is a linear secret-sharing homomorphism.*

The following simple proposition illustrates the value of considering LSS homomorphisms.

Proposition 1. *Let U and V be two finite-dimensional $\mathbb{F}$-vector spaces, and let $(\mathcal{S}, \phi)$ be a LSS homomorphism from U to V. Given $v \in V$ and $[\![v, \boldsymbol{r}_v]\!]_V$, applying ϕ to each share leads to $[\![\phi(v), \phi(\boldsymbol{r}_v)]\!]_U$.*[2]

Proof. Simply notice that $\phi\left([\![v, \boldsymbol{r}_v]\!]_V\right) = \phi(M\boldsymbol{r}_v) = M\phi(\boldsymbol{r}_v) = [\![\phi(v), \phi(\boldsymbol{r}_v)]\!]_U$. □

2.4 LSSS with Bilinear Maps

In Section 2.3 we saw how the parties could locally convert from sharings in one vector space to another vector space, provided there is a linear transformation between the two. The goal of this section is to extend this to the case of bilinear

[2] We extend the definition of ϕ to operate on vectors over V pointwise.

maps. More precisely, let U, V, W be $\mathbb{F}$-vector spaces of dimension d,[3] and let $\mathcal{S} = (M, \mathsf{label})$ be an LSSS over $\mathbb{F}$. From Section 2.2, $\mathcal{S}$ is also an LSSS over U, V and W. Let $\phi : U \times V \to W$ be a bilinear map, that is, the functions $\phi(\cdot, v)$ for $v \in V$ and $\phi(u, \cdot)$ for $u \in U$ are linear. In this section we show how the parties can obtain $[\![\phi(u,v)]\!]_W$ from $[\![u]\!]_U$ and $[\![v]\!]_V$, for $u \in U$ and $v \in V$.

Unlike the case of a linear transformation, this operation requires communication among the parties. Intuitively, this is achieved by using a generalization of "multiplication triples" to the context of bilinear maps. At a high level, the parties can preprocess "bilinear triples" $([\![\alpha]\!]_U, [\![\beta]\!]_V, [\![\phi(\alpha,\beta)]\!]_W)$ where $\alpha \in U$ and $\beta \in V$ are uniformly random, open $\delta = u - \alpha$ and $\epsilon = v - \beta$, and compute $[\![\phi(u,v)]\!]_W$ as

$$[\![\phi(\delta + \alpha, \epsilon + \beta)]\!]_W = \phi(\delta, \epsilon) + \phi(\delta, [\![\beta]\!]_V) + \phi([\![\alpha]\!]_U, \epsilon) + [\![\phi(\alpha,\beta)]\!]_W .$$

We formalize this intuition below where we describe the protocol in detail. For the protocol we assume a functionality $\mathcal{F}_{\mathsf{OuterProd}}$ that produces random shares $[\![a_1]\!], \ldots, [\![a_d]\!], [\![b_1]\!], \ldots, [\![b_d]\!]$ over $\mathbb{F}$, together with $[\![a_i b_j]\!]$ for $i, j \in \{1, \ldots, d\}$. This is used to produce the "bilinear triples" sketched above. Also, in the protocol below we assume that $\{u_1, \ldots, u_d\}$ is a basis for U and that $\{v_1, \ldots, v_d\}$ is a basis for V.

Protocol Π_{bilinear}

Inputs: $[\![u]\!]_U$ and $[\![v]\!]_V$.
Output: $[\![w]\!]_W$ where $w = \phi(u, v) \in W$.

OFFLINE PHASE

1. The parties call $\left(\{[\![a_i]\!]\}_{i=1}^d, \{[\![b_i]\!]\}_{i=1}^d, \{[\![a_i b_j]\!]\}_{i,j=1}^d\right) \leftarrow \mathcal{F}_{\mathsf{OuterProd}}$.
2. The parties use the LSS homomorphisms $x \mapsto x \cdot u_i$ and $x \mapsto x \cdot v_i$ to locally compute $[\![\alpha]\!]_U = \sum_{i=1}^d [\![a_i]\!] \cdot u_i$ and $[\![\beta]\!]_V = \sum_{i=1}^d [\![b_i]\!] \cdot v_i$, respectively.
3. The parties compute $[\![\phi(a_i u_i, b_j v_j)]\!]_W \leftarrow [\![a_i b_j]\!] \cdot \phi(u_i, v_j)$ using the LSS homomorphisms $x \mapsto x \cdot \phi(u_i, v_j)$.
4. The parties compute locally $[\![\phi(\alpha,\beta)]\!]_W = \sum_{i,j=1}^d [\![\phi(a_i u_i, b_j v_j)]\!]_W$.

ONLINE PHASE

1. The parties open $\delta \leftarrow [\![u]\!]_U - [\![\alpha]\!]_U$ and $\epsilon \leftarrow [\![v]\!]_V - [\![\beta]\!]_V$
2. The parties use the LSS homomorphism $\phi(\delta, \cdot)$ to compute $[\![\phi(\delta,\beta)]\!]_W \leftarrow \phi(\delta, [\![\beta]\!]_V)$, and similarly they use the LSS homomorphism $\phi(\cdot, \epsilon)$ to compute $[\![\phi(\alpha,\epsilon)]\!]_W \leftarrow \phi([\![\alpha]\!]_U, \epsilon)$.
3. The parties compute locally and output $[\![\phi(u,v)]\!]_W = \phi(\delta,\epsilon) + [\![\phi(\delta,\beta)]\!]_W + [\![\phi(\alpha,\epsilon)]\!]_W + [\![\phi(\alpha,\beta)]\!]_W$.

[3] It is not necessary for these spaces to have the same dimension, but we assume this for simplicity in the notation.

3 Instantiations

In the previous section we developed a theory for LSS homomorphisms and secure computation for bilinear maps based on an arbitrary linear secret sharing scheme and an arbitrary linear transformation between vector spaces. In this section we instantiate the vector spaces with elliptic curves, and the bilinear maps with cryptographic pairings, which allows us to securely evaluate cryptographic primitives based on elliptic curves, as we will show in subsequent sections. Additionally, we instantiate the secret sharing scheme with three types of sharings: additive secret sharing with MACs, used in dishonest-majority protocols [16], Shamir secret sharing, used for honest majority protocols [17], and replicated secret sharing for the specific case of 3 parties with an honest majority [1].

3.1 Instantiating the Vector Spaces with Elliptic Curves

Let $\mathbb{G}$ be an elliptic curve group of order a prime p, which in particular means that $\mathbb{G}$ is an $\mathbb{F}$-vector space, and let G be a generator of $\mathbb{G}$. Consider the isomorphism $\phi : \mathbb{F} \to \mathbb{G}$ given by $x \mapsto x \cdot G$. Let $\mathcal{S} = (M, \mathsf{label})$ be an LSSS over $\mathbb{F}$. Given what we have seen so far, $\mathcal{S}$ can be seen as an LSSS over $\mathbb{G}$. To secret-share a curve point $P \in \mathbb{G}$, the dealer samples random points $(P_1, \ldots, P_t)$, computes $(Q_1, \ldots, Q_m)^\intercal = M \cdot (P, P_1, \ldots, P_t)^\intercal \in \mathbb{G}^m$, and sends Q_i to party $P_{\mathsf{label}(i)}$. Furthermore, if $s \in \mathbb{F}$ is secret shared as $[\![s]\!]$, the LSS homomorphism property applied to ϕ implies that each party can locally multiply its share by the generator G to obtain $[\![s \cdot G]\!]_{\mathbb{G}}$.

Now, consider the scalar multiplication map $f : \mathbb{F} \times \mathbb{G} \to \mathbb{G}$ given by $f : x, P \mapsto x \cdot P$. Using Π_{Bilinear} with f we can obtain the protocol $\Pi_{\mathsf{ScalarMul}}$, described below, which computes a scalar multiplication between a scalar and point when both scalar and point are secret-shared. We remark that this protocol was presented in [36] and as such our presentation here can be considered as illustrating that Π_{Bilinear} generalizes the techniques in their work. Also, when the dimension d is equal to 1, the functionality $\mathcal{F}_{\mathsf{OuterProd}}$ becomes a multiplication triple functionality $\mathcal{F}_{\mathsf{MulTriple}}$ that produces $([\![a]\!], [\![b]\!], [\![a \cdot b]\!])$, where $a, b \in \mathbb{F}$ are uniformly random.

Protocol $\Pi_{\mathsf{ScalarMul}}$

Inputs: $[\![x]\!]$ and $[\![P]\!]_{\mathbb{G}}$
Outputs: $[\![x \cdot P]\!]_{\mathbb{G}}$

OFFLINE PHASE

1. Parties call $([\![a]\!], [\![b]\!], [\![a \cdot b]\!]) \leftarrow \mathcal{F}_{\mathsf{MulTriple}}$.
2. Parties use the LSS homomorphism $x \mapsto x \cdot G$ for a generator G of $\mathbb{G}$ to compute $[\![B]\!]_{\mathbb{G}} = [\![b]\!] \cdot G$ and $[\![C]\!] = [\![a \cdot b]\!] \cdot G$.

ONLINE PHASE

1. Parties open $d \leftarrow [\![x]\!] - [\![a]\!]$ and $Q \leftarrow [\![P]\!]_{\mathbb{G}} - [\![B]\!]_{\mathbb{G}}$.
2. Using the LSS homomorphism, parties compute $[\![E]\!]_{\mathbb{G}} = [\![a]\!] \cdot Q$ and $[\![F]\!]_{\mathbb{G}} = d \cdot [\![B]\!]_{\mathbb{G}}$.
3. Parties compute locally $[\![x \cdot P]\!]_{\mathbb{G}} = [\![E]\!]_{\mathbb{G}} + [\![F]\!]_{\mathbb{G}} + d \cdot Q + [\![C]\!]_{\mathbb{G}}$.

Bilinear Pairings. Consider $\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T$ elliptic curve groups of order a prime p. As usual in the field of pairing-based cryptography, we use additive notation for the groups $\mathbb{G}_1, \mathbb{G}_2$, and multiplicative notation for $\mathbb{G}_T$. We denote by $0_{\mathbb{G}_1}, 0_{\mathbb{G}_2}$ and $1_{\mathbb{G}_T}$ the identities of $\mathbb{G}_1, \mathbb{G}_2$ and $\mathbb{G}_T$, respectively. Consider a pairing $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$ satisfying:

1. For all $G \in \mathbb{G}_1, H \in \mathbb{G}_2$ and $a, b \in \mathbb{F}$, $e(aG, bH) = e(G, H)^{ab}$.
2. For $P_1 \in \mathbb{G}_1$, $P_2 \in \mathbb{G}_2$ with $P_1 \neq 0$, $P_2 \neq 0$, $e(P_1, P_2) \neq 1$.
3. The map e can be computed efficiently.

This notation will be used for the rest of the paper. In the context of Section 2, the groups $\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T$ can be viewed as $\mathbb{F}$-vector spaces of dimension 1, so we can apply the techniques presented there to compute $[\![e(P_1, P_2)]\!]_{\mathbb{G}_T}$ from $[\![P_1]\!]_{\mathbb{G}_1}$ and $[\![P_2]\!]_{\mathbb{G}_2}$. We summarize the resulting protocol below. We let G_1 and G_2 be some generators of $\mathbb{G}_1$ and $\mathbb{G}_2$, respectively.

Protocol Π_{pairing}

Inputs: $[\![P_1]\!]_{\mathbb{G}_1}$ and $[\![P_2]\!]_{\mathbb{G}_2}$.
Output: $[\![e(P_1, P_2)]\!]_{\mathbb{G}_T}$.

OFFLINE PHASE

1. The parties call $([\![a]\!], [\![b]\!], [\![a \cdot b]\!]) \leftarrow \mathcal{F}_{\mathsf{MulTriple}}$.
2. The parties use the LSS homomorphisms $x \mapsto x \cdot G_1$ and $x \mapsto x \cdot G_2$ to locally compute $[\![Q_1]\!]_{\mathbb{G}_1} = [\![a]\!] \cdot G_1$ and $[\![Q_2]\!]_{\mathbb{G}_2} = [\![b]\!] \cdot G_2$, respectively.
3. Using the LSS homomorphism $x \mapsto e(G_1, G_2)^x$, the parties compute $[\![e(Q_1, Q_2)]\!] = [\![e(a \cdot G_1, b \cdot G_2)]\!]_{\mathbb{G}_T} \leftarrow e(G_1, G_2)^{[\![ab]\!]}$

ONLINE PHASE

1. The parties open $D_1 \leftarrow [\![P_1]\!]_{\mathbb{G}_1} - [\![Q_1]\!]_{\mathbb{G}_1}$ and $D_2 \leftarrow [\![P_2]\!]_{\mathbb{G}_2} - [\![Q_2]\!]_{\mathbb{G}_2}$
2. The parties use the LSS homomorphism $e(Q_1, \cdot)$ to compute $[\![e(D_1, Q_2)]\!]_{\mathbb{G}_T} \leftarrow e(D_1, [\![Q_2]\!]_{\mathbb{G}_1})$, and similarly they use the LSS homomorphism $e(\cdot, D_2)$ to compute $[\![e(Q_1, D_2)]\!]_{\mathbb{G}_T} \leftarrow e([\![Q_1]\!]_{\mathbb{G}_1}, D_2)$.
3. The parties compute locally and output $[\![e(P_1, P_2)]\!]_{\mathbb{G}_T} = e(D_1, D_2) \cdot [\![e(D_1, Q_2)]\!]_{\mathbb{G}_T} \cdot [\![e(Q_1, D_2)]\!]_{\mathbb{G}_T} \cdot [\![e(Q_1, Q_2)]\!]_{\mathbb{G}_T}$.

3.2 Instantiating the Secret Sharing Schemes

In what follows we instantiate the secret sharing scheme $\mathcal{S}$ with three popular schemes: Additive SS, replicated SS and Shamir SS.

Additive SS. In this scheme each party P_i gets a uniformly random value $r_i \in \mathbb{F}$ subject to $\sum_{i=1}^{n} r_i = s$, where $s \in \mathbb{F}$ is the secret. This scheme is $(n-1, n)$-secure. Let us denote additive secret sharing of s by $[\![s]\!]^{\mathsf{add}}$, and abusing notation, we write $[\![s]\!]^{\mathsf{add}} = (r_1, \ldots, r_n)$, where each r_i is the share of party P_i. Given an elliptic curve group $\mathbb{G}$ of order p, having G as generator, the parties can obtain shares of $s \cdot G$ by locally multiplying the generator G by their share r_i; that is, $[\![s \cdot G]\!]^{\mathsf{add}} = (r_1 \cdot G, \ldots, r_n \cdot G)$.

In the scheme above, at reconstruction time, a maliciously corrupt party can lie about its share, causing the reconstructed value to be incorrect. To help solve this issue, actively secure protocols in the dishonest majority share a secret s as $[\![s]\!]^{\mathsf{add}}$, together with $[\![r \cdot s]\!]^{\mathsf{add}}$, where r is a *global* uniformly random value that is also shared as $[\![r]\!]^{\mathsf{add}}$. We denote this by $[\![s]\!]^{\mathsf{add}*}$. At reconstruction time, the adversary may open $[\![s]\!]^{\mathsf{add}}$ to $s + \delta$ where δ is some error known to the adversary. To ensure that $\delta = 0$ (so the correct value is opened), the parties compute $(s + \delta)\,[\![r]\!]^{\mathsf{add}} - [\![r \cdot s]\!]^{\mathsf{add}}$, open this value, and check it equals 0. It can be easily shown that, if $\delta \neq 0$, this check passes with probability at most $1/|\mathbb{F}|$.

The same check can be performed over $\mathbb{G}$: The sharings $[\![s \cdot G]\!]^{\mathsf{add}}_{\mathbb{G}}$ are accompanied by $[\![r \cdot s \cdot G]\!]^{\mathsf{add}}_{\mathbb{G}}$, where r is a *global* uniformly random value that is also shared as $[\![r]\!]^{\mathsf{add}}$. At reconstruction time $[\![s \cdot G]\!]^{\mathsf{add}}_{\mathbb{G}}$ can be opened to $(s + \delta) \cdot G$, and to ensure $\delta = 0$ the parties open $[\![r]\!]^{\mathsf{add}}_{\mathbb{G}} \cdot (s + \delta) \cdot G - [\![r \cdot s \cdot G]\!]^{\mathsf{add}}_{\mathbb{G}}$ and check that this point is the identity. It is easy to see that, like in the case over $\mathbb{F}$, the check passes with probability at most $1/|\mathbb{F}|$ if $\delta \neq 0$. We denote this "robust" sharing of $s \cdot G$ by $[\![s \cdot G]\!]^{\mathsf{add}*}_{\mathbb{G}}$.

Shamir SS. Let $\mathbb{F}_{\leq d}[x]$ be the ring of polynomials over $\mathbb{F}$ of degree at most d. Consider a setting with n parties, and let $0 < t < n$. In this scheme each party P_i gets $f(i)$ where $f(x) \in_R \mathbb{F}_{\leq t}[x]$. We denote $[\![s]\!]^{\mathsf{shm}}_{\mathbb{F}} = (f(1), \ldots, f(n))$.

Let $2t+1 = n$, and let U, V, W be $\mathbb{F}$-vector spaces of dimension d,[4] and let $\phi : U \times V \to W$ be a bilinear map. Now, consider shared values $[\![x_1]\!]^{\mathsf{shm}}_U, \ldots, [\![x_L]\!]^{\mathsf{shm}}_U$ and $[\![y_1]\!]^{\mathsf{shm}}_V, \ldots, [\![y_L]\!]^{\mathsf{shm}}_V$. A protocol $\Pi^{\mathsf{shm}}_{\mathsf{DotProd}}$ that, on input these shares, produces $[\![z]\!]^{t}_W$, where $z = \sum_{\ell=1}^{L} \phi(x_\ell y_\ell)$, can be instantiated by using L calls to the Π_{bilinear} protocol, although this incurs in a communication complexity that is at least proportional to L. Instead, we consider a different protocol that produces an output that may be incorrect by an additive amount $\delta \in W$ known to the adversary, but has a communication complexity that is independent of L. This is explained in detail in Section A.2 in the appendix. The resulting protocol,

[4] As in Section 2, the condition that all three spaces have the same dimension is not necessary.

which can be seen as a generalization of the multiplication protocol in [23], has a communication complexity of $d \cdot \log(|\mathbb{F}|) \cdot 5.5 \cdot n$ bits.

Replicated SS. This is a $(1, 2)$-secure LSSS for 3 parties. In this scheme each party P_i gets (r_i, r_{i+1}), where the sub-indexes wrap modulo 3, and $s = r_1 + r_2 + r_3$, where $s \in \mathbb{F}$ is the secret. Like above, let U, V, W be $\mathbb{F}$-vector spaces of dimension d, and let $\phi : U \times V \to W$ be a bilinear map, and consider shared values $[\![x_1]\!]_U^{\mathsf{rep}}, \ldots, [\![x_L]\!]_U^{\mathsf{rep}}$ and $[\![y_1]\!]_V^{\mathsf{rep}}, \ldots, [\![y_L]\!]_V^{\mathsf{rep}}$. Although a protocol $\Pi_{\mathsf{DotProd}}^{\mathsf{rep}}$ that, on input these shares, produces $[\![z]\!]_W^{\mathsf{rep}}$, where $z = \sum_{\ell=1}^{L} \phi(x_\ell y_\ell)$, can be instantiated by using L calls to the Π_{bilinear} protocol, this would again incur in a communication complexity that is at least proportional to L. Instead, we consider a different protocol that produces an output that may be incorrect by an additive amount $\delta \in W$ known to the adversary, but has a communication complexity that is independent of L. This is explained in detail in Section A.3 in the appendix. The resulting protocol, which can be seen as a generalization of the multiplication protocol in [1], has a communication complexity of $d \cdot \log(|\mathbb{F}|) \cdot 3$ bits.

Remark 1. The protocols $\Pi_{\mathsf{DotProd}}^{\mathsf{shm}}$ and $\Pi_{\mathsf{DotProd}}^{\mathsf{rep}}$ above instantiate a functionality that we denote by $\mathcal{F}_{\mathsf{DotProd}}$ that, on input $([\![x_i]\!]_U^{\mathsf{shm/rep}})_{i=1}^{L}$ and $([\![y_i]\!]_V^{\mathsf{shm/rep}})_{i=1}^{L}$, produces $[\![z + \delta]\!]_W^{\mathsf{shm/rep}}$, where $z = \sum_{\ell=1}^{L} \phi(x_\ell y_\ell)$ and $\delta \in W$ is an error known by the adversary, with a communication that is independent of L. In the case that the error δ is not allowed, we denote the corresponding functionality by $\mathcal{F}_{\mathsf{DotProd}*}$, and it can be instantiated by feeding the protocols above through the compiler from [11], at a cost of increasing by only twice the communication complexity, in the amortized setting.[5]

4 Secure Computation over Elliptic Curves

The previous section showed a "toolbox" for performing computation over elliptic curves: We can secret-share elliptic curve points, perform addition locally, and securely compute the analogue of elliptic curve multiplication, which are pairings. This can potentially enable a new range of applications that require secure computation over elliptic curves. However, we typically use elliptic curve computation not to operate on curve points, but on bit-strings that are mapped to and *from* these curves. Consider for example the following use-case: Parties hold $[\![m]\!]_{\mathbb{F}}$ and wish to encrypt it using, e.g., El-Gamal. Using an LSS homomorphism on $[\![m]\!]$ would effectively encode m in the exponent, and then we could use secure computation over elliptic curves to compute the encryption of m.

The above works for encryption. However, what if the goal is to decrypt? That is, given some ciphertext and a secret-shared private key, the parties wish to compute shares of the corresponding plaintext. Using secure computation over

[5] Although the results from [11] are set for the case in which $U = V = W = \mathbb{F}$, it is not hard to extend them to the more general case.

the elliptic curve, the parties can compute shares of $m \cdot G$. Now, obtaining shares of m from $[\![m \cdot G]\!]$ is not that straightforward as it involves solving a discrete-log problem. If $m \cdot G$ was in the clear and m was known to be bounded, then m could be recovered by exhaustive search. However, performing exhaustive search in MPC is not feasible, so it is not obvious how to decode $[\![m]\!]$ from $[\![m \cdot G]\!]$.

The issue above arises from the fact that the encoding of $[\![m]\!]$ was done using the LSS homomorphism $x \mapsto x \cdot G$, which is highly efficient due to its linearity, but has a "one-wayness" to it, making it very hard to decode. In the following, we show a different way of encoding a shared field element $[\![m]\!]$ in such a way that, although the encoding itself is interactive (and therefore less efficient than the LSS homomorphism encoding described above), the decoding process is practically efficient. This enables a seamless interplay between traditional secure computation over $\mathbb{F}$, and secure computation over an elliptic curve group as defined here.

In the following, as well as in the rest of the paper, we will rely on several secure computation functionalities, some of them which have been introduced already. We list them here in brief. Also, for a functionality/protocol $\mathcal{F}_{\mathsf{abc}}/\Pi_{\mathsf{abc}}$, we denote by $\mathcal{C}_{\mathsf{abc}}$ its total communication cost, in bits.

- $\mathcal{F}_{\mathsf{MulTriple}}$ outputs a triple $([\![a]\!], [\![b]\!], [\![c]\!])$ where $c = ab$. Note that this is a special case of $\mathcal{F}_{\mathsf{OuterProd}}$ for $d = 1$.
- $\mathcal{F}_{\mathsf{DotProd}}$ takes as input $([\![x_i]\!])_{i=1}^{L}$ and $([\![y_i]\!])_{i=1}^{L}$, and produces $[\![z + \delta]\!]$, where $z = \sum_{\ell=1}^{L} \phi(x_\ell y_\ell)$ and $\delta \in \mathbb{F}$ is an error known by the adversary. $\mathcal{F}_{\mathsf{DotProd}*}$ is similar, except it does not accept such error.
- $\mathcal{F}_{\mathsf{Mul}}$ takes two inputs $[\![x]\!]$ and $[\![y]\!]$, and outputs $[\![w]\!]$ where $w = xy$. $\mathcal{F}_{\mathsf{Mul}}$ is a particular case of $\mathcal{F}_{\mathsf{DotProd}*}$ for $L = 1$.
- $\mathcal{F}_{\mathsf{Rand}}(K)$ outputs $[\![x]\!]$ where $x \in K$, where K is a $\mathbb{F}$-vector space. Notice that it is enough to have a functionality which samples a secret-shared field element; to get a secret point, parties can locally apply an appropriate LSS homomorphism to obtain a secret-shared group element.
- $\mathcal{F}_{\mathsf{Coin}}(K)$ outputs a uniformly random $s \in K$ to all parties.

4.1 Secure Encoding and Decoding

We now show how to map secret-shared messages into curve points, and back, in the presence of an active adversary and an honest majority. Consider the following commonly used injective encoding for encoding bit-strings into points on the curve $\mathbb{G}$ over $\mathbb{F}$ (see [20]): To encode a message $m \in \{0,1\}^\ell$, with $\ell \leq (1/2 - \epsilon) \log_2 p$ for a fixed $\epsilon \in (0, 1/2)$, pick a random integer $x \in [0, p-1]$ such that $m = x \mod 2^\ell$. If x is a valid curve-point for $\mathbb{G}$, then output (x, y), and otherwise pick a new random x and start over. We denote this encoding by En and its inverse as De (notice that De simple discards y and returns $x \mod 2^\ell$).

Our aim now is to implement $(\mathsf{En}, \mathsf{De})$ securely; that is, we wish to compute $[\![\mathsf{En}(x)]\!]$ given $[\![x]\!]$ with $x \in \{0,1\}^\ell$, and $[\![\mathsf{De}(X)]\!]$ given $[\![X]\!]_{\mathbb{G}}$ with $\mathsf{En}(m) = X \in \mathbb{G}$ for some m. For this we will use two protocols, described in brief here and presented in detail in Appendix D. The first protocol is Π_{IsSqr}, which takes as

input a secret-shared value $[\![x]\!]$ and outputs 1 if x is a square, and 0 otherwise. That is, if Π_{IsSqr} outputs 1, then there exists a value y such that $x^2 = y \mod p$. The other protocol is Π_{Sqrt} which, on input a square $[\![x]\!]$, outputs $[\![y]\!]$ satisfying $y = x^2 \mod p$.

In the following, we assume that the curve is given as $y^2 = x^3 + ax + b$ where a and b are constants.

Decoding. We begin with decoding. Given a secret-sharing $[\![\mathsf{En}(m)]\!]_{\mathbb{G}}$ where $\mathsf{En}(m) = (x, y)$ and $m \equiv x \mod 2^\ell$, the goal is to obtain $[\![m]\!]$. Besides $[\![\mathsf{En}(m)]\!]_{\mathbb{G}}$, we assume that we also have access to a secret-sharing of the upper $\ell - \log_2 p$ bits of x and we denote this value as $[\![r]\!]$. Write $[\![z]\!]_{\mathbb{G}} = [\![\mathsf{En}(m)]\!]_{\mathbb{G}}$ and let x_i, resp. y_i be the values that comprise the i'th party's share of z. To decode z, each party first re-shares the x_i and y_i they hold, after which everyone computes the point addition formula over all the coordinates. In a nutshell, this is the same idea used when decomposing a number into bits. In this scenario, parties mask the value they want to bit-decompose and then compute a binary adder to unmask each bit.

Protocol Π_{Decode}

Inputs: $[\![X]\!]_{\mathbb{G}}$, $[\![r]\!]$ where r was the randomness added during encoding.
Outputs: $[\![m]\!]$ the encoded message, secret-shared over the basefield.

1. Each party P_i parses their share of $[\![X]\!]_{\mathbb{G}}$ as the pair (x_i, y_i) and secret-shares $[\![x_i]\!]$, $[\![y_i]\!]$ towards the other parties.
2. Parties apply a parity check matrix to check that the reshared values are consistent (see Appendix A.3 for details).
3. For $j = 2, \ldots, t+1$ where t is the number of corrupt parties, compute the curve addition of the shares over the secret-shared coordinates:
   (a) Invoke $[\![a]\!] = \mathcal{F}_{\mathsf{Rand}}(\mathbb{F})$.
   (b) $[\![z]\!] \leftarrow \mathcal{F}_{\mathsf{Mul}}([\![x_j - x_{j-1}]\!], [\![a]\!])$ and open z.
   (c) Compute $[\![d]\!] = [\![(x_j - x_{j-1})^{-1}]\!] = z^{-1}[\![a]\!]$, $[\![\lambda]\!] = \mathcal{F}_{\mathsf{Mul}}([\![y_j - y_{j-1}]\!], [\![d]\!])$ and finally $[\![\lambda^2]\!] = \mathcal{F}_{\mathsf{Mul}}([\![\lambda]\!], [\![\lambda]\!])$.
   (d) Compute $[\![x']\!] = [\![\lambda^2]\!] - [\![x_j]\!] - [\![x_{j-1}]\!]$.
   (e) Compute $[\![y'']\!] = \mathcal{F}_{\mathsf{Mul}}([\![\lambda]\!], [\![x_j - x']\!])$ and $[\![y']\!] = [\![y'']\!] - [\![y_j]\!]$.
   (f) Set $[\![x_j]\!] = [\![x']\!]$ and $[\![y_j]\!] = [\![y']\!]$.
4. Output $[\![x_{t+1}]\!] - [\![r]\!]$.

Protocol Π_{Decode} computes the injective encoding with complexity $(t+1)(\mathcal{C}_{\mathsf{Rand}}(1) + \mathcal{C}_{\mathsf{Mul}}(4) + \mathcal{C}_{\mathsf{Open}}(1)) + \mathcal{C}_{\mathsf{Share}}(n) + \mathcal{C}_{\mathsf{Check}}(n)$.

Lemma 1. *Protocol Π_{Decode} securely outputs the lower ℓ bits of $[\![X]\!]_{\mathbb{G}}$.*

Proof. Let $X_i = (x_i, y_i)$ be the i'th party's share of $X = (x, y)$. Notice that X can be reconstructed as a linear combination of the X_i's; in particular, $X = \sum_{i=1}^{t+1} X_i$ (we omit constants in this linear combination for the sake of simplicity). This linear combination is computed in step 3 in the protocol, so, at step 3.f, parties

hold shares of the coordinates of X, secret-shared over the base field. Finally, $[\![x]\!] - [\![r]\!]$ removes the randomness located in the upper $\log_2 p - \ell$ bits of x. Step 1 potentially poses a problem, as a corrupt party may secret-share an incorrect value. However, the parity check applied in step 2 ensures this cannot happen, as the adversary can only modify at most t shares.

Encoding. To encode a value $x \in \mathbb{F}$, recall that we first need to add a bit of randomness to it, in order to have a chance at hitting a valid x-coordinate for our curve. Let ℓ be an upper bound on the size of x, i.e., $x \leq 2^\ell$. We first consider a straightforward, but ultimately insecure, approach utilizing $\mathcal{F}_{\mathsf{Coin}}$: Parties use $\mathcal{F}_{\mathsf{Coin}}$ to sample a random value $r < p$ such that its lower ℓ bits are 0. Parties then call $\Pi_{\mathsf{IsSqr}}([\![x]\!] + r)$, and restart the process (i.e., go back and pick another r) if this protocol outputs 0. However this fails to be secure. Indeed, if x is of low entropy, then revealing whether or not $[\![x]\!] + r$ is a square, reveals information about x itself (in particular, the adversary can rule out values x' for which $x' + r$ *is* a square).

We must thus resort to fancier machinations that allows us to sample an appropriate r without revealing it. Luckily, sampling a random value where its lower bits are zero has been used before—in particular in connection with secure truncation protocols (see e.g., [14]). We thus assume a functionality $\mathcal{F}_{\mathsf{sRand}}$ which outputs a secret-shared r suitable for our purposes (see short description in Appendix C). The final thing we require is a tuple $([\![R]\!]_{\mathbb{G}}, [\![r_x]\!], [\![r_y]\!])$ where $R = (r_x, r_y)$. Such a tuple can be generated by sampling a random $[\![R]\!]_{\mathbb{G}}$ and then using step 2 in Π_{Decode} to obtain $[\![r_x]\!]$ and $[\![r_y]\!]$.

Protocol Π_{Encode}

Inputs: $[\![m]\!]$ the message to be encoded.
Outputs: $[\![\mathsf{En}(m)]\!]_{\mathbb{G}}$, $[\![r]\!]$.

1. Sample $[\![r]\!] = \mathcal{F}_{\mathsf{sRand}}$ and compute $[\![x]\!] = [\![m]\!] + [\![r]\!]$.
2. Call $\Pi_{\mathsf{IsSqr}}([\![x]\!])$. If the return value is 0, go back to the previous step.
3. Call $[\![y]\!] = \Pi_{\mathsf{Sqrt}}([\![x^3]\!] + [\![x]\!]\, a + b)$. Note that parties now have $[\![x]\!]$, $[\![y]\!]$ which are secret-sharings of $\mathsf{En}(m)$ in the field.
4. Parties then compute the curve addition formula between the points $([\![x]\!], [\![y]\!])$ and $([\![r_x]\!], [\![r_y]\!])$. Let $([\![z_x]\!], [\![z_y]\!])$ be the result.
5. $[\![z_x]\!]$ and $[\![z_y]\!]$ is opened. Write $Z = (z_x, z_y)$.
6. Output $[\![\mathsf{En}(m)]\!]_{\mathbb{G}} = [\![X]\!]_{\mathbb{G}} = Z - [\![R]\!]_{\mathbb{G}}$ and $[\![r]\!]$.

Protocol Π_{Encode} computes the injective encoding of m with complexity $\mathcal{C}_{\mathsf{Encode}} \leq \mathcal{C}_{\mathsf{sRand}}(3) + \mathcal{C}_{\mathsf{IsSqr}}(3) + \mathcal{C}_{\mathsf{Sqrt}} + 2\mathcal{C}_{\mathsf{Open}}(1) + \mathcal{C}_{\mathsf{Rand}}(1) + \mathcal{C}_{\mathsf{Mul}}(4)$. Its security comes from the fact that, at the end of step 5, parties hold $Z = X + R$, and since R is random, nothing is revealed about X. Notice that an exact cost is not possible, since step 2 may repeat. However, [20] proves that a suitable r is found in expected 3 iterations.

5 Digital Signatures in MPC

In this section we show how the techniques presented in the previous sections can be used to securely sign and verify messages that are secret shared, using keys that are similarly secret-shared. More precisely, we present here three protocols: First, a key generation protocol Π_{Keygen} for generating $(\mathsf{pk}, [\![\mathsf{sk}]\!])$ securely where pk is a public key and $[\![\mathsf{sk}]\!]$ a secret-shared private key. Second, a signing protocol Π_{Sign} protocol that on input a secret shared message $[\![m]\!]$ and $[\![sk]\!]$ output from Π_{Keygen} outputs $[\![\sigma]\!]$ where σ is a signature on m under sk. Finally, we present a verification protocol Π_{Verify} which on input $[\![m]\!]$, $[\![\sigma]\!]$ and pk outputs $[\![b]\!]$ where b is a value indicating whether or not σ is a valid signature on m under the private key corresponding to the public key pk.

We choose to the PS signature scheme [33] as our starting point. The primary reason for choosing the PS scheme is that signatures are short and independent of the message length, and that messages do not need to be hashed prior to signing. Interestingly, computing PS signatures securely leads to a number of optimizations that are made possible since e.g., the secret key is not known by any party.

5.1 The PS Signature Scheme

The following signature scheme due to Pointcheval and Sanders [33] (henceforth PS) signs a vector of messages $\mathbf{m} \in \mathbb{F}^r$ as follows (we present the multi-message variant here):

- $\mathsf{Setup}(1^\lambda)$: Output $pp \leftarrow (p, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e)$, a type-3 pairing.
- $\mathsf{Keygen}(pp)$: Select random $H \leftarrow \mathbb{G}_2$ and $(x, y_1, \ldots, y_r) \leftarrow \mathbb{F}^{r+1}$. Compute $(X, Y_1, \ldots, Y_r) = (xH, y_1H, \ldots, y_rH)$ set $\mathsf{sk} = (x, y_1, \ldots, y_r)$ and $\mathsf{pk} = (H, X, Y_1, \ldots, Y_r)$.
- $\mathsf{Sign}(\mathsf{sk}, \mathbf{m})$: Select random $G \leftarrow \mathbb{G}_1 \setminus \{0\}$ and output the signature $\sigma = (G, (x + \sum_{i=1}^{r} m_i y_i) \cdot G)$.
- $\mathsf{Verify}(\mathsf{pk}, \mathbf{m}, \sigma)$: Parse σ as (σ_1, σ_2). If $\sigma_1 \neq 0$ and $e(\sigma_1, X + \sum m_i Y_i) = e(\sigma_2, H)$ output 1. Otherwise output 0.

The remainder of this section will focus on how to instantiate the PS signature scheme securely.

5.2 Distributed PS Signatures

The Π_{Keygen} protocol presented below shows how to generate keys suitable for signing messages of r blocks. The protocol proceeds in a straight forward manner: parties invoke $\mathcal{F}_{\mathsf{Coin}}$ and $\mathcal{F}_{\mathsf{Rand}}$ a suitable number of times to generate the private key and then use an appropriate LSS homomorphism to compute the public key.

Protocol Π_{Keygen}

Inputs: $pp = (p, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e)$, r
Outputs: $(\mathsf{pk}, [\![\mathsf{sk}]\!])$

1. Parties invoke $\mathcal{F}_{\mathsf{Coin}}(\mathbb{G}_2)$ to obtain H, and invoke $\mathcal{F}_{\mathsf{Rand}}(\mathbb{F})$ a total of $r+1$ times to obtain $([\![x]\!], [\![y_1]\!], \ldots, [\![y_r]\!])$.
2. Let $\phi_2 : \mathbb{F} \to \mathbb{G}_2$ be LSS-homomorphism given by $\phi_2 : x \mapsto xH$. Using ϕ_2, compute $[\![X]\!]_{\mathbb{G}_2} = \phi_2([\![x]\!])$ and $[\![Y_i]\!]_{\mathbb{G}_2} = \phi_2([\![y_i]\!])$ for $i = 1, \ldots, r$.
3. Parties open $X \leftarrow [\![X]\!]_{\mathbb{G}_2}$ and $Y_i \leftarrow [\![y_i]\!]_{\mathbb{G}_2}$ for $i = 1, \ldots, r$. Output the pair $(\mathsf{pk}, [\![\mathsf{sk}]\!])$ where $\mathsf{pk} = (H, X, Y_1, \ldots, Y_r)$ and $[\![\mathsf{sk}]\!] = ([\![x]\!], [\![y_1]\!], \ldots, [\![y_r]\!])$.

The communication complexity of Π_{Keygen} is $\mathcal{C}_{\mathsf{Keygen}} = \mathcal{C}_{\mathsf{Coin}}(1) + \mathcal{C}_{\mathsf{Rand}}(r+1) + \mathcal{C}_{\mathsf{Open}}(r+1)$ field elements.

Next up is computing Sign on secret-shared inputs, a process which is straightforward, given the tools we have described so far: Parties only need to pick a random group element (which we do using $\mathcal{F}_{\mathsf{Coin}}$) and then call $\mathcal{F}_{\mathsf{DotProd}}$ to compute the second component of the signature.

Protocol Π_{Sign}

Inputs: $[\![\mathsf{sk}]\!] = ([\![x]\!], [\![y_1]\!], \ldots, [\![y_r]\!])$, $[\![\mathbf{m}]\!] = ([\![m_1]\!], \ldots, [\![m_r]\!])$
Outputs: $[\![\sigma]\!]$

1. Parties obtain $\sigma_1 \in_R \mathbb{G}_1$ by invoking $\mathcal{F}_{\mathsf{Coin}}(\mathbb{G}_1)$. If $\sigma_1 = 0$, repeat this step.
2. Parties invoke $[\![z]\!] \leftarrow \mathcal{F}_{\mathsf{DotProd}*}\left((([\![y_i]\!])_{i=1}^r, ([\![m_i]\!])_{i=1}^r\right)$ and then compute $[\![w]\!] = [\![x]\!] + [\![z]\!]$.
3. Parties use the LSS homomorphism $x \mapsto x \cdot \sigma_1$ to compute locally $[\![\sigma_2]\!]_{\mathbb{G}_1} \leftarrow \Pi_{\mathsf{ScalarMul}}([\![w]\!], \sigma_1)$.
4. Output $(\sigma_1, [\![\sigma_2]\!]_{\mathbb{G}_1})$.

Protocol Π_{Sign} produces a correct signature with communication complexity $\mathcal{C}_{\mathsf{Coin}}(1) + \mathcal{C}_{\mathsf{DotProd}*}(r)$. Observe that, if the secret-sharing scheme is instantiated either with Shamir or Replicated secret sharing, the communication complexity becomes independent of r.

Remark 2. If we replace $\mathcal{F}_{\mathsf{DotProd}*}$ above by $\mathcal{F}_{\mathsf{DotProd}}$, the effect would be that an adversary could add any point $\delta \in \mathbb{G}_1$ to the output, getting $(\sigma_1, [\![\sigma_2 + \delta]\!]_{\mathbb{G}_1})$, causing the signature to be wrong. This is actually acceptable in some settings, in particular, the one we consider in Section 6. Furthermore, as we saw in Section 3, this makes the protocol much more efficient.

Finally, we show verification. However, to permit an efficient instantiation we make the following very minor modification to Verify, which we dub Verify^*:

On successful signature verification, Verify^* outputs 1 and otherwise Verify^* outputs non-zero. Clearly Verify and Verify^* carry the same amount of information, however the latter is less strict on the value indicating a failure—in particular, it does not mandate that a failure is indicated by the output 0. (As we will see, the output will be an element of $\mathbb{G}_T$ for which a "0" does not exist.)

Protocol Π_{Verify^*}

Inputs: $\mathsf{pk} = (H, X, Y_1, \ldots, Y_r)$, $[\![\mathbf{m}]\!] = ([\![m_i]\!])_{i=1}^r$, $\sigma = (\sigma_1, [\![\sigma_2]\!]_{\mathbb{G}_1})$
Outputs: $[\![b]\!]_{\mathbb{G}_T} = [\![\mathsf{Verify}^*(\mathsf{pk}, \mathbf{m}, \sigma)]\!]$

1. If $\sigma_1 = 0$ then output $[\![\mu]\!]_{\mathbb{G}_T} \leftarrow \mathcal{F}_{\mathsf{Rand}}(\mathbb{G}_T)$.
2. Compute $[\![\alpha]\!]_{\mathbb{G}_T} = e([\![\sigma_2]\!], H)$ using the LSS Homomorphism $x \mapsto xH$.
3. Locally compute $[\![\beta]\!]_{\mathbb{G}_T} = e(\sigma_1, X + \sum_{i=1}^r [\![m_i]\!] Y_i)$ using LSS homomorphisms.
4. Output $[\![b]\!]_{\mathbb{G}_T} \leftarrow \Pi_{\mathsf{ScalarMul}}([\![\rho]\!], [\![\alpha]\!]_{\mathbb{G}_T} / [\![\beta]\!]_{\mathbb{G}_T})$ where $[\![\rho]\!]$ was obtained by invoking $\mathcal{F}_{\mathsf{Rand}}$.

The communication complexity of the Π_{Verify^*} protocol is $\mathcal{C}_{\mathsf{Rand}}(1)+\mathcal{C}_{\mathsf{ScalarMul}}(1)$. We now argue security.

Lemma 2. *Protocol Π_{Verify^*} outputs a secret-sharing of 1 if $\sigma = (\sigma_1, [\![\sigma_2]\!]_{\mathbb{G}_1})$ is a valid signature on $[\![\mathbf{m}]\!]$ with public key pk, otherwise the protocol outputs a secret-sharing of a uniformly random element.*

Proof. Note that $[\![\alpha]\!]_{\mathbb{G}_T} / [\![\beta]\!]_{\mathbb{G}_T} = [\![e(\sigma_1, X + \sum_i m_i Y_i)/e(\sigma_2, H)]\!]_{\mathbb{G}_T}$ which is $1_{\mathbb{G}_T}$ if and only if $e(\sigma_1, X + \sum_i m_i Y_i) = e(\sigma_2, H)$; that is, if the signature is valid. Thus we have that the distribution of $[\![b]\!]_{\mathbb{G}_T} = [\![(a/\beta)^\rho]\!]_{\mathbb{G}_T}$ is either uniformly random (if $\alpha \neq \beta$), or $1_{\mathbb{G}_T}$ (if $\alpha = \beta$). To see that $[\![b]\!]_{\mathbb{G}_T}$ is uniformly random when $\alpha \neq \beta$ it suffices to note that α/β is a generator of $\mathbb{G}_T$ and that ρ was picked at random.

6 Applications to Proactive Secret Sharing

In a real-life scenario secret-sharing techniques can be used to distribute knowledge of a secret among multiple parties in such a way that only certain threshold of parties (or in general, certain subsets defined in some specific way) can reconstruct the secret, whereas any other set of parties obtains no information whatsoever about the secret. Hence, privacy of the secret is guaranteed as long as the adversary cannot get hold of more than certain number of shares. However, as time passes, the chances of information leakage increase and it may become easier for the adversary to gain access to enough shares as to reconstruct the secret. Proactive Secret Sharing (PSS) deals with this problem by "refreshing" the shares with certain periodicity so that the shares gathered by an adversary before this process become "incompatible" with the new shares produced by the

PSS scheme. Furthermore, some PSS schemes, which we call *dynamic PSS*, allow the secret to be transferred from one set of parties to a new one, having perhaps a different number of participants or even a different privacy threshold.

Many works have been devoted to developing and improving PSS schemes in the past [25,4,31,34,9,29]. To date, the most efficient solution for dynamic PSS is CHURP [29], which works by first performing a fast *optimistic* proactivization in which cheating can be detected, but no particular cheater can be pointed out. In case some malicious behavior is detected, the proactivization is repeated with a more expensive method that is able to detect cheaters. In this work we focus on *PSS with abort*, which, as the optimistic protocol in [29], allows transferring a secret from one committee of parties to another one while ensuring privacy of the secret, but a malicious adversary may cause the whole process to abort. However, if the protocol does not abort then it is guaranteed that the correct secret is transmitted to the new committee.

In what follows we show how to apply the secure signatures developed in Section 5 to improve the efficiency of dynamic PSS with abort. Our construction is fundamentally simple: First we develop a highly efficient protocol for proactivizing a secret that guarantees privacy, but allows the adversary to tamper with the transmitted secret. Then, we use our signatures to transmit a signature on the secret, that can be checked by the receiving committee. In this way, due to the unforgeability properties of the PS signature scheme, an adversary cannot make the receiving committee accept and incorrectly transmitted message. This construction leads to a 9-fold improvement in terms of communication with respect to the optimistic protocol from [29].

6.1 Proactive Secret Sharing

First we present the definitions of proactive secret sharing, or PSS for short. We remark that our goal is not to provide formal definitions of these properties but rather a high level description of what a PSS scheme is, so that we can present in a clear manner our optimizations to the work of [29].

In a PSS scheme a set of n parties have shares of a secret $[\![s]\!] = (s_1, \ldots, s_n)$ under a threshold t. At a given stage, a proactivization mechanism is executed, from which the parties obtain $[\![s']\!] = (s'_1, \ldots, s'_n)$. A PSS scheme should satisfy the following properties:

- *(Correctness).* It must hold that $s = s'$
- *(Privacy).* An adversary corrupting a set of at most t parties before the proactivization, and also a (potentially different) set of at most t parties after the proactivization, cannot learn anything about the secret s.

The PSS schemes we consider in this work are *dynamic* in that the set of parties holding the secret before the proactivization step may be different than the set of parties holding the secret afterwards.

6.2 Partial PSS

In what follows we denote by $\mathsf{C} = \{P_i\}_{i=1}^n$ and $\mathsf{C}' = \{P'_i\}_{i=1}^n$ the old a new committees, respectively. Furthermore, we denote $\mathcal{U} = \{P_i\}_{i=1}^{t+1}$ and $\mathcal{U}' = \{P'_i\}_{i=1}^{t+1}$. Also, we consider Shamir Secret Sharing, as defined in Section 3.2. We assume that $n = 2t + 1$. Our protocol $\Pi_{\mathsf{PartialPSS}}$ is inspired by the protocol from [5], except that, since we do not require the transmitted message to be correct, we can remove most of the bottlenecks like the use of hyper-invertible matrices or consistency checks to ensure parties send shares consistently.

Protocol $\Pi_{\mathsf{PartialPSS}}(\llbracket s \rrbracket^{\mathsf{C}})$

Inputs A shared value $\llbracket s \rrbracket^{\mathsf{C}} = (s_1, \ldots, s_n)$ among a committee C.
Output: Either a consistently shared value $\llbracket s' \rrbracket^{\mathsf{C}'}$ or abort. If all parties behave honestly then $s' = s$.

1. Each $P_i \in \mathsf{C}$ samples $s_{i1}, \ldots, s_{i,t+1} \in_R \mathbb{F}$ such that $s_i = \sum_{j=1}^{t+1} s_{ij}$ and sends s_{ij} to P_j for $j = 1, \ldots, t+1$.
2. Each $P_i \in \mathcal{U}$ samples $r_{ki} \in_R \mathbb{F}$ for $k = 1, \ldots, t$, and sets $r_{0,i} = 0$.
3. Each $P_i \in \mathcal{U}$ sets $a_{ij} = s_{ji} + \sum_{k=0}^{t} r_{ki} \cdot j^k$ and sends a_{ij} to P'_j, for each $j = 1, \ldots, n$.
4. Each $P'_j \in \mathsf{C}'$ sets $s'_j := \sum_{i=1}^{t+1} a_{ij}$.
5. The parties in C' output the shares $(s'_1, \ldots, s'_n)$.

Theorem 1. *Protocol $\Pi_{\mathsf{PartialPSS}}$ satisfies the following properties.*

1. *The resulting sharings are consistent. Furthermore, if all the parties behave honestly, then the underlying secret is the same as provided as input.*
2. *An adversary simultaneously controlling t parties in C and t parties in C' does not learn anything about the secret input s.*

Proof. We begin by introducing some notation. Let $\mathcal{A} \subseteq \mathsf{C}$ and $\mathcal{A}' \subseteq \mathsf{C}'$ be the corresponding subsets of corrupt parties. For an honest party P_i it should hold that $s_i = \sum_{j=1}^{t+1} s_{ij}$, where s_{ij} is the additive share sent by P_i to P_j in step 1. However, for $P_i \in \mathcal{A}$, this may not be the case, so we define $\delta_i \in \mathbb{F}$ such that $s_i + \delta_i = \sum_{j=1}^{t+1} s_{ij}$. Finally, each $P_i \in \mathcal{U}$ is supposed to send a_{ij} in step 3, but naturally, parties in $\mathcal{A} \cap \mathcal{U}$ may not follow this. We define ϵ_{ij} for $P_i \in \mathcal{A} \cap \mathcal{U}$ and $j = 1, \ldots, n$ in such a way that $a_{ij} + \epsilon_{ij}$ is the value sent by P_i to P'_j in step 3.

It is easy to see that the value reconstructed by P'_j in step 4 is $s'_j = \sum_{i=1}^{t+1} a_{ij} = \epsilon_j + \delta_j + s_j + \sum_{k=0}^{t} r_k \cdot j^k$, where $\epsilon_j = \sum_{i=1}^{t+1} \epsilon_{ij}$, $r_k = \sum_{i=1}^{t+1} r_{ki}$ (notice that $r_0 = 0$). This can be written as $s'_j = \gamma_j + h(j)$, where $h(x) = f(x) + g(x) \in \mathbb{F}_{\leq t}[x]$, $g(x) = \sum_{k=0}^{t} r_k \cdot x^k \in \mathbb{F}_{\leq t}[x]$ and $\gamma_j = \epsilon_j + \delta_j$.

From the above it follows that the final sharings $s'_j = \gamma_j + h(j)$ output by the honest parties $P'_j \in \mathsf{C}' \setminus \mathcal{A}'$ are consistent: The adversary knows all γ_i, so it can re-define $s'_j \leftarrow s'_j - \gamma_j + q(j)$ for $P'_j \in \mathcal{A}'$, where $q(j) \in \mathbb{F}_{\leq t}[x]$ is such

that $q(i) = \gamma_i$ for $P_i \in \mathsf{C}' \setminus \mathcal{A}'$, and in this way the sharings $(s'_1, \ldots, s'_j)$ are consistent with the polynomial $h(x) + q(x) \in \mathbb{F}_{\leq t}[x]$. Furthermore, if all parties behave honestly then $q(x) \equiv 0$, so the shares $(s'_1, \ldots, s'_n)$ are consistent with the polynomial $h(x)$ which satisfies $h(0) = f(0) + g(0) = s + 0 = s$.

Finally, we show that privacy holds. To see this, it suffices to show that $(s'_1 - s_1, \ldots, s'_n - s_n)$ are uniform shares of some value that the adversary knows. We first claim that, fromt the point of view of the adversary, $(g(1), \ldots, g(n))$ are uniformly random shares of 0. This holds because $s'_j - s_j = (s_j + q(j) + g(j)) - s_j = q(j) + g(j)$

Now we argue privacy. For this we assume that $q(x) \equiv 0$ (that is, the adversary did not cheat overall). This simplifies notation, but it is also without loss of generality because as we saw above the worst thing an adversary can do is shifting the secret by an amount the adversary itself knows. First, notice that the view of the adversary is

$$(\underbrace{\{s_{ij}\}_{P_i \in \mathcal{A}, P_j \in \mathcal{U}}, \{g_i(x)\}_{P_i \in \mathcal{U} \cap \mathcal{A}}}_{\text{Sampled locally}}, \underbrace{\{s_{ij}\}_{P_i \in \mathsf{C}, P_j \in \mathcal{U} \cap \mathcal{A}}}_{\text{Received in step 1}}, \underbrace{\{a_{ij}\}_{P_i \in \mathcal{U}, P'_j \in \mathcal{A}'}}_{\text{Received in step 4}}),$$

where $g_i(x) = \sum_{k=0}^{t} r_{ki} \cdot x^t$ (notice that $g(x) = \sum_{i=1}^{t+1} g_i(x)$). We claim that this view is independent of the secret s. To see this, we define a simulator $\mathcal{S}$ that, on input $(\{s_{ij}\}_{P_i \in \mathcal{A}, P_j \in \mathcal{U}}, \{g_i(x)\}_{P_i \in \mathcal{U} \cap \mathcal{A}})$ and without knowledge of s, produces an indistinguishable view.

The simulator $\mathcal{S}$ is defined as follows:

- Sample $\mathtt{s}_{ij} \in_R \mathbb{F}$ for $P_i \in \mathsf{C} \setminus \mathcal{A}, P_j \in \mathcal{U} \cap \mathcal{A}$, and set $\mathtt{s}_{ij} := s_{ij}$ for $P_i \in \mathcal{A}, P_j \in \mathcal{U} \cap \mathcal{A}$.
- Define $\mathtt{a}_{ij} := \mathtt{s}_{ji} + g_i(j)$ for $P_i \in \mathcal{U} \cap \mathcal{A}, P'_j \in \mathcal{A}'$, and $\mathtt{a}_{ij} \in_R \mathbb{F}$ for $P_i \in \mathcal{U} \setminus \mathcal{A}, P'_j \in \mathcal{A}'$
- Output

$$(\{\mathtt{s}_{ij}\}_{P_i \in \mathcal{A}, P_j \in \mathcal{U}}, \{\mathtt{g}_i(x)\}_{P_i \in \mathcal{A}}, \{\mathtt{s}_{ij}\}_{P_i \in \mathsf{C}, P_j \in \mathcal{U} \cap \mathcal{A}}, \{\mathtt{a}_{ij}\}_{P_i \in \mathcal{U}, P'_j \in \mathcal{A}'}).$$

The two views are perfectly indistinguishable: $\{s_{ij}\}_{P_i \in \mathsf{C}, P_j \in \mathcal{U} \cap \mathcal{A}} \equiv \{\mathtt{s}_{ij}\}_{P_i \in \mathsf{C}, P_j \in \mathcal{U} \cap \mathcal{A}}$ because, given that $|\mathcal{U} \cap \mathcal{A}| \leq t < t + 1$, in the real execution the honest parties $P_i \in \mathsf{C} \setminus \mathcal{A}$ sample $\{s_{ij}\}_{P_j \in \mathcal{U} \cap \mathcal{A}}$ independently and uniformly at random, like in the simulation. Also $\{a_{ij}\}_{P_i \in \mathcal{U}, P'_j \in \mathcal{A}'} \equiv \{\mathtt{a}_{ij}\}_{P_i \in \mathcal{U}, P'_j \in \mathcal{A}'}$ given the rest of the views because, in the real execution, $\{a_{ij}\}_{P_i \in \mathcal{U} \setminus \mathcal{A}, P'_j \in \mathcal{A}'}$ are uniformly random since they are only conditioned on $a_j = \sum_{i=1}^{t+1} a_{ij} = s_j + g(j)$ for $P'_j \in \mathcal{A}'$, but since $|\mathcal{A}'| \leq t$ and $g(x) \in_R \mathbb{F}_{\leq t}[x]$ with $g(0) = 0$, $\{g(j)\}_{P'_j \in \mathcal{A}'}$ are independent and uniform so $\{a_j\}_{P_j \in \mathcal{A}'}$ look uniform and independent to the adversary. □

Extending to group elements. Protocol $\Pi_{\mathsf{PartialPSS}}$ can be extended in a straightforward way to proactivize shares $[\![\alpha]\!]^{\mathsf{C}}_{\mathbb{G}}$, where $\mathbb{G}$ is an elliptic curve group by running the exact same protocol "in the exponent". More formally, the LSSS homomorphism $x \mapsto x \cdot G$, where G is a generator of $\mathbb{G}$, is used.

Communication Complexity. In protocol $\Pi_{\mathsf{PartialPSS}}$ there is a total of $n(n+1)$ field elements communicated.

6.3 Simple and Efficient PSS with Abort

The protocol $\Pi_{\mathsf{PartialPSS}}$ presented in the previous section guarantees privacy and consistency of the new sharings, but it does not satify the main property of a PSS, which is guaranteeing that the secret remains the same, if a malicious party deviates from the protocol. More precisely, a malicious party may disrupt the output as $[\![s+\gamma]\!]^{\mathsf{C}'} \leftarrow \Pi_{\mathsf{PartialPSS}}([\![s]\!]^{\mathsf{C}})$, where γ is some value known by the adversary. This is of course not ideal, but it can be easily fixed by making use of the secure signatures we proposed in Section 5. In a nutshell, the committee C uses $\Pi_{\mathsf{PartialPSS}}$ to send to C' not only the secret s, but also a signature on this secret using a secret-key shared by C. Then, upon receiving shares of the message-signature pair, the parties in C' proceed to verifying this pair securely using C's public key, and if this check passes then it can be guaranteed that the message was correct, since the adversary cannot produce a valid message-signature pair for a new message.

The protocol is presented more formally in Protocol Π_{PSS} below.

Protocol $\Pi_{\mathsf{PSS}}([\![s]\!]^{\mathsf{C}})$

Inputs A shared value $[\![s]\!]^{\mathsf{C}} = (s_1, \ldots, s_n)$ among a committee C.
Output: Consistent shares $[\![s]\!]^{\mathsf{C}'}$ or abort.
Setup: Parties in C have a shared secret-key $[\![\mathsf{sk}_{\mathsf{C}}]\!]^{\mathsf{C}}$, and its corresponding public key pk_{C} is known by the parties in C'.[a]

1. Parties in C call $(\sigma_1, [\![\sigma_2]\!]^{\mathsf{C}}) \leftarrow \Pi_{\mathsf{Sign}}([\![\mathsf{sk}_{\mathsf{C}}]\!]^{\mathsf{C}}, [\![s]\!]^{\mathsf{C}})$.
2. Parties in $\mathsf{C} \cup \mathsf{C}'$ call $[\![s']\!]^{\mathsf{C}'} \leftarrow \Pi_{\mathsf{PartialPSS}}([\![s]\!]^{\mathsf{C}})$ and $[\![\sigma_2']\!]^{\mathsf{C}'} \leftarrow \Pi_{\mathsf{PartialPSS}}([\![\sigma_2]\!]^{\mathsf{C}})$.
3. $P_1, \ldots, P_{t+1}$ all send σ_1 to the parties in C'. If some party in $P_j \in \mathsf{C}'$ receives two different σ_1 from two different parties, then the parties abort.
4. Parties in C' call $v \leftarrow \Pi_{\mathsf{Verify}^*}([\![s']\!]^{\mathsf{C}'}, (\sigma_1, [\![\sigma_2']\!]^{\mathsf{C}'}), \mathsf{pk}_{\mathsf{C}})$. If $v = 0$ then the parties in C' output $[\![s']\!]^{\mathsf{C}'}$. Else, they abort.

[a] This can be easily generated by using protocol Π_{Keygen}.

Intuitively, the protocol guarantees that the parties do not abort if and only if the message is transmitted correctly. This follows from the unforgeability of the PS signature scheme: If an adversary can cause the parties to accept with a wrong message/signature pair, then this would constitute a forged signature. The fact that privacy is maintained regardless of whether the parties abort or not is more subtle, and follows from the fact that this bit of information indicating

whether the protocol aborted can be shown to be independent of the secret, and only depends on the errors introduced by the adversary.

We summarize these properties in Theorem 2 below, where we provide a sketch of the corresponding simulation-based proof. However, in our proof we do not reduce to the unforgeability of the signature scheme directly, but rather to a hard problem over elliptic curves directly. This is easier and cleaner in our special setting, given that the signatures are produced and checked within the same protocol. The computational problem we reduce the security of Protocol Π_{PSS} to is the following, which can be seen as a natural variant of Computational Diffie-Hellman (CDH) problem over $\mathbb{G}_1$.

Definition 5 (co-CDH assumption). *Let $G \in \mathbb{G}_1$ and $G' \in \mathbb{G}_2$ be generators. Given (G, G', aG, bG') for $a, b, \in_R \mathbb{F}$, an adversary cannot efficiently find $(ab)G$.*

With this assumption at hand, which is assumed to hold for certain choices of pairing settings [19], we can discuss the security of Π_{PSS}.

Theorem 2. *Protocol Π_{PSS} instantiates the PSS-with-abort functionality described in Section 6.1, that is, if the parties do not abort in the protocol Π_{PSS}, then the parties in C' have shares $[\![s]\!]^{\mathsf{C}'}$, where $[\![s]\!]^{\mathsf{C}}$ was the input provided to the protocol. Furthermore, privacy of s is satisfied regardless of whether the parties abort or not.*

Proof (Sketch). We only provide a sketch of the corresponding simulation-based proof. Let $s' = s + \delta$ and $\sigma_2' = \sigma_1 + \gamma$, where $\delta \in \mathbb{F}$ and $\gamma \in \mathbb{G}_1$ are the errors introduced by the adversary in the $\Pi_{\mathsf{PartialPSS}}$ protocol. Our simulator simply emulates the role of the honest parties, with these virtual honest parties using random shares as inputs. The simulator also emulates all the necessary functionalities like $\mathcal{F}_{\mathsf{DotProd}*}$, $\mathcal{F}_{\mathsf{Coin}}$ and $\mathcal{F}_{\mathsf{Rand}}$. Using an argument along the lines of the proof of Theorem 1, the simulator is then able to learn the errors δ and γ. The simulator then makes the virtual parties abort if $\delta \neq 0$ or $\gamma \neq 0_{\mathbb{G}_1}$.

We show that the simulated execution is indistinguishable to the adversary from a real execution. To see this, first observe that in the real execution, the honest parties abort if the output of Verify^* is not 0. Furthermore, it is easy to see that the output of $\Pi_{\mathsf{Verify}^*}([\![s']\!]^{\mathsf{C}'}, (\sigma_1, [\![\sigma_2']\!]^{\mathsf{C}'}), \mathsf{pk}_{\mathsf{C}})$ is equal to 0 if and only if $\delta \cdot e(\sigma_1, Y) = e(\gamma, H)$. Given this, the only scenario in which the two executions (real and simulated) could differ is if $\delta \neq 0$ or $\gamma \neq 0_{\mathbb{G}_1}$, but $\delta \cdot e(\sigma_1, Y) = e(\gamma, H)$, since in this case the honest parties in the real execution do not abort, but the honest parties in the ideal execution do. However, we show this cannot happen: If $\delta \neq 0$ or $\gamma \neq 0_{\mathbb{G}_1}$, then $\delta \cdot e(\sigma_1, Y) \neq e(\gamma, H)$, with overwhelming probability.

To see why the claim above holds, we make a reduction to the co-CDH problem defined above: An adversary gets challenged with $(\alpha_1 H, \alpha_2 H')$, and its goal is to find $\alpha_1\alpha_2 H$. The adversary then plays the simulator above, but uses $\sigma_1 = \alpha_1 H$ and $Y = \alpha_2 H'$. Now suppose that in the simulation $\delta \neq 0$ and $\delta \cdot e(\sigma_1, Y) = e(\gamma, H')$. We can see then that this equation implies that $\delta\alpha_1\alpha_2 = \beta$, where $\beta \in \mathbb{F}$ is such that $\gamma = \beta H'$. In particular, it implies that $\alpha_1\alpha_2 H = \delta^{-1}\beta H = \delta^{-1}\gamma$, so the adversary, who knows δ and γ, can compute

$\alpha_1\alpha_2 H$ as above, thus breaking co-CDH. Finally, it is easy to see that if $\gamma \neq 0$ and $\delta \cdot e(\sigma_1, Y) = e(\gamma, H)$, then $\delta \neq 0$ with high probability since otherwise $e(\gamma, H) = 0$, so the same argument as above works. This finishes the sketch of the simulation-based proof of the theorem. □

Proactivizing multiple shared elements. If multiple shared elements $[\![s_1]\!]^{\mathsf{C}}, \ldots, [\![s_L]\!]^{\mathsf{C}}$ are to be proactivized, we can make use of the fact that the signature scheme described in Section 5 allows for cheap signing and verification of long messages without penalty in communication.

Optimizing the signatures. As we noted in Section 5.2, we can use the more efficient functionality $\mathcal{F}_{\mathsf{DotProd}}$ instead of $\mathcal{F}_{\mathsf{DotProd}*}$, at the expense of allowing the adversary to produce incorrect signatures by adding any error to the second component of the signature. However, this is completely acceptable in our setting. In fact, the adversary can already add an error to the second component of the signature when using the $\Pi_{\mathsf{PartialPSS}}$ protocol. Hence, in our protocol Π_{PSS} we use the modified version of Π_{Sign} that uses $\mathcal{F}_{\mathsf{DotProd}}$ instead of $\mathcal{F}_{\mathsf{DotProd}*}$.

Using AMD codes. The fact that the worst that can happen in the $\Pi_{\mathsf{PartialPSS}}$ protocol is that the transmitted message is wrong by an additive amount known by the adversary implies that other methods to ensure correctness of the transmitted value can be devised. This is described in Section B.2 in the appendix. Although the overall computation is much more efficient since it does not involve any public-key operations, the communication of the method we present here is worse by a factor of 2.

Communication Complexity. The communication complexity of the Π_{PSS} protocol is $\mathcal{C}_{\mathsf{PartialPSS}}(L+1) + \mathcal{C}_{\mathsf{Sign}}(L) + \mathcal{C}_{\mathsf{Verify}}(L)$. Recall that $\mathcal{C}_{\mathsf{Sign}}(L) = \mathcal{C}_{\mathsf{Coin}}(1) + \mathcal{C}_{\mathsf{DotProd}}(L)$, and $\mathcal{C}_{\mathsf{Verify}}(L) = \mathcal{C}_{\mathsf{Rand}}(1) + \mathcal{C}_{\mathsf{ScalarMul}}(1)$ For the case of Shamir secret sharing, $\mathcal{C}_{\mathsf{Rand}}(1) = 2n \log |\mathbb{F}|$, using the protocol from [17] and amortizing over multiple calls to $\mathcal{F}_{\mathsf{Rand}}$. Also, $\mathcal{C}_{\mathsf{DotProd}}(L) = 5.5n \log |\mathbb{F}|$, and $\mathcal{C}_{\mathsf{ScalarMul}}(1) = 5.5n \log |\mathbb{F}|$ too, using the specialized bilinear protocol $\Pi^{\mathsf{shm}}_{\mathsf{DotProd}}$ for Shamir SS described in Section 3.2. We ignore the cost $\mathcal{C}_{\mathsf{Coin}}(1)$ since it can be instantiated non-interactively using a PRG.

Given the above, the total communication complexity of the Π_{PSS} protocol is

$$\log(|\mathbb{F}|) \cdot ((L+1) \cdot n \cdot (n+1) + 13n) \text{ bits.}$$

Comparison with CHURP. This dynamic PSS protocol, proposed in [29], is the state of the art in terms of communication complexity. At a high level, CHURP is made of two main protocols, Opt-CHURP, which is able to detect malicious behavior during the proactivization but is not able to point out which party or parties cheated, and Exp-CHURP, which performs proactivization while enabling cheater detection at the expense of being heavier in terms of communication. Since in this work we have described a PSS protocol *with abort*, we compare our protocol against Opt-CHURP.

The total communication complexity of Opt-CHURP is $9Ln^2 \log |\mathbb{F}|$ bits in point-to-point channels, plus $256n$ bits over a blockchain.[6] Furthermore, although not mentioned in our protocol, a lot of the communication that appears in the $13n$ term in our Π_{PSS} protocol can be regarded as preprocessing, that is, it is independent of the message being transmitted and can be computed in advance, before the proactivization phase.

Finally, we note that our novel protocol Π_{PSS} is much more conceptually simple than Opt-CHURP. This is because, unlike in Opt-CHURP, our protocol does not require the expensive use of commitments and proofs at the individual level (i.e. *per party*) in order to ensure correctness of the transmitted value. Instead, we compute a *global* signature of the secret and check its validity after the proactivization.

7 Applications to Input Certification

MPC, classically, does not put any restriction on what kind of inputs are allowed, yet such a property has its place in many applications. (For example, one might want to ensure that the two parties in the *millionaires problem* do not lie about their fortunes).

Signatures seem like the obvious candidate primitive for certifying inputs in MPC: A trusted party $\mathcal{T}$ will sign all inputs x_i of party P_i that need certification. Then, after P_i have shared its input $[\![x_i']\!]$, which it may change if it is misbehaving, parties will verify that $[\![x_i']\!]$ is a value that was previously signed by $\mathcal{T}$. While this approach clearly works (if P_i could get away with sharing x_i', then P_i produced a forgery) it is nevertheless hindered by the fact that signature verification is expensive to compute on secret-shared values, arising from the fact that the usual first step in verifying a signature is hashing the message, which is prohibitely expensive in MPC. In this section we show that by using our secure PS signatures from Section 5, this approach is not longer infeasible, and in fact, it is quite efficient.

7.1 Certifying inputs with PS signatures

We consider a setting in which n parties $P_1, \ldots, P_n$ wish to compute a function $f(\mathbf{x}_1, \ldots, \mathbf{x}_n)$, where $\mathbf{x}_i \in \mathbb{F}^L$ corresponds to the input of party P_i. We assume that all parties hold the public key pk of some trusted authority $\mathcal{T}$, who provided each P_i with a PS signature (σ_1^i, σ_2^i) on its input $\mathbf{x}_i$. We also assume a functionality $\mathcal{F}_{\mathsf{Input}}$ that, on input $\mathbf{x}_i$ from P_i, distributes to the parties consistent shares $[\![x_{i1}]\!], \ldots, [\![x_{iL}]\!]$. This is implemented in different ways dependending on the secret-sharing scheme. We also assume the existence of a broadcast channel.

Using the protocols presented in Section 5, a very simple protocol for certifying inputs. Our protocol, $\Pi_{\mathsf{CertInput}}$, allows a party P_i to distribute shares of its input, only if this input has been previously certified.

[6] For a more detailed derivation of this complexity, see Section B.1 in the appendix.

Protocol $\Pi_{\mathsf{CertInput}}$

Input: Index $i \in \{1, \ldots, n\}$ and $\left((x_i)_{i=1}^L, \sigma_1, \sigma_2\right)$ from P_j.
Output: $([\![x_i]\!])_i$ where $\mathsf{Verify}(\mathsf{pk}, ([\![x_i]\!]_i), (\sigma_1, \sigma_2)) = 1$, or abort.

1. P_j calls $\mathcal{F}_{\mathsf{Input}}$ to distribute $\left(([\![x_i]\!])_i, [\![\sigma_2]\!]_{\mathbb{G}_1}\right)$. Also, P_j broadcasts σ_1 to all parties.
2. Parties call $[\![r]\!]_{\mathbb{G}_T} \leftarrow \Pi_{\mathsf{Verify}^*}(\mathsf{pk}, ([\![x_i]\!])_{i=1}^L, \sigma_1, [\![\sigma_2]\!]_{\mathbb{G}_1})$.
3. Parties open $[\![r]\!]_{\mathbb{G}_T}$, who output $([\![x_i]\!])_i$ if $r = 1_{\mathbb{G}_T}$ and abort otherwise.

The security of the protocol follows seamlessly from the unforgeability of the PS signatures, proven in [33].

Complexity analysis. The communication complexity of the protocol $\Pi_{\mathsf{CertInput}}$ is $\mathcal{C}_{\mathsf{Input}}(L) + \mathcal{C}_{\mathsf{Verify}}(L) + \mathcal{C}_{\mathsf{Open}}(1)$ bits.

Optimization if multiple parties provide input. If all parties $P_1, \ldots, P_n$ use $\Pi_{\mathsf{CertInput}}$ to certify their input, each party can call $\Pi_{\mathsf{CertInput}}$, which, in the case that a protocol with guaranteed output delivery is used to compute Π_{Verify}, allows parties to identify exactly which party provided a faulty input. However, one can improve the communication complexity if a "global" abort is accepted, that is, if the parties do not abort then *all* the inputs are correctly certified, but if they do abort, then it is not possible to identify which party provided an incorrect input (however, for protocols without guaranteed output delivery, this is acceptable since the abort can already happen due to malicious behavior in other parts of the protocol).

The optimization works as follows. Consider the n $\Pi_{\mathsf{CertInput}}$ executions, corresponding to all parties. At the end of step 2, n shares $[\![r_1]\!]_{\mathbb{G}_T}, \ldots, [\![r_n]\!]_{\mathbb{G}_T}$ have been produced. The parties then locally compute $[\![r]\!]_{\mathbb{G}_T} = \prod_{i=1}^n [\![r_i]\!]_{\mathbb{G}_T}$ (recall that $\mathbb{G}_T$ is a multiplicative group), open r, and accept the secret-shared inputs if and only if this opened value equals $1_{\mathbb{G}_T}$. Notice that, if at least one signature is incorrect, then at least one r_i is uniformly random, so r will be uniformly random too and therefore the probability that it equals $1_{\mathbb{G}_T}$ in this case is at most $1/|\mathbb{G}_T|$.

7.2 Comparison with [8]

Certifying inputs for MPC with the help of signatures has been studied previously in [8]. However, the approach followed in that work is conceptually much more complex than the one we presented here. At a high level, instead of verifying the signature in MPC, the parties jointly produce commitments of the secret-shared inputs, and then each input owner uses these commitments, together with the signatures, to prove via an interactive protocol (that roughly resembles a zero-knowledge proof of knowledge) "posession" of the signatures. Furthermore, the protocols presented in [8] depend on the underlying secret-sharing scheme

used, and two ad-hoc constructions, one for Shamir secret-sharing (using the MPC protocol from [17]) and another one for additive secret sharing (using the MPC protocol from [16]), are presented. Instead, our approach is completely general and applies to any linear secret-sharing scheme, as defined in Section 2.

We present in Section 8.1 a more experimental and quantitative comparison between our work and [8]. We observe that, in general, our approach is at least 2 times more efficient.

8 Implementation and Benchmarking

We implemented our protocols with the help of the RELIC toolkit [2] using the pairing-friendly BLS12-381 curve as the choice of parameters. This curve has embedding degree $k = 12$ and a 255-bit prime-order subgroup, and became popular after the ZCash cryptocurrency [6]. It is now in the process of standardization due to its attractive performance characteristics, including an efficient towering of extensions, efficient GLV endomorphisms for scalar multiplications in the pairing groups, cyclotomic squarings for fast exponentiation in $\mathbb{G}_T$, among others. In terms of security, the choice is motivated by recent attacks against the DLP in $\mathbb{G}_T$ [28] and are supported by the analysis in [30]. Our implementations make use of all optimizations implemented in RELIC, including Assembly acceleration for Intel 64-bit platforms, and extend the supported algorithms to allow computation of arbitrarily-sized linear combinations of $\mathbb{G}_2$ points through NAF interleaving. We take special care to batch operations which can performed simultaneously, for example merging scalar multiplications together or combining the two pairing computations within MPC signature verification as a product of pairings. We deliberately enabled the variable-time but faster algorithms in the library relying on the timing-attack resistance built in MPC, since computations will be performed essentially over ephemeral data. The resulting code is available in the library repository.

We benchmarked our implementation on an Intel Core i7-7820X Skylake processor running at 3.6GHz. We turned off HyperThreading and TurboBoost to reduce noise in the benchmarks. Each procedure was executed 10^4 times and the averages are computed and reported in Table 1. It can be seen from the table that the MPC versions of scalar multiplications and exponentiations introduce a computational overhead ranging from 2.17 to 2.77, while pairing computation becomes only 30% slower. We notice that performance impact is higher for exponentiation in $\mathbb{G}_T$ due to a less efficient implementation in RELIC. This is justified by the lower prevalence of such operations in pairing-based protocols compared to operations in $\mathbb{G}_1$ and $\mathbb{G}_2$. For the PS protocol, key generation and signature verification in MPC are penalized approximately by a 2-factor, while the cost of signature computation stays essentially the same. There is no performance penalty for signature computation involving many messages because of the batching possibility in the PS signature scheme.

Operation		Local (cc)	Two-party (cc)
Scalar multiplication in $\mathbb{G}_1$		386	840
Scalar multiplication in $\mathbb{G}_2$		1,009	2,417
Exponentiation in $\mathbb{G}_T$		1,619	4,483
Pairing computation		3,107	4,063
PS key generation	(1 msg)	2,670	4,723
PS signature computation	(1 msg)	626	532
PS signature verification	(1 msg)	5,153	11,514
PS key generation	(10 msgs)	11,970	23,464
PS signature computation	(10 msgs)	656	532
PS signature verification	(10 msgs)	11,131	16,216

Table 1. Efficiency comparison between local computation and two-party computation of the main operations in pairing groups and PS signature computation/verification. We display execution times in 10^3 clock cycles (cc) for each of the main operations in the protocols and report the average for each of the two parties.

8.1 Certified Inputs

Here we compare our protocol for input certification from Section 7 with the experimental results reported in [8]. To perform a fair comparison, we converted the timing results from the second half of Table 2 in [8] to clock cycles using the reported CPU frequency of 2.1GHz for an Intel Sandy Bridge Xeon E5-2620 machine. Each procedure in our implementation was executed 10^4 times for up to 10^2 messages, after which we decreased the number of executions linearly with the increase in messages by the same factor. We used as reference the largest running time of the two running parties (input provider and other party) reported in [8], since the computation would be bounded by the maximum running time. Our results are shown in Table 2, and show that our implementations are competitive for small numbers of messages, but improve on related work by a 2-factor when the number of messages is at least 100. While the two benchmarking machines are different (Intel Sandy Bridge and Skylake), our implementations do not make use of any performance feature specific to Skylake, such as more advanced vector instruction sets. Hence we claim that the performance of our implementations would not be substantially different in Sandy Bridge, and just converting performance figures to clock cycles makes the results generally comparable.

References

1. T. Araki, J. Furukawa, Y. Lindell, A. Nof, and K. Ohara. High-throughput semi-honest secure three-party computation with an honest majority. In E. R. Weippl, S. Katzenbeisser, C. Kruegel, A. C. Myers, and S. Halevi, editors, *ACM CCS 2016*, pages 805–817. ACM Press, Oct. 2016.

Protocol	Number of messages					
	1	10	10^2	10^3	10^4	10^5
Ours	11,514	16,216	66,204	567,171	5,542,000	58,191,400
Related work[8]	11,445	18,690	103,950	970,200	9,723,000	111,090,000

Table 2. Efficiency comparison between our certified input protocol from Section 7 and the one presented in [8]. Numbers are measured in thousands of clock cycles (cc).

2. D. F. Aranha, C. P. L. Gouvêa, T. Markmann, R. S. Wahby, and K. Liao. RELIC is an Efficient LIbrary for Cryptography. `https://github.com/relic-toolkit/relic`.
3. J. Bar-Ilan and D. Beaver. Non-cryptographic fault-tolerant computing in constant number of rounds of interaction. In P. Rudnicki, editor, *8th ACM PODC*, pages 201–209. ACM, Aug. 1989.
4. J. Baron, K. El Defrawy, J. Lampkins, and R. Ostrovsky. How to withstand mobile virus attacks, revisited. In M. M. Halldórsson and S. Dolev, editors, *33rd ACM PODC*, pages 293–302. ACM, July 2014.
5. J. Baron, K. El Defrawy, J. Lampkins, and R. Ostrovsky. Communication-optimal proactive secret sharing for dynamic groups. In T. Malkin, V. Kolesnikov, A. B. Lewko, and M. Polychronakis, editors, *ACNS 15*, volume 9092 of *LNCS*, pages 23–41. Springer, Heidelberg, June 2015.
6. E. Ben-Sasson, A. Chiesa, C. Garman, M. Green, I. Miers, E. Tromer, and M. Virza. Zerocash: Decentralized anonymous payments from bitcoin. In *2014 IEEE Symposium on Security and Privacy*, pages 459–474. IEEE Computer Society Press, May 2014.
7. M. Blanton and F. Bayatbabolghani. Efficient server-aided secure two-party function evaluation with applications to genomic computation. *PoPETs*, 2016(4):144–164, Oct. 2016.
8. M. Blanton and M. Jeong. Improved signature schemes for secure multi-party computation with certified inputs. In J. López, J. Zhou, and M. Soriano, editors, *ESORICS 2018, Part II*, volume 11099 of *LNCS*, pages 438–460. Springer, Heidelberg, Sept. 2018.
9. C. Cachin, K. Kursawe, A. Lysyanskaya, and R. Strobl. Asynchronous verifiable secret sharing and proactive cryptosystems. In V. Atluri, editor, *ACM CCS 2002*, pages 88–97. ACM Press, Nov. 2002.
10. J. Camenisch and A. Lysyanskaya. Signature schemes and anonymous credentials from bilinear maps. In M. Franklin, editor, *CRYPTO 2004*, volume 3152 of *LNCS*, pages 56–72. Springer, Heidelberg, Aug. 2004.
11. K. Chida, D. Genkin, K. Hamada, D. Ikarashi, R. Kikuchi, Y. Lindell, and A. Nof. Fast large-scale honest-majority MPC for malicious adversaries. In H. Shacham and A. Boldyreva, editors, *CRYPTO 2018, Part III*, volume 10993 of *LNCS*, pages 34–64. Springer, Heidelberg, Aug. 2018.
12. R. Cramer, I. Damgård, and U. Maurer. General secure multi-party computation from any linear secret-sharing scheme. In *International Conference on the Theory and Applications of Cryptographic Techniques*, pages 316–334. Springer, 2000.
13. R. Cramer, I. B. Damgård, and J. B. Nielsen. *Secure multiparty computation*. Cambridge University Press, 2015.

14. A. Dalskov, D. Escudero, and M. Keller. Secure evaluation of quantized neural networks. Cryptology ePrint Archive, Report 2019/131, 2019. https://eprint.iacr.org/2019/131.
15. A. Dalskov, M. Keller, C. Orlandi, K. Shrishak, and H. Shulman. Securing DNSSEC keys via threshold ECDSA from generic MPC. Cryptology ePrint Archive, Report 2019/889, 2019. https://eprint.iacr.org/2019/889.
16. I. Damgård, M. Keller, E. Larraia, V. Pastro, P. Scholl, and N. P. Smart. Practical covertly secure MPC for dishonest majority - or: Breaking the SPDZ limits. In J. Crampton, S. Jajodia, and K. Mayes, editors, *ESORICS 2013*, volume 8134 of *LNCS*, pages 1–18. Springer, Heidelberg, Sept. 2013.
17. I. Damgård and J. B. Nielsen. Scalable and unconditionally secure multiparty computation. In A. Menezes, editor, *CRYPTO 2007*, volume 4622 of *LNCS*, pages 572–590. Springer, Heidelberg, Aug. 2007.
18. P. Feldman. A practical scheme for non-interactive verifiable secret sharing. In *28th FOCS*, pages 427–437. IEEE Computer Society Press, Oct. 1987.
19. D. Fiore and R. Gennaro. Publicly verifiable delegation of large polynomials and matrix computations, with applications. In T. Yu, G. Danezis, and V. D. Gligor, editors, *ACM CCS 2012*, pages 501–512. ACM Press, Oct. 2012.
20. P.-A. Fouque, A. Joux, and M. Tibouchi. Injective encodings to elliptic curves. In C. Boyd and L. Simpson, editors, *ACISP 13*, volume 7959 of *LNCS*, pages 203–218. Springer, Heidelberg, July 2013.
21. J. A. Garay, R. Gennaro, C. Jutla, and T. Rabin. Secure distributed storage and retrieval. *Theoretical Computer Science*, 243(1-2):363–389, 2000.
22. P. Gemmell and M. Sudan. Highly resilient correctors for polynomials. *Information processing letters*, 43(4):169–174, 1992.
23. V. Goyal and Y. Song. Malicious security comes free in honest-majority mpc. Cryptology ePrint Archive, Report 2020/134, 2020. https://eprint.iacr.org/2020/134.
24. A. Herzberg, M. Jakobsson, S. Jarecki, H. Krawczyk, and M. Yung. Proactive public key and signature systems. In R. Graveman, P. A. Janson, C. Neuman, and L. Gong, editors, *ACM CCS 97*, pages 100–110. ACM Press, Apr. 1997.
25. A. Herzberg, S. Jarecki, H. Krawczyk, and M. Yung. Proactive secret sharing or: How to cope with perpetual leakage. In D. Coppersmith, editor, *CRYPTO'95*, volume 963 of *LNCS*, pages 339–352. Springer, Heidelberg, Aug. 1995.
26. A. Kate, G. M. Zaverucha, and I. Goldberg. Constant-size commitments to polynomials and their applications. In M. Abe, editor, *ASIACRYPT 2010*, volume 6477 of *LNCS*, pages 177–194. Springer, Heidelberg, Dec. 2010.
27. J. Katz, A. J. Malozemoff, and X. Wang. Efficiently enforcing input validity in secure two-party computation. Cryptology ePrint Archive, Report 2016/184, 2016. http://eprint.iacr.org/2016/184.
28. T. Kim and R. Barbulescu. Extended tower number field sieve: A new complexity for the medium prime case. In M. Robshaw and J. Katz, editors, *CRYPTO 2016, Part I*, volume 9814 of *LNCS*, pages 543–571. Springer, Heidelberg, Aug. 2016.
29. S. K. D. Maram, F. Zhang, L. Wang, A. Low, Y. Zhang, A. Juels, and D. Song. CHURP: Dynamic-committee proactive secret sharing. In L. Cavallaro, J. Kinder, X. Wang, and J. Katz, editors, *ACM CCS 2019*, pages 2369–2386. ACM Press, Nov. 2019.
30. A. Menezes, P. Sarkar, and S. Singh. Challenges with assessing the impact of NFS advances on the security of pairing-based cryptography. In *Mycrypt*, volume 10311 of *Lecture Notes in Computer Science*, pages 83–108. Springer, 2016.

31. R. Ostrovsky and M. Yung. How to withstand mobile virus attacks (extended abstract). In L. Logrippo, editor, *10th ACM PODC*, pages 51–59. ACM, Aug. 1991.
32. C. Peikert. On error correction in the exponent. In S. Halevi and T. Rabin, editors, *TCC 2006*, volume 3876 of *LNCS*, pages 167–183. Springer, Heidelberg, Mar. 2006.
33. D. Pointcheval and O. Sanders. Short randomizable signatures. In K. Sako, editor, *CT-RSA 2016*, volume 9610 of *LNCS*, pages 111–126. Springer, Heidelberg, Feb. / Mar. 2016.
34. D. A. Schultz, B. Liskov, and M. Liskov. Mobile proactive secret sharing. In R. A. Bazzi and B. Patt-Shamir, editors, *27th ACM PODC*, page 458. ACM, Aug. 2008.
35. A. Shamir. How to share a secret. *Communications of the Association for Computing Machinery*, 22(11):612–613, Nov. 1979.
36. N. P. Smart and Y. Talibi Alaoui. Distributing any elliptic curve based protocol. In M. Albrecht, editor, *17th IMA International Conference on Cryptography and Coding*, volume 11929 of *LNCS*, pages 342–366. Springer, Heidelberg, Dec. 2019.
37. Y. Zhang, M. Blanton, and F. Bayatbabolghani. Enforcing input correctness via certification in garbled circuit evaluation. In S. N. Foley, D. Gollmann, and E. Snekkenes, editors, *ESORICS 2017, Part II*, volume 10493 of *LNCS*, pages 552–569. Springer, Heidelberg, Sept. 2017.

Supplementary Material

A Some Linear Secret Sharing Schemes

A.1 Additive Secret-Sharing

In this scheme each party P_i gets a uniformly random value $r_i \in \mathbb{F}$ subject to $\sum_{i=1}^{n} r_i = s$, where $s \in \mathbb{F}$ is the secret. More formally, this scheme $\mathcal{S}_{\mathsf{add}}$ is defined as $(M_{\mathsf{add}}, \mathsf{label}_{\mathsf{add}})$, where $M_{\mathsf{add}} \in \mathbb{F}^{n \times n}$ is given below, and $\mathsf{label}_{\mathsf{add}}(i) = i$:

$$\begin{pmatrix} r_1 \\ r_2 \\ \vdots \\ r_{n-1} \\ s - r_1 - \cdots - r_{n-1} \end{pmatrix} = \underbrace{\begin{pmatrix} 0 & 1 & 0 & \cdots & 0 \\ 0 & 0 & 1 & \cdots & 0 \\ & & \vdots & & \\ 0 & 0 & 0 & \cdots & 1 \\ 1 & -1 & -1 & \cdots & -1 \end{pmatrix}}_{M_{\mathsf{add}} \in \mathbb{F}^{n \times n}} \cdot \begin{pmatrix} s \\ r_1 \\ r_2 \\ \vdots \\ r_{n-1} \end{pmatrix}$$

It is easy to see that this scheme is $(n-1, n)$-secure. Let us denote additive secret sharing of s by $[\![s]\!]^{\mathsf{add}}$, and abussing notation, we write $[\![s]\!]^{\mathsf{add}} = (r_1, \ldots, r_n)$, where each r_i is the share of party P_i. Given an elliptic curve group $\mathbb{G}$ of order p, having G as generator, the parties can obtain shares of $s \cdot G$ by locally multiplying the generator G by their share r_i; that is, $[\![s \cdot G]\!]^{\mathsf{add}} = (r_1 \cdot G, \ldots, r_n \cdot G)$.

Reconstruction. The scheme $\mathcal{S}_{\mathsf{add}}$ is mostly used in the dishonest majority setting. However, at reconstruction time, a maliciously corrupt party can lie about his share, causing the reconstructed value to be incorrect. To help solve this issue, actively secure protocols in the dishonest majority share a secret s as $[\![s]\!]^{\mathsf{add}}$, together with $[\![r \cdot s]\!]^{\mathsf{add}}$, where r is a *global* uniformly random value that is also shared as $[\![r]\!]^{\mathsf{add}}$. We denote this by $[\![s]\!]^{\mathsf{add}*}$. At reconstruction time, the adversary may open $[\![s]\!]^{\mathsf{add}}$ to $s + \delta$ where δ is some error known to the adversary. To ensure that $\delta = 0$ (so the correct value is opened), the parties compute $(s + \delta)\,[\![r]\!]^{\mathsf{add}} - [\![r \cdot s]\!]^{\mathsf{add}}$, open this value, and check it equals 0. It is easy to see that this value equals $r \cdot \delta$, but since the adversary may cheat in this opening, this opened value may be $r \cdot \delta - \epsilon$. However, if $\delta \neq 0$, this opened value equals 0 if and only if $r = \epsilon/\delta$, which happens with probability at most $1/|\mathbb{F}|$ since ϵ and δ are chosen independently of the uniformly random r.

The same check can be performed over $\mathbb{G}$: The sharings $[\![s \cdot G]\!]^{\mathsf{add}}_{\mathbb{G}}$ are accompanied by $[\![r \cdot s \cdot G]\!]^{\mathsf{add}}_{\mathbb{G}}$, where r is a *global* uniformly random value that is also shared as $[\![r]\!]^{\mathsf{add}}$. At reconstruction time $[\![s \cdot G]\!]^{\mathsf{add}}_{\mathbb{G}}$ can be opened to $(s + \delta) \cdot G$, and to ensure $\delta = 0$ the parties open $[\![r]\!]^{\mathsf{add}}_{\mathbb{G}} \cdot (s + \delta) \cdot G - [\![r \cdot s \cdot G]\!]^{\mathsf{add}}_{\mathbb{G}}$ and check that this point is the identity. It is easy to see that, like in the case over $\mathbb{F}$, the check passes with probability at most $1/|\mathbb{F}|$ if $\delta \neq 0$.

A.2 Shamir Secret-Sharing

Consider a setting with n parties, and let $0 < t < n$. In this scheme each party P_i gets $f(i)$ where $f(x) \in_R \mathbb{F}_{\leq t}[x]$ subject to $f(0) = s$, and $s \in \mathbb{F}$ is the secret.[7] We denote $[\![s]\!]^{\mathsf{shm}}_{\mathbb{F}} = (f(1), \ldots, f(n))$. More formally, this scheme $\mathcal{S}_{\mathsf{shm}}$ is defined as $(M_{\mathsf{shm}}, \mathsf{label}_{\mathsf{shm}})$, where $M_{\mathsf{shm}} \in \mathbb{F}^{n \times (t+1)}$ is given below, and $\mathsf{label}_{\mathsf{shm}}(i) = i$:

$$\begin{pmatrix} s_1 \\ s_2 \\ \vdots \\ s_{n-1} \\ s_n \end{pmatrix} = \underbrace{\begin{pmatrix} 1^0 & 1^1 & 1^2 & \cdots & 1^t \\ 2^0 & 2^1 & 2^2 & \cdots & 2^t \\ & & \vdots & & \\ (n-1)^0 & (n-1)^1 & (n-1)^2 & \cdots & (n-1)^t \\ n^0 & n^1 & n^2 & \cdots & n^t \end{pmatrix}}_{M_{\mathsf{shm}} \in \mathbb{F}^{n \times (t+1)}} \cdot \begin{pmatrix} s \\ r_1 \\ r_2 \\ \vdots \\ r_t \end{pmatrix}$$

It is easy to see that this scheme is $(n-1, n)$-secure. Over a vector space V, sharing a point $\alpha \in V$ is done by sampling $r_1, \ldots, r_t \in_R V$, and setting the i-th share to be $\alpha_i = \alpha + \sum_{j=1}^{t} i^j \cdot r_j$. In this way, $\alpha_i = f(i)$, where $f(x) = \alpha + \sum_{j=1}^{t} x^j \cdot r_j \in_R V_{\leq t}[x]$. We denote this by $[\![S]\!]^{\mathsf{shm}}_V$.

Reconstruction. Consider a shared value $[\![s]\!]^{\mathsf{shm}} = (f(1), \ldots, f(n))$. If $t \geq n/2$, then it can be shown that, like in the additive scheme from Section 3.2, the adversary can succeed in opening an incorrect value by modifying the shares of the corrupt parties. However, if $t < n/2$, this cannot be done: The honest parties will be able to *detect* that the opened value is not correct. Furthermore, if $t < n/3$, the honest parties can do better: On top of detecting whether the open value is the right one, they can *correct* the errors and compute the right secret. We describe these below, and we also discuss extensions to elliptic curves.

Error detection ($t < n/2$). Assume $t < n/2$, and suppose that a most t shares among $(s_1, \ldots, s_n)$ are incorrect. If all shares $(s_1, \ldots, s_n)$ lie in a polynomial of degree at most t, then the reconstructed secret must be correct, given that a polynomial of degree at most t is determined by *any* $t+1$ points, in particular, it is determined by the $t+1 \leq n-t$ correct shares. In this way, by verifying if all the shares lie in a polynmial of the right degree, the parties can detect whether the reconstructed value is correct or not. This can be done by interpolating a polynomial of degree at most t using the first $t+1$ shares, and then checking whether the other shares are consistent with this polynomial.

Alternatively, the parties can use the *parity check matrix* $H \in \mathbb{F}^{(n-t-1) \times n}$, which satisfies that $A \cdot (s_1, \ldots, s_n)^T$ is the zero-vector if and only if the shares s_i are consistent with a polynomial of degree at most t. This check can be performed for the group sharings $[\![P]\!]_{\mathbb{G}}$ as well.

[7] We assume that $|\mathbb{F}| > n + 1$

Error correction ($t < n/3$). If $t < n/2$ then the parties can detect whether a reconstructed value is correct or not, but they cannot "fix" the errors in case the value is not correct. Under the additional condition $t < n/3$, this can actually be done, that is, the parties can reconstruct the correct value, regardless of any changes the adversary does to the shares from corrupted parties. The algorithm to achieve this proceeds, at least conceptually, as follows: The parties find a subset of $2t + 1$ shares among the announced shares that lies in a polynomial of degree at most t; this set exists because there are at least $n - t \geq 2t + 1$ correct shares. Then, the secret given by this polynomial is taken as the right secret. This is correct because of the same reason as in the previous case: This polynomial is determined by any set of $t + 1$ points among the $2t + 1$ ones that are consistent, and in particular, it is determined by the $t+1 = 2t+1-t$ correct shares, since at most t of them can be incorrect.

The main bottleneck in the reconstruction algorithm sketched above is finding a consistent subset of $2t+1$ shares, since there are exponentially-many such sets. To this end, an error-correction algorithm like Berlekamp Welch is used [22], which has a running time that is polynomial in n.

Finally, it is important to remark that, unlike the error-detection mechanism above, this error-correction procedure *cannot* be performed over the group $\mathbb{G}$. This interesting result was shown in [32].

Dot Products of Shared Vectors. Let $2t+1 = n$, and let U, V, W be $\mathbb{F}$-vector spaces of dimension d with bases $\{u_i\}_{i=1}^d$, $\{v_i\}_{i=1}^d$ and $\{w_i\}_{i=1}^d$, respectively.[8] Consider a bilinear map $\phi : U \times V \to W$. For the rest of this section we consider Shamir secret sharing, and we omit the superscript shm from the sharings, and consider explicitly the degree of the polynomial used for the sharing: $[\![\cdot]\!]^h$ denotes Shamir secret sharing using polynomials of degree at most h.

Consider shared values $[\![x_1]\!]_U^t, \ldots, [\![x_L]\!]_U^t, [\![y_1]\!]_V^t, \ldots, [\![y_L]\!]_V^t$. In this section we describe a protocol to compute $[\![z + \delta]\!]_W^t$, where $z = \sum_{\ell=1}^L \phi(x_\ell y_\ell)$ and $\delta \in W$ is some error known to the adversary. The main building blocks of the protocol are the following:

- The parties can locally obtain $[\![\phi(\alpha, \beta)]\!]_W^{2t}$ from $[\![\alpha]\!]_U^t$ and $[\![\beta]\!]_V^t$. To see this, write $[\![\alpha]\!]_U^t = (f(1), \ldots, f(n))$ and $[\![\beta]\!]_U^t = (g(1), \ldots, g(n))$, for some $f(x) \in U_{\leq t}[x]$ and $g(x) \in V_{\leq t}[x]$ such that $f(0) = \alpha$ and $g(0) = \beta$. Write $f(x) = \sum_{i=0}^t x^i \cdot r_i$ and $g(x) = \sum_{i=0}^t x^i \cdot s_i$, and let $h(x) = \sum_{i,j=1}^t x^{i+j} \cdot \phi(r_i, s_j) \in W_{\leq 2t}[x]$. It is easy to see that $h(0) = \phi(\alpha, \beta)$ and that $h(i) = \phi(f(i), g(i))$ for all $i = 1, \ldots, n$, so $[\![\phi(\alpha, \beta)]\!]_W^{2t} = (h(1), \ldots, h(n))$.
- There exists a protocol Π_{DoubleSh} that produces a pair $([\![w]\!]_W^t, [\![w]\!]_W^{2t})$, where $w \in_R W$. Such a pair can be produced from d pairs $([\![r_i]\!]_{\mathbb{F}}^t, [\![r_i]\!]_{\mathbb{F}}^{2t})$ by defining $[\![w]\!]_W^k = \sum_{i=1}^d [\![r_i]\!]^k \cdot w_i$ for $k = t, 2t$. These pairs over $\mathbb{F}$ can be produced using the protocol from [17].

[8] As in Section 2, the condition that all three spaces have the same dimension is not necessary.

With these tools at hand we are ready to describe our main protocol.

Protocol $\Pi^{\mathsf{shm}}_{\mathsf{DotProd}}$

Inputs: Shared values $[\![x_1]\!]_U, \ldots, [\![x_L]\!]_U, [\![y_1]\!]_V, \ldots, [\![y_L]\!]_V$.
Output: $[\![z+\delta]\!]_W$, where $z = \sum_{\ell=1}^{L} \phi(x_\ell, y_\ell)$ and $\delta \in W$ is some error known to the adversary.

1. Call $([\![w]\!]^t_W, [\![w]\!]^{2t}_W) \leftarrow \Pi_{\mathsf{DoubleSh}}$
2. Parties locally compute $[\![\phi(x_\ell, y_\ell)]\!]^{2t}_W \leftarrow \phi([\![x_\ell]\!]^t_U, [\![y_\ell]\!]^t_V)$, for $\ell = 1, \ldots, L$;
3. Parties compute $[\![e]\!]_W = [\![w]\!]^{2t}_W + \sum_{\ell=1}^{L} [\![\phi(u_\ell, v_\ell)]\!]^{2t}_W$ and send the shares of e to P_1.
4. P_1 uses the $n = 2t+1$ shares received to reconstruct $e+\delta$ (where δ is the error the adversary may introduce by lying about its shares), and broadcasts[a] $e+\delta$ to all parties.
5. All parties set $[\![z+\delta]\!]^t_W = (e+\delta) - [\![w]\!]^t_W$.

[a] A proper broadcast channel must be used.

The protocol is private because the only value that is opened is e, which is a perfectly masked version of the sensitive value z, given that w is uniformly random and unknown to the adversary. The communication complexity of $\Pi^{\mathsf{shm}}_{\mathsf{DotProd}}$ is $\mathcal{C}^{\mathsf{shm}}_{\mathsf{DotProd}} = d \cdot \log(|\mathbb{F}|) \cdot 5.5 \cdot n$, using the optimization from [23].

A.3 Replicated Secret Sharing

This is a $(1,2)$-secure LSSS for 3 parties. In this scheme each party P_i gets (r_i, r_{i+1}), where the sub-indexes wrap modulo 3, and $s = r_1 + r_2 + r_3$, where $s \in \mathbb{F}$ is the secret. We denote $[\![s]\!]^{\mathsf{rep}}_{\mathbb{F}} = ((r_1, r_2), (r_2, r_3), (r_3, r_1))$. More formally, this scheme $\mathcal{S}_{\mathsf{rep}}$ is defined as $(M_{\mathsf{rep}}, \mathsf{label}_{\mathsf{rep}})$, where $M_{\mathsf{rep}} \in \mathbb{F}^{6\times 3}$ is given below, and $\mathsf{label}_{\mathsf{rep}}(i) = \lceil i/2 \rceil$ for $i = 1, \ldots, 6$.

$$\begin{pmatrix} r_1 \\ r_2 \\ r_2 \\ s - r_1 - r_2 \\ s - r_1 - r_2 \\ r_1 \end{pmatrix} = \underbrace{\begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 1 \\ 1 & -1 & -1 \\ 1 & -1 & -1 \\ 0 & 1 & 0 \end{pmatrix}}_{M_{\mathsf{rep}} \in \mathbb{F}^{6\times 3}} \cdot \begin{pmatrix} s \\ r_1 \\ r_2 \end{pmatrix}$$

Reconstruction. Consider a shared value $[\![s]\!]^{\mathsf{rep}} = ((r_1, r_2), (r_2, r_3), (r_3, r_1))$. To open this share, P_1 sends (r_1, r_2), P_2 sends (r_2, r_3), and P_3 sends $(H(r_3), H(r_1))$, where H is a collision resistant hash function. To verify that the opening is done correctly, the shares announced by P_1 and P_2 are checked against the hashes

announced by P_3. If they are consistent, since at most one party is corrupt, the secret is correct.

Dot Products of Shared Vectors. Like in Section A.2, let U, V, W be $\mathbb{F}$-vector spaces of dimension d with bases $\{u_i\}_{i=1}^d$, $\{v_i\}_{i=1}^d$ and $\{w_i\}_{i=1}^d$, respectively, and consider a bilinear map $\phi : U \times V \to W$. For the rest of this section we consider replicated secret sharing, and we omit the superscript rep from the sharings.

Consider shared values $[\![x_1]\!]_U, \ldots, [\![x_L]\!]_U, [\![y_1]\!]_V, \ldots, [\![y_L]\!]_V$. In this section we describe a protocol to compute $[\![z + \delta]\!]_W$, where $z = \sum_{\ell=1}^{L} \phi(x_\ell y_\ell)$ and $\delta \in W$ is some error known to the adversary. The only building blocks required for this protocol are the following:

- The parties can locally obtain $[\![\phi(\alpha, \beta)]\!]_W^{\mathsf{add}}$ from $[\![\alpha]\!]_U^{\mathsf{rep}}$ and $[\![\beta]\!]_V^{\mathsf{rep}}$. To see this, write $[\![\alpha]\!]_U^{\mathsf{rep}} = ((\alpha_1, \alpha_2), (\alpha_2, \alpha_3), (\alpha_3, \alpha_1))$ and $[\![\beta]\!]_U^{\mathsf{rep}} = ((\beta_1, \beta_2), (\beta_2, \beta_3), (\beta_3, \beta_1))$, where $\alpha = \alpha_1 + \alpha_2 + \alpha_3$ and $\beta = \beta_1 + \beta_2 + \beta_3$. Let $\gamma_i = \phi(\alpha_i, \beta_i) + \phi(\alpha_{i+1}, \beta_i) + \phi(\alpha_i, \beta_{i+1})$, for $i = 1, 2, 3$, which can be computed locally by party P_i. It is easy to see that $\phi(\alpha, \beta) = \gamma_1 + \gamma_2 + \gamma_3$, which completes the claim.
- A protocol for generating random shares $[\![0]\!]_W^{\mathsf{rep}}$. This can be done by generating d random shares $[\![0]\!]_{\mathbb{F}}^{\mathsf{rep}}, \ldots, [\![0]\!]_{\mathbb{F}}^{\mathsf{rep}}$, and setting $[\![0]\!]_W^{\mathsf{rep}} = \sum_{i=1}^{d} [\![0]\!]_{\mathbb{F}}^{\mathsf{rep}} \cdot w_i$. Furthermore, generating each $[\![0]\!]_{\mathbb{F}}^{\mathsf{rep}}$ can be done *non-interactively* by distributing some shared keys among the parties in a setup phase, as shown in [1].
- An interactive protocol for obtaining $[\![w + \delta]\!]_W^{\mathsf{rep}}$ from $[\![w]\!]_W^{\mathsf{add}}$, where $\delta \in W$ is an additive error known to the adversary. If $[\![w]\!]_W^{\mathsf{add}} = (\eta_1, \eta_2, \eta_3)$, this is achieved by letting each P_i send η_i to P_{i+1}, so $[\![w]\!]_W^{\mathsf{rep}} = ((\eta_1, \eta_2), (\eta_2, \eta_3), (\eta_3, \eta_1))$. It is shown in [1] that the only attack the adversary may carry in this protocol is adding an error δ.

Our main protocol is described below.

Protocol $\Pi_{\mathsf{DotProd}}^{\mathsf{rep}}$

Inputs: Shared values $[\![x_1]\!]_U, \ldots, [\![x_L]\!]_U, [\![y_1]\!]_V, \ldots, [\![y_L]\!]_V$.
Output: $[\![z + \delta]\!]_W$, where $z = \sum_{\ell=1}^{L} \phi(x_\ell, y_\ell)$ and $\delta \in W$ is some error known to the adversary.

1. Parties locally compute $[\![\phi(x_\ell, y_\ell)]\!]_W^{\mathsf{add}} \leftarrow \phi([\![x_\ell]\!]_U^{\mathsf{rep}}, [\![y_\ell]\!]_V^{\mathsf{rep}})$, for $\ell = 1, \ldots, L$;
2. Parties sample $[\![0]\!]_W^{\mathsf{add}}$ and then locally compute $[\![z]\!]_W^{\mathsf{add}} = [\![0]\!]_W^{\mathsf{add}} + \sum_{\ell=1}^{L} [\![\phi(x_\ell, y_\ell)]\!]_W^{\mathsf{add}}$.
3. Parties convert $[\![z + \delta]\!]_W^{\mathsf{rep}} \leftarrow [\![z]\!]_W^{\mathsf{add}}$.

B Appendix to Section 6

B.1 Comparison with CHURP.

This dynamic PSS protocol, proposed in [29], is the state of the art in terms of communication complexity. At a high level, CHURP is made of two main protocols, Opt-CHURP, which is able to detect malicious behavior during the proactivization but is not able to point out which party or parties cheated, and Exp-CHURP, which performs proactivization while enabling cheater detection at the expense of being heavier in terms of communication. Since in this work we have described a PSS protocol *with abort*, we compare our protocol against Opt-CHURP.

The protocol Opt-CHURP is comprised of three main subprotocols: Opt-ShareReduce, Opt-Proactivize and Opt-ShareDist. In the first sub-protocol, Opt-ShareReduce, the parties in C distribute shares of their shares towards the parties in C'. A threshold of $2t$ is used for these "two-level" shares to account for the fact that the adversary may control t parties in each committee C and C'. We could avoid such high degree sharing in our $\Pi_{\mathsf{PartialPSS}}$ protocol since there the parties do not share their shares directly. In Opt-ShareReduce, to ensure that a party sends the right share, the parties must also communicate commitments and witnesses for certain polynomial commitment scheme (see [29] for details). The concrete communication complexity of this step is $12Ln^2$ elements, where L is the amount of shared field elements being proactivized.

In the second stage, Opt-Proactivize the parties in C' produce reduced-shares (that is, "shares of shares") of 0 that are added to the reduce-shares of the secret. We will not discuss the details fo this procedure here, beyond mentioning that this requires the parties to exchange shares and proofs in order to ensure the correctness of this method. This incurs a communication complexity of $5Ln^2$ field elements, on top of requiring publishing n hashes on a blockchain, say $256n$ bits using SHA256, which is a requirement that our protocol Π_{PSS} does not have.

In the final stage, Opt-ShareDist, each party in C' sends the reduce-shares of the i-th share to party P'_i, who reconstructs the refreshed share. Again, opening information for certain commitments must be transmitted. This leads to a communication complexity of $2Ln^2$.

Given the above, the total communication complexity of Opt-CHURP is $9Ln^2 \log |\mathbb{F}|$ bits in point-to-point channels, plus $256n$ bits over a blockchain. We see that, asymptotically, our protocol Π_{PSS} represents an improvement factor of around 9, on top of not making use of a blockchain. Furthermore, although not mentioned in our protocol, a lot of the communication that appears in the $13n$ term in our Π_{PSS} protocol can be regarded as preprocessing, that is, it is independent of the message being transmitted and can be computed in advance, before the proactivization phase.

Finally, we note that our novel protocol Π_{PSS} is much more conceptually simple than Opt-CHURP. This is because, unlike in Opt-CHURP, our protocol does not require the expensive use of commitments and proofs at the individual level (i.e. *per party*) in order to ensure correctness of the transmitted value.

Instead, we compute a *global* signature of the secret and check its validity after the proactivization.

B.2 Using AMD Codes.

Finally, we want to discuss a potential optimization of our Π_{PSS} that does not make use of signatures and public-key cryptography in general. The fact that the worst that can happen in the $\Pi_{\mathsf{PartialPSS}}$ protocol is that the transmitted message is wrong by an additive amount known by the adversary implies that other methods to ensure correctness of the transmitted value can be devised. The basic idea, which is highly used in the dishonest majority setting to ensure correct openings, is to use authentication codes. To transmit a secret $[\![s]\!]^{\mathsf{C}}$, the parties in C begin by calling $[\![r]\!]^{\mathsf{C}} \leftarrow \mathcal{F}_{\mathsf{Rand}}$ and then $[\![s \cdot r]\!] \leftarrow \mathcal{F}_{\mathsf{Mul}}([\![s]\!], [\![r]\!])$.[9]. Next, the parties in C send to C' the shares $[\![s+\delta]\!]^{\mathsf{C}'} \leftarrow \Pi_{\mathsf{PartialPSS}}([\![s]\!]^{\mathsf{C}})$ and $[\![s \cdot r + \epsilon]\!]^{\mathsf{C}'} \leftarrow \Pi_{\mathsf{PartialPSS}}([\![s \cdot r]\!]^{\mathsf{C}})$. After these shares are sent, the parties in C open r to each other and then $P_1, \ldots, P_{t+1}$ send r, in the clear, to the parties in C'.[10]

Once every party in C' receives $t+1$ identical copies of r, parties in C' compute $[\![z]\!] = r \cdot [\![s+\delta]\!]^{\mathsf{C}'} - [\![s \cdot r + \epsilon]\!]^{\mathsf{C}'}$, then open z and check that $z \stackrel{?}{=} 0$, and abort if this is not the case. It is easy to see that the check passes if and only if $r \cdot \delta = \epsilon$, which, if $\delta \neq 0$, is satisfied with probability $1/|\mathbb{F}|$ since it is equivalent to $r = \epsilon/\delta$, and r is uniformly random and independent of ϵ and δ. By setting $|\mathbb{F}| \geq 2^{\kappa}$, where κ is the statistical security parameter, we obtain a PSS protocol in which the adversary can succeed in transferring an incorrect message with only negligible probability.

If the message to be transmitted is longer than one field element, say $[\![s_1]\!]^{\mathsf{C}}, \ldots, [\![s_L]\!]^{\mathsf{C}}$, then the protocol can be modified for better efficiency by letting the parties in C call $[\![r_i]\!] \leftarrow \mathcal{F}_{\mathsf{Rand}}$ for $i = 1, \ldots, L$, and then computing $[\![m]\!] \leftarrow \mathcal{F}_{\mathsf{DotProd}}(\{[\![s_i]\!]\}_i, \{[\![r_i]\!]\}_i)$, where $m = \sum_{i=1}^{L} s_i r_i$, that is, $\mathcal{F}_{\mathsf{DotProd}}$ is a functionality that takes the dot product of two shared vectors, which for some protocols can be instantiated at the same communication cost of a single multiplication, as we saw in Section 3. Then, the parties in C send $\{[\![s_i]\!]\}_i$, $[\![m]\!]$ and $\{r_i\}_i$ to the parties in C', who run a similar check to the one above.

This method is in general more efficient than the signature-based method we presented earlier in terms of computation. However, in terms of communication, a downside of this technique is that the long "key" $\{[\![r_i]\!]\}_i$ must be sampled and transmitted each time a new message must be proactivized, whereas, with the signatured-based approach, the secret/public key pair is only sampled once per committee, and it is used to transfer as many secret-shared messages as required. As a result, the signature-based method has 0 overhead with respect

[9] This multiplication may allow additive errors, which allows for a much more efficient protocol [11]

[10] Alternatively, only P_1 may send r while the other parties $P_2, \ldots, P_{t+1}$ only send a hash of this value.

to the length of the message. Unfortunately, as mentioned above, the downside of the signature-based approach is that, although it involves less communication overall than the MAC-based approach, it involves much heavier computation as it is based on elliptic curves, so it may only be benefitial when operating over wide area networks.

C Functionality $\mathcal{F}_{\mathsf{sRand}}$

The functionality $\mathcal{F}_{\mathsf{sRand}}$ used in the secure injective encoding in Section 4.1 has also seen other uses, in particular in connection with secure truncation protocols such as in [14]. $\mathcal{F}_{\mathsf{sRand}}$ can easily be realized with a functionality for generating random bits. To obtain a k bit value r such that its lower ℓ bits are zero, do the following:

1. Sample $k-\ell$ random bits $[\![b_i]\!]$ for $i = 0, \ldots, k-\ell-1$.
2. Each party locally computes $[\![r]\!] = -2^{k-1}b_{k-\ell-1} + 2^{\ell}\sum_{i=0}^{k-\ell-1} 2^i b_i$.

D Protocols Π_{IsSqr} and Π_{Sqrt}

We present here two protocols: One for testing if a number is a square, and another for computing the root of a square number. Note that neither protocol is private if the input is 0. However, for our purposes this is fine as we use them on random values only.

Protocol Π_{IsSqr}

Inputs: $[\![x]\!]$.
Outputs: 1 if x is a quadratic residue modulo p and 0 otherwise.

1. Invoke $[\![b]\!] \leftarrow \mathcal{F}_{\mathsf{Rand}}(\mathbb{F})$ and compute $[\![c]\!] \leftarrow \mathcal{F}_{\mathsf{Mul}}([\![b]\!], [\![b]\!])$.
2. Compute $[\![d]\!] \leftarrow \mathcal{F}_{\mathsf{Mul}}([\![x]\!], [\![c]\!])$ and open d.
3. Compute $d^{(p-1)/2} = x^{(p-1)/2}c^{(p-1)/2}$.
4. If $d \in \{0, -1\}$ output 0. Otherwise ($d = 1$) output 1.

Protocol Π_{IsSqr} has complexity $\mathcal{C}_{\mathsf{IsSqr}} = \mathcal{C}_{\mathsf{Rand}}(1) + \mathcal{C}_{\mathsf{Mul}}(2) + \mathcal{C}_{\mathsf{Open}}(1)$

Lemma 3. *Protocol Π_{IsSqr} secure computes the Legendre symbol x.*

Proof. Since $c = b^2$, its Legendre symbol is 1. Thus the Legendre symbol of d is determined entirely by x. Notice that $b \neq 0$ with probability $1 - 1/|\mathbb{F}|$. As for privacy: Since b is random, $b^2 = c$ is random as well and thus acts as a multiplicative mask of x. Thus revealing d reveals nothing about x, except whether x is a square or not. □

We next show how to compute the square root of a number modulo p. In Π_{Sqrt} below we assume that $p \equiv 3 \pmod 4$ as that allows for an efficient method

of finding y such that $x = y^2 \pmod{p}$, given x. More precisely, given x, we can find y by computing $y = x^{(p+1)/4}$. Observe that $y^2 = (x^{(p+1)/4})^2 = x^{(p+1)/2} = x \cdot x^{(p-1)/2} = x$ since x is a square. (In practice, p is chosen such that it is congruent to 3 modulo 4 for exactly this reason, so our protocol is compatible with all standardized curves.) It remains to figure out how to compute this formula without revealing x, which we do following a similar approach as in Π_{IsSqr}. More precisely, we produce a couple of random values of a specific format and use them as a multiplicative mask on the input. The masked input is then opened, and we compute the square root of the masked value. Finally, the mask is removed, in order to obtain the final result. The values that we need for the mask can be produced using the $\mathcal{F}_{\mathsf{MulTriple}}$ functionality and a trick for computing the inverse of a random element as described in [3].

Protocol Π_{Sqrt}

Inputs: $[\![x]\!]$ where x has a square root.
Outputs: $[\![y]\!]$ such that $y^2 = x$.

OFFLINE PHASE

1. Obtain a random triple $([\![a]\!], [\![b]\!], [\![c = a \cdot b]\!]) \leftarrow \mathcal{F}_{\mathsf{MulTriple}}$.
2. Open c and compute $c^{-1} [\![b]\!] = [\![(a \cdot b)^{-1} b]\!] = [\![a^{-1}]\!]$.
3. Compute $[\![a^2]\!] \leftarrow \mathcal{F}_{\mathsf{Mul}}([\![a]\!], [\![a]\!])$.
4. Store the values $([\![a^2]\!], [\![a^{-1}]\!])$.

ONLINE PHASE

1. Compute $[\![z]\!] \leftarrow \mathcal{F}_{\mathsf{Mul}}([\![x]\!], [\![a^2]\!])$ and open z.
2. Output $[\![y]\!] = z^{(p+1)/4} \cdot [\![a^{-1}]\!]$.

Protocol Π_{Sqrt} computes the square root of its input with complexity $\mathcal{C}_{\mathsf{Sqrt}} = \mathcal{C}_{\mathsf{MulTriple}}(1) + \mathcal{C}_{\mathsf{Mul}}(2) + \mathcal{C}_{\mathsf{Open}}(2)$.

Lemma 4. *Protocol Π_{Sqrt} computes the square root of x securely.*

Proof. Observe that $z^{(p+1)/4} = (xa^2)^{(p+1)/4} = x^{(p+1)/4} a$, and thus we obtain $y = z^{(p+1)/4} a^{-1} = x^{(p+1)/4}$ as desired (as with Π_{IsSqr}, the mask a is non zero with high probability). As for privacy, it suffices to note that a is random and thus acts as a mask for the input, and thus z leaks nothing about x. □