Improved Threshold Signatures, Proactive Secret Sharing and Input Certification from LSS Homomorphisms

Diego F. Aranha[1], Anders Dalskov[2], Daniel Escudero[1], and Claudio Orlandi[1]

[1] Aarhus University, Denmark
[2] Partisia, Denmark

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., ESORICS 2020) that moving from a secret-sharing scheme over $\mathbb{F}_p$ to a secret sharing over an elliptic curve group $\mathbb{G}$ of order p can be done non-interactively by multiplying the share unto a generator of $\mathbb{G}$. We generalize this idea and show that it can also be used to compute arbitrary bilinear maps and in particular pairings over elliptic curves.

We present several practical applications using our techniques: First we show how to securely realize the Pointcheval-Sanders signature scheme (CT-RSA 2016) in MPC. Second we present a construction for dynamic proactive secret-sharing which outperforms the current state of the art from CCS 2019. Third we present a construction for MPC input certification using digital signatures that we show experimentally to outperform the previous best solution in this area.

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 *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 $s \in \mathbb{F}_p$ (here $\mathbb{F}_p$ denotes integers modulo a prime p) is secret-shared by sampling uniformly random $s_1, \ldots, s_n \in \mathbb{F}_p$ subject to $s_1 + \cdots + s_n \equiv s \bmod p$. Another well-known example of a (t, n)-secure LSSS is Shamir secret sharing [38] that distributes a secret $s \in \mathbb{F}_p$ by sampling a random polynomial $f(x)$ over $\mathbb{F}_p$ of degree at most t such that $f(0) = s$, and where the i'th share is defined as $s_i = f(i)$.

Linear secret-sharing schemes are information-theoretic in nature: 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 [23] or secure multiparty computation [12]. Linear secret-sharing schemes can be augmented with techniques from public-key cryptography, such as elliptic-curve cryptography. As an example, consider (a variant of) Feldman's scheme for verifiable secret sharing[3] [20]: To distribute a secret $s \in \mathbb{F}_p$, 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 an elliptic-curve group $\mathbb{G}$ of order p for which the discrete-log problem is hard. Each party can now 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 [29], or to securely compute ECDSA signatures [16,39]. The key idea behind these techniques is that the group $\mathbb{G}$, having order p, is isomorphic to $\mathbb{F}_p$ as an additive group. Although the general idea of using secret sharing "in the exponent" has been used multiple times in the literature, this has been done in a rather ad-hoc way, for specific linear secret-sharing schemes and only for elliptic curves. Thus, a more formal and general treatment of these techniques is currently missing.

1.1 Our Contributions

In this work we expand the range of applications of the techniques mentioned above by considering the case of secure signatures, proactive secret sharing and input certification, providing novel protocols in each of these settings that improve over the state of the art. We also provide experimental results for some of our protocols. Furthermore, along the way we formalize and generalize the idea of "secret sharing in the exponent" by using an adequate mathematical definition of linear secret sharing, 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. Our framework neatly generalizes the techniques used in some prior work like [16,39]. We extend this notion and show how generic multiplication triples over $\mathbb{F}_p$ can be used to securely compute general bilinear maps, of which bilinear *pairings* are a particular case.

The contributions made in this work are summarized below. This listing also serves as an overview of the rest of the paper.

– We introduce the concept of **linear secret-sharing homomorphism** (LSS homomorphisms) which can be seen as a generalization formalization of the idea of "putting the share in the exponent". An adequate mathematical foundation for LSS homomorphisms is presented, and we show how generic multiplication triples can be used to compute securely any bilinear map. This is done in Section 2.

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

- We demonstrate how LSS homomorphisms allow computation of scalar products, thus showing that it generalizes previously used techniques in e.g., [16,39]. We furthermore show that it is possible to use our techniques to compute bilinear pairings over secret-shared data using any secure computation protocol. This is done in Section 2.5, where the first part shows how to compute scalar multiplications and bilinear pairings, and where the second part shows how to instantiate our techniques with various popular secret-sharing schemes.
- To illustrate the usefulness of our LSS homomorphisms, we provide 3 applications. The first of these is a demonstration of how digital signatures can be computed and verified on secret-shared data. This is done in Section 3.
- Our second application demonstrates a protocol for dynamic proactive secret-sharing (PSS). This uses the digital signatures and the result is a dynamic PSS protocol with better communication complexity than the current state of the art. This is done in Section 4.
- Our final application is input certification. We present a method for verifying that a certain party provided input to a secure computation that was previously certified by a trusted party. We benchmark our protocol experimentally and show that it significantly outperforms the previous best solution for input certification for any number of parties. The protocol is presented in Section 5, and our experiments are presented in Section 6.

1.2 Related Work

Previous works [16,39] 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. Also, in [19] the authors present protocols to securely compute over elliptic curves (and also over lattices). The authors consider key generation of elliptic-curve ElGamal, as well as decryption, based on generic MPC protocols. In addition, a protocol for solving the discrete log of a secret-shared value is presented. We present an alternative to such an decoding scheme in Appendix D.2 which can be seen as complimentary to their approach.

In [11] the authors construct protocols for multiplying matrices and other bilinear operations such as convolutions based on the observation that the widely used Beaver multiplication technique [6] extends to these operations as well. This turns out to be a particular instantiation of our framework from Section 2 when the vector spaces are instantiated with matrix spaces and the bilinear map is instantiated with matrix product.

Multiple works have addressed the problem of proactive secret-sharing. It was originally proposed in [28,34], and several works have built on top of these techniques [27,37,5,4,32], including ours. Among these, the closest to our work is the state-of-the-art [32], 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 4.

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 [8,30,41,9]. Among these, only [9] tackles the problem from a more general perspective, having multiple parties and different protocols. In [9], 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. The authors of [9] present constructions of this type of signatures, and use them to solve the input certification problem. However, the techniques from [9] differ from ours at a fundamental level: Their protocols first computes 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 [18] and one from [17]. 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, or special verification protocol are needed.

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 $\boldsymbol{e}_A^{\mathsf{T}}(M_A\boldsymbol{r}_v) = v$.*

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 $[\![\phi(v), \phi(\boldsymbol{r}_v)]\!]_U$.*[4]

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

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 maps. More precisely, let U, V, W be $\mathbb{F}$-vector spaces of dimension d,[5] 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.

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" [6] to the context of bilinear maps. At a high level, the parties 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

$$\begin{aligned}\phi(\delta, \epsilon) + \phi(\delta, [\![\beta]\!]_V) + \phi([\![\alpha]\!]_U, \epsilon) + [\![\phi(\alpha, \beta)]\!]_W &= [\![\phi\left(\delta + \alpha, \epsilon + \beta\right)]\!]_W \\ &= [\![\phi\left(u, v\right)]\!]_W .\end{aligned}$$

Appendix A formalizes this intuition and defines a protocol Π_{bilinear} parameterized by the map ϕ, which takes as input $[\![u]\!]_U$, $[\![v]\!]_V$ and outputs $[\![w]\!]_W$ with $w = \phi(u, v)$.

2.5 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. 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 $(P_1, \dots, P_t)$, computes $(Q_1, \dots, Q_m)^\intercal = M \cdot (P, P_1, \dots, P_t)^\intercal \in \mathbb{G}^m$, 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

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

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

each party can locally multiply its share by the generator G to obtain $[\![s \cdot G]\!]_{\mathbb{G}}$. By instantiating the secret-sharing scheme with popular constructions such as additive or Shamir secret-sharing, we obtain different techniques used in previous works in the literature, as cited in the introduction.

Now, by choosing different bilinear maps we also obtain some techniques used in previous works, such as [16,39]. 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 $\Pi_{\mathsf{ScalarMul}}$ (more precisely, $\Pi_{\mathsf{ScalarMul}}$ is a special case of Π_{Bilinear} when the LSS homomorphism is f and the dimensions of the inputs are 1), 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 [39] and as such our presentation here can be considered as illustrating that Π_{Bilinear} generalizes the techniques in their work. We assume access to a triple preprocessing 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 denote generators of $\mathbb{G}_1$ and $\mathbb{G}_2$, respectively.

Protocol $\mathit{\Pi}_{\mathsf{pairing}}$

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 Threshold Signature Schemes

In this section we show how our techniques 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 $\mathit{\Pi}_{\mathsf{Keygen}}$ 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 $\mathit{\Pi}_{\mathsf{Sign}}$ protocol that on input a secret shared message $[\![m]\!]$ and $[\![sk]\!]$ output from $\mathit{\Pi}_{\mathsf{Keygen}}$ outputs $[\![\sigma]\!]$ where σ is a signature on m under sk. Finally, we present a verification protocol $\mathit{\Pi}_{\mathsf{Verify}}$ 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 use the signature scheme [36] by Pointcheval and Sanders (henceforth PS) 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.[6]. 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.

[6] A downside of e.g., ECDSA signatures is that messages have to be hashed first, which creates a significant problem when messages are secret-shared, as hashing secret-shared data is quite expensive

Primitives for MPC. For this section, and for the rest of the paper, we will rely on the existence of several functionalities to securely compute on secret-shared data. 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 $(\llbracket a \rrbracket, \llbracket b \rrbracket, \llbracket c \rrbracket)$ where $c = ab$.
- $\mathcal{F}_{\mathsf{DotProd}}$ takes as input $(\llbracket x_i \rrbracket)_{i=1}^{L}$ and $(\llbracket y_i \rrbracket)_{i=1}^{L}$, and produces $\llbracket z \rrbracket$, where $z = \sum_{\ell=1}^{L} \phi(x_\ell y_\ell)$.
- $\mathcal{F}_{\mathsf{Mul}}$ takes two inputs $\llbracket x \rrbracket$ and $\llbracket y \rrbracket$, and outputs $\llbracket w \rrbracket$ 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 $\llbracket x \rrbracket$ 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.

The functionalities above are defined irrespectively of whether the adversary is passive (that is, it respect the protocol specification) or active (the adversary may deviate arbitrarily).[7] The following functionality only makes sense for settings with active security.

- $\mathcal{F}_{\mathsf{DotProd*}}$ takes as input $(\llbracket x_i \rrbracket)_{i=1}^{L}$ and $(\llbracket y_i \rrbracket)_{i=1}^{L}$, and produces $\llbracket z + \delta \rrbracket$, where $z = \sum_{\ell=1}^{L} \phi(x_\ell y_\ell)$ and $\delta \in \mathbb{F}$ is an error provided by the adversary.

The reason to consider this dot product functionality, which produces incorrect results, is that (1) for some secret-sharing schemes this functionality can be instantiated with a communication complexity that is independent of the length L, and (2) that it suffices for some of the applications we consider later on. How these functionalities are instantiated depends naturally on the choice of secret-sharing scheme. We discuss instantiations for popular secret-sharing schemes, including the ones we will focus on what follows (additive and Shamir secret-sharing), in Section B in the Appendix.

3.1 The PS Signature Scheme

The PS signature scheme 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.

[7] One caveat is that the shares on their own may not define the secret if the adversary is allowed to change the corrupt parties' shares, which is the case for an active adversary. This is an issue for example for additive secret sharing with a dishonest majority (which can be fixed by adding homomorphic MACs), but not for Shamir secret sharing with an honest majority. We discuss this in detail in Section B in the Appendix.

– $\mathsf{Keygen}(pp)$: 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.

3.2 Threshold PS Signatures

The Π_{Keygen} protocol presented below shows how to generate keys suitable for signing messages of r blocks. The protocol proceeds as follows: 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 (assumed to be generated by a $\mathcal{F}_{\mathsf{Input}}$ functionality) given the tools we have described so far. The Π_{Sign} protocol below outputs a signature $(\sigma_1, [\![\sigma_2]\!]_{\mathbb{G}_1})$. The reasons for keeping σ_1 public are (1) that it simplifies things when we use this later, and (2) makes signing more efficient. If, however, σ_1 cannot be revealed then Π_{Pairing} is needed for step 3.

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 $[\![z]\!] \leftarrow \mathcal{F}_{\mathsf{DotProd}}(([\![y_i]\!])_{i=1}^{r}, ([\![m_i]\!])_{i=1}^{r})$ and then compute $[\![w]\!] = [\![x]\!] + [\![z]\!]$.

3. Parties use the LSS homomorphism $x \mapsto x \cdot \sigma_1$ to compute locally $[\![\sigma_2]\!]_{\mathbb{G}_1} \leftarrow \mathit{\Pi}_{\mathsf{ScalarMul}}([\![w]\!], \sigma_1)$.
4. Output $(\sigma_1, [\![\sigma_2]\!]_{\mathbb{G}_1})$.

Protocol $\mathit{\Pi}_{\mathsf{Sign}}$ produces a correct signature with communication complexity $\mathcal{C}_{\mathsf{Coin}}(1) + \mathcal{C}_{\mathsf{DotProd}}(r)$.

Finally, we show a verification protocol $\mathit{\Pi}_{\mathsf{Verify}}$ in which a secret-shared $\mathbb{G}_T$ element $[\![b]\!]_{\mathbb{G}_T}$ where $b = 1_{\mathbb{G}_T}$ if an only if the signature was valid. While this is not a bit, it nevertheless carries the same information. Below the signature we verify is $(\sigma_1, [\![\sigma_2]\!]_{\mathbb{G}_1})$, however if this is not the case (in particular, if σ_1 is secret-shared) then $\mathit{\Pi}_{\mathsf{Pairing}}$ is needed in step 4.

Protocol $\mathit{\Pi}_{\mathsf{Verify}}$

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: $[\![b]\!]_{\mathbb{G}_T} = [\![1_{\mathbb{G}_T}]\!]$ if $\mathsf{Verify}(\mathsf{pk}, \mathbf{m}, \sigma) = 0$ and a random value otherwise.

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 $[\![b]\!]_{\mathbb{G}_T} \leftarrow \mathit{\Pi}_{\mathsf{ScalarMul}}([\![\rho]\!], [\![\alpha]\!]_{\mathbb{G}_T} / [\![\beta]\!]_{\mathbb{G}_T})$ where $[\![\rho]\!]$ was obtained by invoking $\mathcal{F}_{\mathsf{Rand}}$.

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

Lemma 1. *Protocol $\mathit{\Pi}_{\mathsf{Verify}}$ outputs a secret-sharing of $0_{\mathbb{G}_T}$ 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.

It is likewise possible to see that any successful attack on $(\mathit{\Pi}_{\mathsf{Keygen}}, \mathit{\Pi}_{\mathsf{Sign}}, \mathit{\Pi}_{\mathsf{Verify}})$ can easily be turned into an attack on the original PS signature scheme, in particular on the EUF-CMA [25] property of the PS signature scheme.

We consider an ideal threshold signature functionality roughly equivalent to the $\mathcal{F}_{\mathsf{tsig}}$ functionality presented in[10], the main difference being that we do not consider key refreshment. It is possible to show that $\mathit{\Pi}_{\mathsf{PS}} = (\mathit{\Pi}_{\mathsf{Keygen}}, \mathit{\Pi}_{\mathsf{Sign}}, \mathit{\Pi}_{\mathsf{Verify}})$ securely realizes this functionality

The $\mathcal{F}_{\mathsf{tsig}}$ functionality records a message as signed once it has received a sign request from $t+1$ parties. During verification, $\mathcal{F}_{\mathsf{tsig}}$ receives a tuple (m, σ, pk) and does one of three things: If (m, σ, b) was previously recorded, then b is returned (that is, the signature was previously verified and b was the result); If m was never signed, then $b = 0$ is returned, and if (m, σ) was not previously verified but m was signed, then $b = \mathsf{Verify}(\mathsf{pk}, m, \sigma)$ is returned.

Importantly, distinguishing between Π_{PS} and $\mathcal{F}_{\mathsf{tsig}}$ happens only if the adversary manages to input a pair (m, σ, pk) such that m was never signed, but $1 = \mathsf{Verify}(\mathsf{pk}, m, \sigma)$. However, this corresponds *precisely* to breaking the EUF-CMA property of the PS signature scheme.

4 Applications to Proactive Secret Sharing

Secret-sharing allows a dealer to distribute a secret such that an adversary with only access to some subset of the shares cannot learn anything about the secret. However as time passes it becomes harder to argue that no leakage beyond this subset takes place, and thus that the secret remains hidden from the adversary. Proactive Secret-sharing (PSS) deals with this problem by periodically "refreshing" (or *proactivizing*) shares such that shares between two proactivization stages become "incompatible".

Typically, the case of interest in the PSS setting is honest majority, since in this case the value of the underlying secret is determined by the shares from the honest parties only. In this section we focus on Shamir secret-sharing, as described in Section B.2 in the Appendix, and we denote such sharings by $[\![\cdot]\!]$. We assume that $2t + 1 = n$. Multiple PSS schemes have been proposed for this case, but for the special situation of *dynamic PSS* (a PSS scheme is dynamic if the number of parties and threshold can change between each proactivization), CHURP is presented in [32]. In a nutshell, CHURP first performs an optimistic proactivization and, if cheating is detected, falls back to a slower method that is able to detect cheaters.

In what follows we show how to use the protocols for signatures developed in Section 3 to obtain a conceptually simple and efficient dynamic PSS with abort. We first 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 signature scheme, an adversary cannot make the receiving committee accept an incorrectly transmitted message. This construction leads to a 9-fold improvement in terms of communication with respect to the optimistic protocol from [32].

We say that the parties have *consistent sharings* of a secret x if each P_i knows a value s_i such that there exists a polynomial $f(x)$ of degree at most t with $f(i) = s_i$ and $f(0) = s$.

4.1 Proactive Secret Sharing

We present here 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 [32].

In a PSS scheme a set of n parties have consistent Shamir shares of a secret $[\![s]\!] = (s_1, \ldots, s_n)$ with 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 satisfies:

– *(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.

4.2 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 $\mathcal{U}' = \{P'_i\}_{i=1}^{t+1}$. As mentioned before, we consider Shamir secret-sharing, as defined in Section B.2, with threshold $t < n/2$. This ensures that the corrupt parties cannot modify their shares without resulting in an error, thanks to error-detection, as discussed in Section B.2 in the Appendix. 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 $\Pi_{\mathsf{PartialPSS}}([\![s]\!]^{\mathsf{C}})$

Inputs A shared value $[\![s]\!]^{\mathsf{C}} = (s_1, \ldots, s_n)$ among a committee C.

Output: Either a consistently shared value $[\![s']\!]^{\mathsf{C}'}$ 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. *Assume that initially the parties in C had consistent shares of a secret s. Then the protocol results in the parties in C' having consistent shares of $s + \delta$, where δ is an additive error known by the adversary.*

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 $\gamma_j = \epsilon_j + \delta_j$.

Now we are ready to argue consistency of the final sharings. The honest parties $P'_j \in \mathsf{C}' \setminus \mathcal{A}'$ output the sharings $s'_j = \gamma_j + h(j)$. On the other hand, the adversary knows all γ_i, so we can re-define the shares $s'_j \leftarrow s'_j - \gamma_j + q(j)$ for $P'_j \in \mathcal{A}'$, where $q(j) \in \mathbb{F}_{\leq t}[x]$ is such that $q(i) = \gamma_i$ for $P_i \in \mathsf{C}' \setminus \mathcal{A}'$.[8] This way the sharings $(s'_1, \ldots, s'_j)$ are consistent with the polynomial $h(x) + q(x) \in \mathbb{F}_{\leq t}[x]$, whose underlying secret is $f(0) + g(0) + q(0) = s + 0 + q(0) = s + \delta$.

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

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

The two views are perfectly indistinguishable: $\{s_{ij}\}_{P_i \in \mathsf{C}, P_j \in \mathcal{U} \cap \mathcal{A}} \equiv \{\mathsf{s}_{ij}\}_{P_i \in \mathsf{C}, P_j \in \mathcal{U} \cap \mathcal{A}}$ because, given that $|\mathcal{U} \cap \mathcal{A}| \leq t < t + 1$, in the real execution the honest parties

[8] Here we are using the fact that $n = 2t + 1$ rather than the more general $n \geq 2t + 1$.

$P_i \in \mathsf{C} \setminus \mathcal{A}$ sample $\{s_{ij}\}_{P_j \in \mathcal{U} \cap \mathcal{A}}$ independently and uniformly at random, like in the simulation. Also $\{a_{ij}\}_{P_i \in \mathcal{U}, P'_j \in \mathcal{A}'} \equiv \{\mathsf{a}_{ij}\}_{P_i \in \mathcal{U}, P'_j \in \mathcal{A}'}$ 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. $\square$

Extending to group elements. $\mathit{\Pi}_{\mathsf{PartialPSS}}$ can be extended to proactivize shares $[\![\alpha]\!]_{\mathbb{G}}^{\mathsf{C}}$, where $\mathbb{G}$ is an elliptic curve group by running the same protocol "in the exponent". More formally, the LSS homomorphism $x \mapsto x \cdot G$, where G is a generator of $\mathbb{G}$, is used. This will be used later on in our protocol. Finally, observe that $\mathit{\Pi}_{\mathsf{PartialPSS}}$ communicates a total of $n(n+1)$ field elements.

4.3 Simple and Efficient PSS with Abort

The protocol $\mathit{\Pi}_{\mathsf{PartialPSS}}$ 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. More precisely, a malicious party may disrupt the output as $[\![s+\gamma]\!]^{\mathsf{C}'} \leftarrow \mathit{\Pi}_{\mathsf{PartialPSS}}([\![s]\!]^{\mathsf{C}})$, where γ is some value known by the adversary. This is of course not ideal, but it can be fixed by making use of the signature protocols proposed in Section 3. In a nutshell, the committee C uses $\mathit{\Pi}_{\mathsf{PartialPSS}}$ 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 $\mathit{\Pi}_{\mathsf{PSS}}$ below. The setup regarding secret/public key pairs is also presented in the protocol.

Protocol $\mathit{\Pi}_{\mathsf{PSS}}([\![s]\!]^{\mathsf{C}})$

Inputs: A shared value $[\![s]\!]^{\mathsf{C}} = (s_1, \dots, s_n)$ 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'. This can be easily generated by using protocol $\mathit{\Pi}_{\mathsf{Keygen}}$ from Section 3.

1. Parties in C call $(\sigma_1, [\![\sigma_2]\!]^{\mathsf{C}}) \leftarrow \mathit{\Pi}_{\mathsf{Sign}}([\![\mathsf{sk}_{\mathsf{C}}]\!]^{\mathsf{C}}, [\![s]\!]^{\mathsf{C}})$.
2. Parties in $\mathsf{C} \cup \mathsf{C}'$ call $[\![s']\!]^{\mathsf{C}'} \leftarrow \mathit{\Pi}_{\mathsf{PartialPSS}}([\![s]\!]^{\mathsf{C}})$ and $[\![\sigma'_2]\!]^{\mathsf{C}'} \leftarrow \mathit{\Pi}_{\mathsf{PartialPSS}}([\![\sigma_2]\!]^{\mathsf{C}})$.
3. $P_1, \dots, P_{t+1}$ 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 $[\![v]\!]^{\mathsf{C}'} \leftarrow \mathit{\Pi}_{\mathsf{Verify}}([\![s']\!]^{\mathsf{C}'}, (\sigma_1, [\![\sigma'_2]\!]^{\mathsf{C}'}), \mathsf{pk}_{\mathsf{C}})$ and open v using error detection. If $v = 0_{\mathbb{G}_T}$ then the parties in C' output $[\![s']\!]^{\mathsf{C}'}$. Else, they abort.

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 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, but essentially follows from the fact that decision to abort can be shown to *independent* of the secret (thus ruling out a selective failure attack). Put differently, a decision depends only on the error introduced by the adversary which is independent of the secret.

We summarize these properties in Theorem 2 below. In our proof we do not reduce to the unforgeability of the signature scheme, but instead to a hard problem over elliptic curves directly. This is easier and cleaner in our particular setting, given that the signatures are produced and checked within the same protocol. The computational problem we reduce the security of Protocol $\mathit{\Pi}_{\mathsf{PSS}}$ 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 (see [21]), we can prove the following about the security of $\mathit{\Pi}_{\mathsf{PSS}}$.

Theorem 2. *Protocol $\mathit{\Pi}_{\mathsf{PSS}}$ instantiates the PSS-with-abort functionality described in Section 4.1, that is, if the parties do not abort in the protocol $\mathit{\Pi}_{\mathsf{PSS}}$, 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 $\sigma_2' = \sigma_2 + \gamma$, where $\delta \in \mathbb{F}$ and $\gamma \in \mathbb{G}_1$ are the errors introduced by the adversary in the $\mathit{\Pi}_{\mathsf{PartialPSS}}$ 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 $\mathit{\Pi}_{\mathsf{Verify}}([\![s']\!]^{\mathsf{C}'}, (\sigma_1, [\![\sigma_2']\!]^{\mathsf{C}'}), \mathsf{pk}_{\mathsf{C}})$ 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. □

Although we did not address this in our security arguments, the setup needed for the protocol Π_{PSS}, namely that the parties in C have a shared secret-key for which the parties in C' know the corresponding public key, can be reused for multiple successful proactivizations. Intuitively, this holds because, if the adversary cheats in the proactivization, Theorem 2 shows that this is detected with overwhelming probability, and if the adversary does not cheat then no extra information about the secret-key from the committee C is leaked to the adversary.

Communication Complexity. The communication complexity of the Π_{PSS} protocol is $\mathcal{C}_{\mathsf{PartialPSS}}(L+1) + \mathcal{C}_{\mathsf{Sign}}(L) + \mathcal{C}_{\mathsf{Verify}}(L)$. We ignore the opening of $[\![v]\!]$ at the end as this is independent of 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 [18] 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 B.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. The dynamic PSS protocol proposed in [32], is to our knowledge 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 requiring more 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,[9] so our novel method

[9] For a more detailed derivation of this complexity, see Section C in the appendix.

presents a 9-fold improvement over the state of the art. 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 conceptually much more simple than Opt-CHURP. 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.

Optimizations. 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 3 allows for cheap signing and verification of long messages without penalty in communication.

Also, as we noted in Section 3.2, 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 $\mathcal{F}_{\mathsf{DotProd}}$.

Additionally, 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, like the MACs described in Section B.1 in the Appendix for additive secret-sharing. 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.

5 Applications to Input Certification

MPC 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 classic *millionaires problem* [40] 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 prohibitively expensive in MPC. In this section we show that by using our secure PS signatures from Section 3, this approach is not longer infeasible, and in fact, it is quite efficient.

5.1 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 $[\![x_{i1}]\!], \ldots, [\![x_{iL}]\!]$. We also assume the existence of a broadcast channel.

Our protocol, $\Pi_{\mathsf{CertInput}}$, allows a party P_i to distribute shares of its input, only if this input has been previously certified.

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

Input: Index $i \in \{1, \ldots, n\}$ and $\big((x_i)_{i=1}^L, \sigma_1, \sigma_2\big)$ 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 $\Big(([\![x_i]\!])_i, [\![\sigma_2]\!]_{\mathbb{G}_1}\Big)$. 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.

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.

Security. Security of $\Pi_{\mathsf{CertInput}}$ follows immediately from the security of the protocols presented in Section 3. Indeed, if a corrupt P_j sends an incorrect share to an *honest* party, then that directly corresponds to creating a forgery in the PS signature scheme.

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 $1/|\mathbb{G}_T|$.

Comparison with [9]. Certifying inputs for MPC with the help of signatures has been studied previously in [9]. 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 [9] depend on the underlying secret-sharing scheme used, and two ad-hoc constructions, one for Shamir secret-sharing (using the MPC protocol from [18]) and another one for additive secret sharing (using the MPC protocol from [17]), 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 6.1 a more experimental and quantitative comparison between our work and [9]. We observe that, in general, our approach is at least 2 times more efficient.

6 Implementation and Benchmarking

We implemented our protocols with the RELIC toolkit [2] using the 128-bit-secure pairing-friendly BLS12-381 curve. This curve has embedding degree $k = 12$ and a 255-bit prime-order subgroup, and became popular after it was adopted by the ZCash cryptocurrency [7]. 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, 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 $\mathbb{G}_T$ [31] and is supported by the analysis in [33]. Our implementation makes use of all optimizations implemented in RELIC, including Intel 64-bit Assembly acceleration, and extend the supported algorithms to allow computation of arbitrarily-sized linear combinations of $\mathbb{G}_2$ points through Pippenger's algorithm. 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 will be contributed back to the library.

We benchmarked our implementation on an Intel Core i7-7820X Skylake CPU clocked at 3.6GHz with HyperThreading and TurboBoost turned off to

Operation		Local (cc)	Two-party (cc)
Scalar multiplication in $\mathbb{G}_1$		386	612
Scalar multiplication in $\mathbb{G}_2$		1,009	1,796
Exponentiation in $\mathbb{G}_T$		1,619	2,772
Pairing computation		3,107	4,063
PS key generation	(1 msg)	2,670	4,723
PS signature computation	(1 msg)	626	654
PS signature verification	(1 msg)	5,153	8,065
PS key generation	(10 msgs)	11,970	23,464
PS signature computation	(10 msgs)	656	668
PS signature verification	(10 msgs)	10,144	12,953

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.

	Number of messages						
	1	10	10^2	10^3	10^4	10^5	10^6
Ours	8,065	12,953	62,714	357,445	2,334,742	22,281,049	220,572,619
[9]	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 5 and the one presented in [9]. Numbers are measured in thousands of clock cycles (cc).

reduce noise in the benchmarks. Each procedure was executed 10^4 times and the averages are 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 1.59 to 1.78, while pairing computation becomes only 30% slower. For the PS protocol, key generation and signature verification in MPC are penalized in comparison to local computation by less than 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.

6.1 Certified Inputs

Here we compare our protocol for input certification from Section 5 with the experimental results reported in [9]. To perform a fair comparison, we converted the timings from the second half of Table 2 in [9] 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 number of messages. We used as reference the largest running time of the two running parties (input provider and other party) reported in [9], since the computation would be bounded by the maximum running time. Our results are shown in Table 2, and show that our implementations are already faster for small numbers of messages, but improve on related work by a factor of 2–5 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 different enough in Sandy Bridge to explain the difference, and just converting performance figures to clock cycles makes the results generally comparable.

References

1. T. Araki, J. Furukawa, et al. High-throughput semi-honest secure three-party computation with an honest majority. In E. R. Weippl, S. Katzenbeisser, et al., eds., *ACM CCS 2016*, pp. 805–817. ACM Press, Oct. 2016.
2. D. F. Aranha, C. P. L. Gouvêa, et al. 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, ed., *8th ACM PODC*, pp. 201–209. ACM, Aug. 1989.
4. J. Baron, K. El Defrawy, et al. How to withstand mobile virus attacks, revisited. In M. M. Halldórsson and S. Dolev, eds., *33rd ACM PODC*, pp. 293–302. ACM, Jul. 2014.
5. —. Communication-optimal proactive secret sharing for dynamic groups. In T. Malkin, V. Kolesnikov, et al., eds., *ACNS 15*, vol. 9092 of *LNCS*, pp. 23–41. Springer, Heidelberg, Jun. 2015.
6. D. Beaver. Efficient multiparty protocols using circuit randomization. In J. Feigenbaum, ed., *CRYPTO'91*, vol. 576 of *LNCS*, pp. 420–432. Springer, Heidelberg, Aug. 1992.
7. E. Ben-Sasson, A. Chiesa, et al. Zerocash: Decentralized anonymous payments from bitcoin. In *2014 IEEE Symposium on Security and Privacy*, pp. 459–474. IEEE Computer Society Press, May 2014.
8. 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.
9. M. Blanton and M. Jeong. Improved signature schemes for secure multi-party computation with certified inputs. In J. López, J. Zhou, et al., eds., *ESORICS 2018, Part II*, vol. 11099 of *LNCS*, pp. 438–460. Springer, Heidelberg, Sep. 2018.
10. R. Canetti, N. Makriyannis, et al. Uc non-interactive, proactive, threshold ecdsa. Cryptology ePrint Archive, Report 2020/492, 2020. `https://eprint.iacr.org/2020/492`.
11. H. Chen, M. Kim, et al. Maliciously secure matrix multiplication with applications to private deep learning. Cryptology ePrint Archive, Report 2020/451, 2020. `https://eprint.iacr.org/2020/451`.

12. R. Cramer, I. Damgård, et al. General secure multi-party computation from any linear secret-sharing scheme. In B. Preneel, ed., *EUROCRYPT 2000*, vol. 1807 of *LNCS*, pp. 316–334. Springer, Heidelberg, May 2000.
13. R. Cramer, I. B. Damgård, et al. *Secure multiparty computation.* Cambridge University Press, 2015.
14. A. Dalskov, D. Escudero, et al. Secure evaluation of quantized neural networks. *Proceedings on Privacy Enhancing Technologies*, 2020(4):355 – 375, 01 Oct. 2020.
15. —. Secure evaluation of quantized neural networks. Cryptology ePrint Archive, Report 2019/131, 2019. `https://eprint.iacr.org/2019/131`.
16. A. Dalskov, M. Keller, et al. Securing dnssec keys via threshold ecdsa from generic mpc. In *Computer Security - ESORICS 2020 - 25th European Symposium on Research in Computer Security, United Kingdom, September 14-18, 2020.* 2020.
17. I. Damgård, M. Keller, et al. Practical covertly secure MPC for dishonest majority - or: Breaking the SPDZ limits. In J. Crampton, S. Jajodia, et al., eds., *ESORICS 2013*, vol. 8134 of *LNCS*, pp. 1–18. Springer, Heidelberg, Sep. 2013.
18. I. Damgård and J. B. Nielsen. Scalable and unconditionally secure multiparty computation. In A. Menezes, ed., *CRYPTO 2007*, vol. 4622 of *LNCS*, pp. 572–590. Springer, Heidelberg, Aug. 2007.
19. B. H. Falk and D. Noble. Secure computation over lattices and elliptic curves. Cryptology ePrint Archive, Report 2020/926, 2020. `https://eprint.iacr.org/2020/926`.
20. P. Feldman. A practical scheme for non-interactive verifiable secret sharing. In *28th FOCS*, pp. 427–437. IEEE Computer Society Press, Oct. 1987.
21. D. Fiore and R. Gennaro. Publicly verifiable delegation of large polynomials and matrix computations, with applications. In T. Yu, G. Danezis, et al., eds., *ACM CCS 2012*, pp. 501–512. ACM Press, Oct. 2012.
22. P.-A. Fouque, A. Joux, et al. Injective encodings to elliptic curves. In C. Boyd and L. Simpson, eds., *ACISP 13*, vol. 7959 of *LNCS*, pp. 203–218. Springer, Heidelberg, Jul. 2013.
23. J. A. Garay, R. Gennaro, et al. Secure distributed storage and retrieval. *Theoretical Computer Science*, 243(1-2):363–389, 2000.
24. P. Gemmell and M. Sudan. Highly resilient correctors for polynomials. *Information processing letters*, 43(4):169–174, 1992.
25. S. Goldwasser, S. Micali, et al. A digital signature scheme secure against adaptive chosen-message attacks. *SIAM Journal on Computing*, 17(2):281–308, Apr. 1988.
26. 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`.
27. A. Herzberg, M. Jakobsson, et al. Proactive public key and signature systems. In R. Graveman, P. A. Janson, et al., eds., *ACM CCS 97*, pp. 100–110. ACM Press, Apr. 1997.
28. A. Herzberg, S. Jarecki, et al. Proactive secret sharing or: How to cope with perpetual leakage. In D. Coppersmith, ed., *CRYPTO'95*, vol. 963 of *LNCS*, pp. 339–352. Springer, Heidelberg, Aug. 1995.
29. A. Kate, G. M. Zaverucha, et al. Constant-size commitments to polynomials and their applications. In M. Abe, ed., *ASIACRYPT 2010*, vol. 6477 of *LNCS*, pp. 177–194. Springer, Heidelberg, Dec. 2010.
30. J. Katz, A. J. Malozemoff, et al. Efficiently enforcing input validity in secure two-party computation. Cryptology ePrint Archive, Report 2016/184, 2016. `http://eprint.iacr.org/2016/184`.

31. T. Kim and R. Barbulescu. Extended tower number field sieve: A new complexity for the medium prime case. In M. Robshaw and J. Katz, eds., *CRYPTO 2016, Part I*, vol. 9814 of *LNCS*, pp. 543–571. Springer, Heidelberg, Aug. 2016.
32. S. K. D. Maram, F. Zhang, et al. CHURP: Dynamic-committee proactive secret sharing. In L. Cavallaro, J. Kinder, et al., eds., *ACM CCS 2019*, pp. 2369–2386. ACM Press, Nov. 2019.
33. A. Menezes, P. Sarkar, et al. Challenges with assessing the impact of NFS advances on the security of pairing-based cryptography. In *Mycrypt*, vol. 10311 of *LNCS*, pp. 83–108. Springer, 2016.
34. R. Ostrovsky and M. Yung. How to withstand mobile virus attacks (extended abstract). In L. Logrippo, ed., *10th ACM PODC*, pp. 51–59. ACM, Aug. 1991.
35. C. Peikert. On error correction in the exponent. In S. Halevi and T. Rabin, eds., *TCC 2006*, vol. 3876 of *LNCS*, pp. 167–183. Springer, Heidelberg, Mar. 2006.
36. D. Pointcheval and O. Sanders. Short randomizable signatures. In K. Sako, ed., *CT-RSA 2016*, vol. 9610 of *LNCS*, pp. 111–126. Springer, Heidelberg, Feb. / Mar. 2016.
37. D. A. Schultz, B. Liskov, et al. Mobile proactive secret sharing. In R. A. Bazzi and B. Patt-Shamir, eds., *27th ACM PODC*, p. 458. ACM, Aug. 2008.
38. A. Shamir. How to share a secret. *Communications of the Association for Computing Machinery*, 22(11):612–613, Nov. 1979.
39. N. P. Smart and Y. Talibi Alaoui. Distributing any elliptic curve based protocol. In M. Albrecht, ed., *17th IMA International Conference on Cryptography and Coding*, vol. 11929 of *LNCS*, pp. 342–366. Springer, Heidelberg, Dec. 2019.
40. A. C.-C. Yao. Protocols for secure computations (extended abstract). In *23rd FOCS*, pp. 160–164. IEEE Computer Society Press, Nov. 1982.
41. Y. Zhang, M. Blanton, et al. Enforcing input correctness via certification in garbled circuit evaluation. In S. N. Foley, D. Gollmann, et al., eds., *ESORICS 2017, Part II*, vol. 10493 of *LNCS*, pp. 552–569. Springer, Heidelberg, Sep. 2017.

A Bilinear maps for MPC

We formalize the intuition from Section 2.4 below where we describe the protocol Π_{bilinear} in detail.

For this protocol we assume a functionality $\mathcal{F}_{\mathsf{OuterProd}}$ that produce random shares $[\![a_1]\!], \dots, [\![a_d]\!], [\![b_1]\!], \dots, [\![b_d]\!]$ over $\mathbb{F}$, together with $[\![a_i b_j]\!]$ for $i, j \in \{1, \dots, d\}$. This is used to produce the "bilinear triples" mentioned earlier. (Notice further that the case where $d = 1$, $\mathcal{F}_{\mathsf{OuterProd}}$ corresponds to a classical triple-preprocessing functionality.) Also, in the protocol below we assume that $\{u_1, \dots, u_d\}$ is a basis for U and that $\{v_1, \dots, v_d\}$ 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 $[\![\phi(u, v)]\!]_W = \phi(\delta, \epsilon) + [\![\phi(\delta, \beta)]\!]_W + [\![\phi(\alpha, \epsilon)]\!]_W + [\![\phi(\alpha, \beta)]\!]_W$.

B Some Linear Secret Sharing Schemes

B.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 $(s+\delta)\llbracket r \rrbracket^{\mathsf{add}} - \llbracket r \cdot s \rrbracket^{\mathsf{add}}$, 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 $\llbracket s \cdot G \rrbracket^{\mathsf{add}}_{\mathbb{G}}$ are accompanied by $\llbracket r \cdot s \cdot G \rrbracket^{\mathsf{add}}_{\mathbb{G}}$, where r is a *global* uniformly random value that is also shared as $\llbracket r \rrbracket^{\mathsf{add}}$. At reconstruction time $\llbracket s \cdot G \rrbracket^{\mathsf{add}}_{\mathbb{G}}$ can be opened to $(s+\delta)\cdot G$, and to ensure $\delta = 0$ the parties open $\llbracket r \rrbracket^{\mathsf{add}}_{\mathbb{G}} \cdot (s+\delta)\cdot G - \llbracket r \cdot s \cdot G \rrbracket^{\mathsf{add}}_{\mathbb{G}}$ 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 $\delta \neq 0$.

B.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.[10] We denote $\llbracket s \rrbracket^{\mathsf{shm}}_{\mathbb{F}} = (f(1), \ldots, f(n))$. 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 $\llbracket S \rrbracket^{\mathsf{shm}}_{V}$.

Reconstruction. Consider a shared value $\llbracket s \rrbracket^{\mathsf{shm}} = (f(1), \ldots, f(n))$. If $t \geq n/2$, then it can be shown that, like in the additive scheme from Section B.1, 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.

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

Error detection ($t < n/2$). Assume $t < n/2$, and suppose that a most t shares among $(s_1, \dots, s_n)$ are incorrect. If all shares $(s_1, \dots, s_n)$ 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 $A \cdot (s_1, \dots, s_n)^T$ 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.

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 [24], 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 [35].

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.[11] 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 $[\![x_1]\!]_U^t, \dots, [\![x_L]\!]_U^t, [\![y_1]\!]_V^t, \dots, [\![y_L]\!]_V^t$. 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 $\delta \in W$

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

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 $[\![\alpha]\!]_U^t = (f(1),\dots,f(n))$ and $[\![\beta]\!]_U^t = (g(1),\dots,g(n))$, 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 $i = 1,\dots,n$, so $[\![\phi(\alpha,\beta)]\!]_W^{2t} = (h(1),\dots,h(n))$.
– 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 [18].

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

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

Inputs: Shared values $[\![x_1]\!]_U, \dots, [\![x_L]\!]_U, [\![y_1]\!]_V, \dots, [\![y_L]\!]_V$.
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 $([\![w]\!]_W^t, [\![w]\!]_W^{2t}) \leftarrow \Pi_{\mathsf{DoubleSh}}$
2. Parties locally compute $[\![\phi(x_\ell,y_\ell)]\!]_W^{2t} \leftarrow \phi([\![x_\ell]\!]_U^t, [\![y_\ell]\!]_V^t)$, for $\ell = 1,\dots,L$;
3. Parties compute $[\![e]\!]_W = [\![w]\!]_W^{2t} + \sum_{\ell=1}^L [\![\phi(u_\ell, v_\ell)]\!]_W^{2t}$ 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 $[\![z+\delta]\!]_W^t = (e+\delta) - [\![w]\!]_W^t$.

[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 $\Pi_{\mathsf{DotProd}}^{\mathsf{shm}}$ is $\mathcal{C}_{\mathsf{DotProd}}^{\mathsf{shm}} = d \cdot \log(|\mathbb{F}|) \cdot 5.5 \cdot n$, using the optimization from [26].

B.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 $[\![s]\!]_{\mathbb{F}}^{\mathsf{rep}} = ((r_1,r_2),(r_2,r_3),(r_3,r_1))$. 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 B.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 $\mathit{\Pi}^{\mathsf{rep}}_{\mathsf{DotProd}}$

Inputs: Shared values $[\![x_1]\!]_U, \dots, [\![x_L]\!]_U, [\![y_1]\!]_V, \dots, [\![y_L]\!]_V$.
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 $[\![\phi(x_\ell, y_\ell)]\!]^{\mathsf{add}}_W \leftarrow \phi([\![x_\ell]\!]^{\mathsf{rep}}_U, [\![y_\ell]\!]^{\mathsf{rep}}_V)$, for $\ell = 1, \dots, L$;
2. Parties sample $[\![0]\!]^{\mathsf{add}}_W$ and then locally compute $[\![z]\!]^{\mathsf{add}}_W = [\![0]\!]^{\mathsf{add}}_W + \sum_{\ell=1}^{L} [\![\phi(x_\ell, y_\ell)]\!]^{\mathsf{add}}_W$.
3. Parties convert $[\![z+\delta]\!]^{\mathsf{rep}}_W \leftarrow [\![z]\!]^{\mathsf{add}}_W$.

C Communication Complexity of CHURP

CHURP, a dynamic PSS protocol proposed in [32], 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 $\mathit{\Pi}_{\mathsf{PartialPSS}}$ 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 [32] for details). The concrete communication complexity of this step is $2Ln^2$ 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 $\mathit{\Pi}_{\mathsf{PSS}}$ 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 $2Ln^2$.

We see then that the total (off-chain) communication complexity in Opt-CHURP is $9Ln^2 \log(|\mathbb{F}|)$ bits.

D Secure Computation over Elliptic Curves

So far we have presented a fairly comprehensive "toolbox" for performing secure computation over elliptic curves. We may view the LSS homomorphism $\phi : \mathbb{F}_p \to \mathbb{G}$ defined by $\phi(x) = x \cdot G$ as a function that encodes x into the exponent of G. While this enables the applications we presented in Section 3, Section 4 and Section 5, it does not an efficient way of *decoding.*

The following example illustrates why this might lead to issues in some applications: Parties hold $[\![m]\!]_{\mathbb{F}}$ and wish to encrypt it using 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. But what if the parties wish to recover $[\![m]\!]$ from the encryption? Clearly, a party cannot recover m_i from $m_i \cdot G$ since m_i (the share) is a random field element. On the other hand, we cannot reconstruct $m \cdot G$ towards a party as that would reveal the message.[12]

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.

D.1 Preliminaries

In the following, we assume $[\![\cdot]\!]$ corresponds to a secret-sharing scheme capable of detecting errors, such as Shamir secret-sharing (cf. B.2). Additionally, we will use two auxiliary functionalities which we describe here.

Functionality $\mathcal{F}_{\mathsf{sRand}}$. The functionality $\mathcal{F}_{\mathsf{sRand}}$ used in the secure injective encoding in Section D.2 has also seen other uses, in particular in connection with secure truncation protocols such as in [15]. $\mathcal{F}_{\mathsf{sRand}}$ 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 $i = 0, \dots, k - \ell - 1$.
2. Each party locally computes $[\![r]\!] = -2^{k-1} b_{k-\ell-1} + 2^{\ell} \sum_{i=0}^{k-\ell-1} 2^i b_i$.

[12] A recent work show how to compute these discrete logs on secret-shared inputs and their method can be seen as complimentary to ours [19]

Protocols $\mathit{\Pi}_{\mathsf{IsSqr}}$ and $\mathit{\Pi}_{\mathsf{Sqrt}}$. 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 $\mathit{\Pi}_{\mathsf{IsSqr}}$

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

Lemma 2. *Protocol $\mathit{\Pi}_{\mathsf{IsSqr}}$ securely 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 $\mathit{\Pi}_{\mathsf{Sqrt}}$ 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 $\mathit{\Pi}_{\mathsf{IsSqr}}$. 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 $\mathit{\Pi}_{\mathsf{Sqrt}}$

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 3. *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. $\square$

D.2 Secure Encoding and Decoding

In the following section we assume that $[\![\cdot]\!]$ corresponds to a secret-sharing scheme capable of detecting errors, such as Shamir secret-sharing. 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 [22]): 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 functionalities: The first protocol is $\mathcal{F}_{\mathsf{IsSqr}}$, which takes as input a secret-shared value $[\![x]\!]$ and outputs 1 if x is a square, and 0 otherwise. That is, if $\mathcal{F}_{\mathsf{IsSqr}}$ outputs 1, then there exists a value y such that $x^2 = y \mod p$. The other protocol is $\mathcal{F}_{\mathsf{Sqrt}}$ which, on input a square $[\![x]\!]$, outputs $[\![y]\!]$ satisfying $y = x^2 \bmod p$.

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 verify that the reshared values are consistent (cf. B.2).
3. For $j = 2, \dots, t+1$ 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 $\mathcal{C}_{\mathsf{Share}}(n) + \mathcal{C}_{\mathsf{Check}}(n) + (t+1)(\mathcal{C}_{\mathsf{Rand}}(1) + \mathcal{C}_{\mathsf{Mul}}(4) + \mathcal{C}_{\mathsf{Open}}(1))$.

Lemma 4. *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. $\square$

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 $\mathcal{F}_{\mathsf{IsSqr}}([\![x]\!] + r)$, 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. The final thing we require is a tuple $(\llbracket R \rrbracket_{\mathbb{G}}, \llbracket r_x \rrbracket, \llbracket r_y \rrbracket)$ where $R = (r_x, r_y)$. Such a tuple can be generated by sampling a random $\llbracket R \rrbracket_{\mathbb{G}}$ and then using step 2 in Π_{Decode} to obtain $\llbracket r_x \rrbracket$ and $\llbracket r_y \rrbracket$.

Protocol Π_{Encode}

Inputs: $\llbracket m \rrbracket$ the message to be encoded.
Outputs: $\llbracket \mathsf{En}(m) \rrbracket_{\mathbb{G}}$, $\llbracket r \rrbracket$.

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

Protocol Π_{Encode} computes the injective encoding of m with complexity

$$\mathcal{C}_{\mathsf{Encode}} = \mathcal{C}_{\mathsf{sRand}}(k) + \mathcal{C}_{\mathsf{IsSqr}}(k) + \mathcal{C}_{\mathsf{Sqrt}}(1) + 2\mathcal{C}_{\mathsf{Open}}(1) + \mathcal{C}_{\mathsf{Rand}}(1) + \mathcal{C}_{\mathsf{Mul}}(4).$$

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. In the cost formula, k denotes the number of repetitions of the first two steps. [22] proves that a suitable r is found in expected 3 iterations (i.e., k has expected value 3).