2019 IEEE Symposium on Security and Privacy

Threshold ECDSA from ECDSA Assumptions: The Multiparty Case

Jack Doerner
j@ckdoerner.net
Northeastern University

Yashvanth Kondi
ykondi@ccs.neu.edu
Northeastern University

Eysa Lee
eysa@ccs.neu.edu
Northeastern University

abhi shelat
abhi@neu.edu
Northeastern University

***Abstract*—Cryptocurrency applications have spurred a resurgence of interest in the computation of ECDSA signatures using threshold protocols—that is, protocols in which the signing key is secret-shared among n parties, of which any subset of size t must interact in order to compute a signature. Among the resulting works to date, that of Doerner et al. [1] requires the most natural assumptions while also achieving the best practical signing speed. It is, however, limited to the setting in which the threshold is two. We propose an extension of their scheme to *arbitrary* thresholds, and prove it secure against a malicious adversary corrupting up to one party less than the threshold under only the Computational Diffie-Hellman assumption in the Random Oracle model, an assumption strictly weaker than those under which ECDSA is proven.**

Whereas the best current schemes for threshold-two ECDSA signing use a Diffie-Hellman Key Exchange to calculate each signature's nonce, a direct adaptation of this technique to a larger threshold t would incur a round count linear in t; thus we abandon it in favor of a new mechanism that yields a protocol requiring $\log(t)+6$ rounds in total. We design a new consistency check, similar in spirit to that of Doerner et al., but suitable for an arbitrary number of participants, and we optimize the underlying two-party multiplication protocol on which our scheme is based, reducing its concrete communication and computation costs.

We implement our scheme and evaluate it among groups of up to 256 of co-located and 128 geographically-distributed parties, and among small groups of embedded devices. We find that in the LAN setting, our scheme outperforms all prior works by orders of magnitude, and that it is efficient enough for use even on smartphones or hardware tokens. In the WAN setting we find that, despite its logarithmic round count, our protocol outperforms the best constant-round protocols in realistic scenarios.

I. Introduction

Threshold Digital Signature Schemes [2] allow a group of individuals to delegate their joint authority to sign a message to any subcommittee among themselves that is larger than a certain, predetermined size. Specifically, a t-of-n threshold signature scheme is a set of protocols that allow n parties to jointly generate a single public key, along with n private shares of a joint secret key sk, and then securely sign messages if and only if t of those parties participate in the signing operation. In addition to the standard unforgeability properties required of all signature schemes, threshold schemes must satisfy the properties of privacy against $t-1$ malicious participants with respect to the secret key shares of honest parties, and correctness against $t-1$ malicious participants with respect to signature output. That is, no group of $t-1$ colluding parties should be able to recover the secret key, even by interacting with additional honest parties, nor should they be able to trick an honest party into signing a message unwillingly. Threshold signature schemes are thus best modeled as a special case of secure multiparty computation (MPC).

The Elliptic Curve Digital Signature Algorithm (ECDSA) is a standardized [3]–[5] derivative of the DSA scheme of David Kravitz [6], which improves upon the efficiency of its forebear by replacing arithmetic modulo a prime with operations over an elliptic curve. It is widely deployed in various web-security technologies such as DNSSec and TLS, in various authentication protocols, in binary signing, and in crypocurrencies, including Bitcoin [7] and Ethereum [8]. Although ECDSA is in widespread use, designing threshold signing protocols for ECDSA has been challenging due to the unusual structure of the signing algorithm: in each signature, a nonce k, its multiplicative inverse $1/k$, and the product sk/k (where sk is the secret key) all appear simultaneously. Computing these values efficiently in the multiparty context is the primary difficulty that threshold schemes must address.

MacKenzie and Reiter [9] constructed a 2-of-2 ECDSA protocol using multiplicative sharings of k and sk, which allowed shares of sk/k and $1/k$ to be computed via local operations, but their protocol required a mechanism to verify that the shares have been computed correctly. For this, they employed additively homomorphic encryption. Gennaro et al. [10] extended this technique, introducing a six-round protocol for general t-of-n signing, and Boneh et al. [11] subsequently optimized their extension in terms of computational efficiency, and reduced the round count to four. Meanwhile Lindell [12] introduced optimizations in the 2-of-2 setting, such that key-generation and signing required only 2.4 seconds and 37 milliseconds in practice, respectively. Unfortunately, these schemes require expensive zero-knowledge proofs, as well as the use of Paillier Encryption [13], which leads both to poor performance and to reliance upon assumptions such as the Decisional Composite Residuosity Assumption (and a new assumption about the Paillier cryptosystem, in the case of Lindell's protocol) that are foreign to the mathematics on which ECDSA is based.

Doerner et al. [1] propose an alternative solution for 2-of-n threshold key generation and signing: while their protocol retains the multiplicative sharings of prior approaches, they forgo operating on Paillier ciphertexts. Instead, they construct a

DOI 10.1109/SP.2019.00024

new, hardened variant of Gilboa's multiplication-by-oblivious-transfer technique [14], by which their protocol converts multiplicative shares into additive shares, and thereby produces additive shares of the final ECDSA signature. Security against malicious adversaries is achieved via a novel *consistency check* that leverages relationships among various elements of an ECDSA signature to ensure that the multipliers receive consistent inputs. Their scheme requires only two rounds and outperforms prior schemes by one to two orders of magnitude in terms of computational efficiency, such that signatures can be produced in under four milliseconds, and key generation for two parties can be completed in under 45 milliseconds. Moreover, their scheme was proven secure using only the Computational Diffie-Hellman Assumption [15], an assumption "native" to elliptic curves and implied by the Generic Group Model [16] (in which ECDSA is proven secure [17]), and the Random Oracle Model.

While the 2-of-n key-generation protocol of Doerner et al. can be generalized to arbitrary thresholds, their signing scheme is in a few respects inherently limited to two parties. As with prior two-party schemes, it uses a Diffie-Hellman Key Exchange [15] to calculate the signature's *instance key* $R = k \cdot G$ (where G is the elliptic curve group generator), given a multiplicative sharing of k. With a threshold larger than two, the long-standing open problem of multiparty key exchange is implicated. A direct extension of the Diffie-Hellman method to t parties would require $t - 1$ rounds, and, though key exchange can be achieved in a sublinear number of rounds via indistinguishability obfuscation [18] in the general case, or bilinear pairings [19] when $t = 3$, neither of these methods results in a practically-efficient protocol with ECDSA-native assumptions. Additionally, the consistency check that ensures security against malicious adversaries is a decidedly two-party construction: it relies upon the asymmetrical roles of the parties, and integrates proof-wise with the aforementioned Diffie-Hellman Exchange. Furthermore, we note that the scheme of Doerner et al. realizes a nonstandard, two-party specific functionality. Though they prove in the Generic Group Model that this functionality confers no additional power to an adversary, it does allow one party to bias the distribution of the instance key to a negligible degree, which gives that party an undesirable subliminal channel [20].

In this work, we describe an extension of the protocols of Doerner et al. to arbitrary thresholds. We formally define a new multiparty functionality, replace the key exchange component with an alternative based on multiparty multiplication, develop a new consistency check, and optimize the underlying primitives for the new setting and protocol structure. We implement our protocol and test it with a large number of parties, showing in particular that 256 parties can jointly sign in about half a second over LAN, and 128 parties require about four seconds to sign when spread around the world.

A. Our Techniques

Recall that an elliptic curve is defined by the tuple $(\mathbb{G}, G, q)$, where $\mathbb{G}$ is the group of order q of points on the curve, and G is the generator for that group. An ECDSA Signature on a message m under the secret key sk comprises a pair (sig, r_x) of integers in $\mathbb{Z}_q$ such that

$$\mathsf{sig} = \frac{H(m) + \mathsf{sk} \cdot r_x}{k}$$

where k is a uniform element from $\mathbb{Z}_q$ and r_x is the x-coordinate of the elliptic curve point $R = k \cdot G$. We frame our task as the construction of a multiparty computation at the end of which participating parties obtain additive shares of such a signature, having supplied secret shares of sk as input. We additionally require a protocol for generating shares of sk. As this protocol will also perform one-time initialization for many subsequent signatures, we refer to it as the *setup* protocol.

Our setup protocol is a natural extension of Doerner et al. [1], requiring only minor changes to ensure security against a dishonest majority of participants. When it completes successfully, each of the n participating parties receives a point on a $(t - 1)$-degree polynomial. The y-intercept of this polynomial is the secret key sk, as per Shamir's secret sharing scheme [21]. This allows any group of t parties to obtain an additive sharing of sk using the appropriate Lagrange coefficients. This additive sharing is the input to our signing protocol. Our signing protocol, however, diverges from that work and can be understood in terms of four logical phases.

1) *Instance Key Multiplication.* Once a group of parties $\mathbf{P}$ (where $|\mathbf{P}| = t$) have agreed to sign a message, each party $\mathcal{P}_i \in \mathbf{P}$ samples a multiplicative share k_i; these shares jointly define the common instance key k. Using a t-party multiplication protocol, the parties obtain both an additive sharing of k and a *multiplicatively-padded* additive sharing of $1/k$.
2) *Secret Key Multiplication.* As the parties now have additive sharings of sk and $1/k$, a GMW-style [22] multiplication protocol can be used to obtain an additive sharing of sk/k.
3) *Consistency Check.* The parties compute $R = k \cdot G$ and verify that consistent and correct inputs were used in the previous phases. Each party broadcasts a set of values that sum to predictable targets if and only if all parties have used inputs in the Secret Key Multiplication phase that are consistent with those used in the Instance Key Multiplication phase. This consistency check is similar in form and purpose to the consistency check employed by Doerner et al., but it operates in a broadcast fashion and enforces additional relationships required due to differences between the previous phases of our protocol and their analogues in Doerner et al.'s scheme.
4) *Signing.* Once the consistency of all inputs has been checked, each party i in the set of participants $\mathbf{P}$ is convinced that it holds v_i, w_i, and R such that for some value k,

$$\sum_{i \in \mathbf{P}} v_i = \frac{1}{k} \quad \text{and} \quad \sum_{i \in \mathbf{P}} w_i = \frac{\mathsf{sk}}{k} \quad \text{and} \quad R = k \cdot G$$

The parties locally compute their shares of the signature

$$\mathsf{sig}_i := v_i \cdot H(m) + w_i \cdot r_x$$

and broadcast them. The signature is then reconstructed

$$\mathsf{sig} := \sum_{i\in\mathbf{P}} \mathsf{sig}_i$$

and verified using the standard verification algorithm.

Our signing protocol is therefore essentially composed of three maliciously secure t-party multipliers, augmented by a check message to enforce consistency of inputs. These multipliers come in two flavors: one that converts a multiplicative sharing into an additive sharing, and a GMW-style multiplier, which produces an additive sharing of the product of two additive sharings. We realize both varieties of multiplier by evaluating a two-party multiplication protocol between each pair of parties. The asymptotic round count of the overall protocol is determined by the fact that data dependencies in the conversion process between multiplicative and additive shares require these two-party multiplication protocols to be evaluated $\log(t)$ sequential groups.

Our two-party multiplier is based upon Oblivious Transfer (OT) and derived from the two-party multiplication protocol of Doerner et al. [1], who were inspired in turn by the semi-honest multiplication protocol of Gilboa [14], but we improve upon the performance of their protocol in terms of both communication and computation. The protocol of Doerner et al. specifies that one of the two parties encodes its input using a high-entropy encoding scheme, and the length of this encoded input determines the number of OT instances required, which in turn strongly determines the performance of the multiplication protocol as a whole. On the other hand, our new protocol specifies that both parties choose random inputs, and later send correction messages to adjust their output values as necessary. Allowing for encodings only of random values rather than requiring the ability to encode specific inputs simplifies the encoding scheme considerably and reduces the number of OT instances by an amount proportional to the ECDSA security parameter, or about 40% in practice. This improvement comes at the cost of one additional round in the general case, but if the parties' inputs are guaranteed to be unknown to the adversary during the evaluation of the protocol (as they are in our case), then the round count need not increase, and in the context of our ECDSA signing scheme, our new multiplier actually reduces the overall round count relative to a naïve composition of the multiplication protocol of Doerner et al.

B. Contributions

1) We present a t-of-n threshold ECDSA signing protocol that requires $\log(t) + 6$ rounds and prove it secure against a malicious adversary who statically corrupts $t-1$ participants using only the Computational Diffie-Hellman Assumption. In addition we modify the setup protocol of Doerner et al. [1] and prove it secure in the same setting.
2) We improve upon the two-party multiplication protocol of Doerner et al., achieving a concrete performance gain of roughly 40%. In our protocol, a randomized Gilboa-style multiplier generates an unauthenticated multiplication triple, and the output shares are later adjusted at the cost of communicating a single field element for each party. Our protocol also supports batched multiplications, with a reduction in communication relative to simple repetition.
3) We describe a folkloric technique for the composition of two-party multipliers to form a t-party multiplier requiring $\log(t) + 2$ rounds, or $\log(t) + 1$ in some circumstances.
4) We provide an implementation of our protocol in the Rust language, and benchmark it on commodity server-class hardware in both the WAN and LAN settings, as well as on embedded devices. In the LAN setting, we evaluate our protocol with up to 256 parties. In the WAN setting, we evaluate with 128 parties spread across 16 datacenters. With respect to signing, our scheme outperforms all prior work in the LAN setting by a factor of 40 or more, and it is competitive in the WAN setting in spite of its round count. Though no prior works report the concrete setup performance of an arbitrary-threshold ECDSA scheme, we conjecture that ours improves dramatically upon them.

C. Organization

We establish the notation and building blocks for our protocols in Section II. We describe our improved protocol for two-party multiplication in Section III, which we use to construct t-party multiplication in Section IV. We specify our t-of-n threshold ECDSA protocol in Section V. We analyze the cost of this protocol in Section VI and provide details of our implementation and its performance in Section VII. Finally, in the full version of this paper, we prove our protocols secure.

II. Preliminaries and Definitions

A. Notation

Throughout this paper, we use $(\mathbb{G}, G, q)$ to represent the elliptic curve over which signatures are calculated, where $\mathbb{G}$ is the group of curve points, G the curve generator, and q the order of the curve. Curve points are represented in $|q| = \kappa$ bits, which is also the curve's security parameter, and we use s to represent the statistical security parameter. Curve points are denoted with capitalized variables and scalars with lowercase. Vectors are given in bold and indexed by subscripts; thus $\mathbf{x}_i$ is the i^{th} element of the vector $\mathbf{x}$, which is distinct from the scalar variable x. We use $=$ for equality, $:=$ for assignment, $\leftarrow$ for sampling from a distribution, and $\overset{c}{\equiv}$ for computational indistinguishability. We make use of a Random Oracle $H^x(y) : \{0,1\}^* \mapsto \mathbb{Z}_q^x$ with its output length varying according to the function's superscript; when the superscript is omitted it is assumed to be 1. We use $\mathcal{P}_i$ to denote the party with index i, and variables may often be subscripted with an index to indicate that they belong to a particular party. When arrays are owned by a party, the party index always comes before the array index. For convenience, when only two parties are present in a context, they are referred to as Alice and Bob.

In functionalities, we assume standard and implicit bookkeeping. In particular, we assume that along with the other messages we specify, session IDs and party IDs are transmitted so that the functionality knows to which instance a message belongs and who is participating in that instance. We assume that the functionality aborts if a party tries to reuse a session ID, send messages out of order, etc. We use `slab-serif` to denote message tokens, which communicate the function of a message to its recipients. For simplicity, we omit from a functionality's specifier all parameters that we do not actively use. For example, many of our functionalities are parameterized by a group $\mathbb{G}$ of order q, but we leave implicit the fact that in any given instantiation all functionalities use the same group.

B. Digital Signatures

Definition 1 (Digital Signature Scheme [23])**.**
A *Digital Signature Scheme* is a tuple of probabilistic polynomial time (PPT) algorithms, $(\mathsf{Gen}, \mathsf{Sign}, \mathsf{Verify})$ such that:

1) Given a security parameter κ, the Gen algorithm outputs a public key/secret key pair: $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{Gen}(1^\kappa)$
2) Given a secret key sk and a message m, the Sign algorithm outputs a signature σ: $\sigma \leftarrow \mathsf{Sign}_{\mathsf{sk}}(m)$
3) Given a message m, signature σ, and public key pk, the Verify algorithm outputs a bit b indicating whether the signature is valid or invalid: $b \coloneqq \mathsf{Verify}_{\mathsf{pk}}(m, \sigma)$

A Digital Signature Scheme satisfies two properties:

1) (Correctness) With overwhelmingly high probability, all valid signatures must verify. Formally, over $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{Gen}(1^\kappa)$ and all messages m in the message space,
$$\Pr_{\mathsf{pk},\mathsf{sk},m}\left[\mathsf{Verify}_{\mathsf{pk}}(m, \mathsf{Sign}_{\mathsf{sk}}(m)) = 1\right] > 1 - \operatorname{negl}(\kappa)$$
2) (Existential Unforgeability) No adversary can forge a signature for any message with greater than negligible probability, even if that adversary has seen signatures for polynomially many messages of its choice. Formally, for all PPT adversaries $\mathcal{A}$ with access to the signing oracle $\mathsf{Sign}_{\mathsf{sk}}(\cdot)$, where $\mathbf{Q}$ is the set of queries $\mathcal{A}$ asks the oracle,
$$\Pr_{\mathsf{pk},\mathsf{sk}}\left[\begin{array}{r}\mathsf{Verify}_{\mathsf{pk}}(m, \sigma) = 1 \wedge m \notin \mathbf{Q} : \\ (m, \sigma) \leftarrow \mathcal{A}^{\mathsf{Sign}_{\mathsf{sk}}(\cdot)}(\mathsf{pk})\end{array}\right] < \operatorname{negl}(\kappa)$$

C. ECDSA

ECDSA is parameterized by a group $\mathbb{G}$ of order q generated by a point G on an elliptic curve over the finite field $\mathbb{Z}_p$ of integers modulo a prime p. Assuming a curve has been fixed, the ECDSA algorithms are as follows [23]:

Algorithm 1. $\mathsf{Gen}(1^\kappa)$**:**

```
1) Uniformly choose a secret key $\mathsf{sk} \leftarrow \mathbb{Z}_q$.
2) Calculate the public key as $\mathsf{pk} \coloneqq \mathsf{sk} \cdot G$.
3) Output $(\mathsf{pk}, \mathsf{sk})$.
```

Algorithm 2. $\mathsf{Sign}(\mathsf{sk} \in \mathbb{Z}_q, m \in \{0,1\}^*)$**:**

```
1) Uniformly choose an instance key $k \leftarrow \mathbb{Z}_q$.
2) Calculate $(r_x, r_y) = R \coloneqq k \cdot G$.
3) Calculate
       $\mathsf{sig} \coloneqq \frac{H(m) + \mathsf{sk} \cdot r_x}{k}$
4) Output $\sigma \coloneqq (\mathsf{sig} \mod q, r_x \mod q)$.
```

Algorithm 3. $\mathsf{Verify}(\mathsf{pk} \in \mathbb{G}, m \in \{0,1\}^*, \sigma \in (\mathbb{Z}_q, \mathbb{Z}_q))$**:**

```
1) Parse $\sigma$ as $(\mathsf{sig}, r_x)$.
2) Calculate
       $(r'_x, r'_y) = R' \coloneqq \frac{H(m) \cdot G + r_x \cdot \mathsf{pk}}{\mathsf{sig}}$
3) Output 1 if and only if $(r'_x \mod q) = (r_x \mod q)$.
```

D. Security Model and Requisite Functionalities

We prove our protocols secure against any number of static corruptions in the Universal Composability (UC) framework, for an introduction to which we refer the reader to Canetti [24]. In this section we introduce a small set of functionalities that we use as building blocks. We begin with a commitment functionality and a committed-zero-knowledge functionality. Informally, the commitment functionality $\mathcal{F}^n_{\mathsf{Com}}$ allows a party to send a commitment to a message to a group of parties, and later reveal the same message to these parties. The functionality $\mathcal{F}^{R_{\mathsf{DL}};n}_{\mathsf{Com\text{-}ZK}}$ allows a party to send a commitment to both an elliptic curve point and a proof of knowledge of its discrete logarithm to a group of parties, and later reveal both. Concretely, $\mathcal{F}^n_{\mathsf{Com}}$ can be instantiated via the folkloric hash-based commitment construction, and $\mathcal{F}^{R_{\mathsf{DL}};n}_{\mathsf{Com\text{-}ZK}}$ via the Schnorr [25] protocol made non-interactive using the Fiat-Shamir [26] or Fischlin [27] transform, though only the latter achieves UC-security. For the sake of efficiency, our implementation uses the Fiat-Shamir transform in spite of this shortcoming.

Functionality 1. $\mathcal{F}^n_{\mathsf{Com}}$**:**

This functionality runs with a group of parties $\{\mathcal{P}_j\}_{j\in[1,n]}$, where one specific party $\mathcal{P}_i$ commits, and all other parties receive the commitment and committed value.

Commit: On receiving $(\texttt{commit}, \mathsf{id}^{\mathsf{com}}, x, \mathbf{I})$ from $\mathcal{P}_i$ where $\mathbf{I} \subseteq [1, n]$, if $(\texttt{commit}, \mathsf{id}^{\mathsf{com}}, \cdot, \cdot)$ does not exist in memory, then store $(\texttt{commit}, \mathsf{id}^{\mathsf{com}}, x, \mathbf{I})$ in memory and send $(\texttt{committed}, \mathsf{id}^{\mathsf{com}}, i)$ to all parties $\mathcal{P}_j$ for $j \in \mathbf{I}$.

Decommit: On receiving $(\texttt{decommit}, \mathsf{id}^{\mathsf{com}})$ from $\mathcal{P}_i$, send $(\texttt{decommitted}, \mathsf{id}^{\mathsf{com}}, x)$ to every party $\mathcal{P}_j$ for $j \in \mathbf{I}$

Functionality 2. $\mathcal{F}^{R_{\mathsf{DL}};n}_{\mathsf{Com\text{-}ZK}}$**:**

This functionality is parameterized by a group $\mathbb{G}$ of order q generated by G. It runs with a group of parties $\{\mathcal{P}_j\}_{j\in[1,n]}$, where one party $\mathcal{P}_i$ is the prover, and the others verify.

Commit Proof: On receiving $(\texttt{com-proof}, \mathsf{id}^{\mathsf{com\text{-}zk}}, x, X, \mathbf{I})$ from party $\mathcal{P}_i$ where $x \in \mathbb{Z}_q$ and $X \in \mathbb{G}$, if $(\texttt{com-proof}, \mathsf{id}^{\mathsf{com\text{-}zk}}, \cdot, \cdot, \cdot)$ does not exist in memory, then send $(\texttt{committed}, \mathsf{id}^{\mathsf{com\text{-}zk}}, i)$ to every party $\mathcal{P}_j$ for $j \in \mathbf{I}$ and store $(\texttt{com-proof}, \mathsf{id}^{\mathsf{com\text{-}zk}}, x, X, \mathbf{I})$ in memory.

Decommit Proof: On receiving $(\texttt{decom-proof}, \mathsf{id}^{\mathsf{com\text{-}zk}})$ from party $\mathcal{P}_i$, if $(\texttt{com-proof}, \mathsf{id}^{\mathsf{com\text{-}zk}}, x, X, \mathbf{I})$ exists in memory, then:

1) If $X = x \cdot G$, send $(\texttt{accept}, \mathsf{id}^{\mathsf{com\text{-}zk}}, X)$ to every party $\mathcal{P}_j$ for $j \in \mathbf{I}$.
2) Otherwise send $(\texttt{fail}, \mathsf{id}^{\mathsf{com\text{-}zk}})$ to every $\mathcal{P}_j$ for $j \in \mathbf{I}$.

In addition, our multiplication protocols make use of Correlated Oblivious Transfer extensions [28], which we model using the $\mathcal{F}^{\eta}_{\mathsf{COTe}}$ functionality of Doerner et al. [1], who derive it in turn from a similar functionality introduced by Keller et al. [29]. We reproduce their functionality here, for completeness. In short, $\mathcal{F}^{\eta}_{\mathsf{COTe}}$ interacts with two parties: A sender, who supplies a vector of correlations, and a receiver, who supplies vector of choice bits. For each vector element, the functionality returns to the sender a random pad, and to the receiver either the same random pad, or the same pad plus the sender's correlation. Concretely, we instantiate this functionality in the same manner as Doerner et al., using the OT-extension protocol of Keller et al. [29], with Doerner et al.'s VSOT (a derivative of Simplest OT [30]) performing the required base OTs.

Functionality 3. $\mathcal{F}^{\eta}_{\mathsf{COTe}}$**:**

This functionality is parameterized by a batch size η and a set of groups $\{\mathbb{G}_i\}_{i\in[1,\eta]}$, one group for each element in a batch (though groups are not necessarily unique). It runs with a sender S and a receiver R, who may participate in the Init phase once, and the Choice and Transfer phases many times.

Init: On receiving $(\texttt{init})$ from both parties, store $(\texttt{ready})$ in memory and send $(\texttt{init-complete})$ to the receiver.

Choice: On receiving $\big(\texttt{choose}, \mathsf{id}^{\mathsf{ext}}, \boldsymbol{\beta}\big)$ from the receiver, if $\big(\texttt{choice}, \mathsf{id}^{\mathsf{ext}}, \cdot\big)$ with the same $\mathsf{id}^{\mathsf{ext}}$ does not exist in memory, and if $(\texttt{ready})$ does exist in memory, and if $\boldsymbol{\beta} \in \{0,1\}^{\eta}$, then send $(\texttt{chosen})$ to the sender and store $\big(\texttt{choice}, \mathsf{id}^{\mathsf{ext}}, \boldsymbol{\beta}\big)$ in memory.

Transfer: On receiving $\big(\texttt{transfer}, \mathsf{id}^{\mathsf{ext}}, \boldsymbol{\alpha}\big)$ from the sender, if a message of the form $\big(\texttt{choice}, \mathsf{id}^{\mathsf{ext}}, \boldsymbol{\beta}\big)$ exists in memory with the same $\mathsf{id}^{\mathsf{ext}}$, and if $\big(\texttt{complete}, \mathsf{id}^{\mathsf{ext}}\big)$ does not exist in memory, and if $\boldsymbol{\alpha} \in \mathbb{G}_1 \times \ldots \times \mathbb{G}_\eta$, then:

1) Sample a vector of random pads $\boldsymbol{\omega}_{\mathsf{S}} \leftarrow \mathbb{G}_1 \times \ldots \times \mathbb{G}_\eta$
2) Send $(\texttt{pads}, \boldsymbol{\omega}_{\mathsf{S}})$ to the sender.
3) Compute $\boldsymbol{\omega}_{\mathsf{R}} := \big\{\boldsymbol{\beta}_i \cdot \boldsymbol{\alpha}_i - \boldsymbol{\omega}_{\mathsf{S},i}\big\}_{i\in[1,\eta]}$.
4) Send $(\texttt{padded-correlation}, \boldsymbol{\omega}_{\mathsf{R}})$ to the receiver.
5) Store $(\texttt{complete}, \mathsf{id}^{\mathsf{ext}})$ in memory.

III. Improved Two-party Multiplication

Doerner et al. [1] built their two party signing protocol atop two-party multiplication, and our protocol retains this property even when the number of signing parties is larger. While their protocol was optimized for the single-use computation setting (in which a small number of multiplications are computed by exactly two parties with no preprocessing), we design a new variant that is optimized for scenarios in which multiple overlapping pairs of parties compose their multiplications with one another. As a result we require a new two-party multiplication functionality. Specifically, our new functionality $\mathcal{F}^{\ell}_{\mathsf{2PMul}}$ involves three main phases. Following the one-time initialization phase, there is a preprocessing phase in which the parties must each send a message to the functionality in a specific order. Following this, they can supply their inputs (either party going first), and as each party's input is supplied, the opposite party's output is delivered. One party is also given the capability to define their own output by rushing in the last phase, which we will discuss in conjunction with the protocol that realizes this functionality. When $\mathcal{F}^{\ell}_{\mathsf{2PMul}}$ is composed, multiple instances can preprocess concurrently, and then inputs can be supplied as data dependencies require. This corresponds to a savings in rounds when instantiated with our multiplication protocol, relative to naïve composition of Doerner et al.'s multiplication protocol.

In addition, we add to both our protocol and our functionality the ability to batch multiplications together, and we make a simplification: whereas the functionality given by Doerner et al. allows an adversary to inject additive error into the output, we give the adversary no such capability. We note that this change is solely for simplicity of proof and ease of understanding: both functionalities output pairs of unauthenticated additive shares, and thus an adversary can always induce an offset.

Functionality 4. $\mathcal{F}^{\ell}_{\mathsf{2PMul}}$**:**

This functionality is parameterized by the group $\mathbb{Z}_q$ over which multiplication is to be performed, and the batch size ℓ. It runs with two parties, Alice and Bob, who may participate in the Init phase once, the remaining phases repeatedly.

Init: Wait for message $(\texttt{init})$ from Alice and Bob. Store $(\texttt{init-complete})$ in memory and send $(\texttt{init-complete})$ to Bob.

Bob-preprocess: On receiving $(\texttt{preprocess}, \mathsf{id}^{\mathsf{mul}})$ from Bob, if $(\texttt{bob-ready}, \mathsf{id}^{\mathsf{mul}})$ with the same $\mathsf{id}^{\mathsf{mul}}$ does not exist in memory, and if $(\texttt{init-complete})$ does exist in memory, then store $(\texttt{bob-ready}, \mathsf{id}^{\mathsf{mul}})$ in memory, and send $(\texttt{bob-ready}, \mathsf{id}^{\mathsf{mul}})$ to Alice.

Alice-preprocess: On receiving $(\texttt{preprocess}, \mathsf{id}^{\mathsf{mul}})$ from Alice, if there exists a message of the form $(\texttt{bob-ready}, \mathsf{id}^{\mathsf{mul}}, \cdot)$ in memory with the same $\mathsf{id}^{\mathsf{mul}}$, and if $(\texttt{alice-ready}, \mathsf{id}^{\mathsf{mul}})$ does not exist in memory, then store $(\texttt{alice-ready}, \mathsf{id}^{\mathsf{mul}})$ in memory, and send $(\texttt{alice-ready}, \mathsf{id}^{\mathsf{mul}})$ to Bob.

Alice-input: On receiving $(\texttt{input}, \mathsf{id}^{\mathsf{mul}}, \mathbf{a})$ from Alice, if a message of the form $(\texttt{alice-ready}, \mathsf{id}^{\mathsf{mul}})$ exists in memory with the same $\mathsf{id}^{\mathsf{mul}}$, and if $(\texttt{alice-complete}, \mathsf{id}^{\mathsf{mul}}, \cdot, \cdot)$ and $(\texttt{bob-complete}, \mathsf{id}^{\mathsf{mul}}, \cdot, \cdot)$ do not exist in memory, and if $\mathbf{a} \in \mathbb{Z}_q$ then:

1) Sample $\mathbf{z}_{\mathsf{B}} \leftarrow \mathbb{Z}_q^{\ell}$
2) Send $(\texttt{output}, \mathsf{id}^{\mathsf{mul}}, \mathbf{z}_{\mathsf{B}})$ to Bob.
3) Store $(\texttt{alice-complete}, \mathsf{id}^{\mathsf{mul}}, \mathbf{a}, \mathbf{z}_{\mathsf{B}})$ in memory.

Bob-input: On receiving $(\texttt{input}, \mathsf{id}^{\mathsf{mul}}, \mathbf{b})$ from Bob, if there exists a message $(\texttt{alice-ready}, \mathsf{id}^{\mathsf{mul}})$ in memory with the same $\mathsf{id}^{\mathsf{mul}}$, and if $(\texttt{bob-complete}, \mathsf{id}^{\mathsf{mul}}, \cdot, \cdot)$ does not exist in memory, and if $\mathbf{b} \in \mathbb{Z}_q$ then:

1) If $(\texttt{alice-complete}, \mathsf{id}^{\mathsf{mul}}, \mathbf{a}, \mathbf{z}_{\mathsf{B}})$ exists in memory, then compute
$$\mathbf{z}_{\mathsf{A}} := \big\{\mathbf{a}_i \cdot \mathbf{b}_i - \mathbf{z}_{\mathsf{B},i}\big\}_{i\in[1,\ell]}$$
and send $(\texttt{output}, \mathsf{id}^{\mathsf{mul}}, \mathbf{z}_{\mathsf{A}})$ to Alice.
2) Otherwise send $(\texttt{bob-complete}, \mathsf{id}^{\mathsf{mul}})$ to Alice.

3) Store $(\texttt{bob-complete}, \mathsf{id}^{\mathsf{mul}}, \mathbf{b})$ in memory.

Alice-input-rush: On receiving $(\texttt{rush}, \mathsf{id}^{\mathsf{mul}}, \mathbf{a}, \mathbf{z}_{\mathsf{A}})$ from Alice, if two messages exist in memory with the forms $(\texttt{alice-ready}, \mathsf{id}^{\mathsf{mul}})$ and $(\texttt{bob-complete}, \mathsf{id}^{\mathsf{mul}}, \mathbf{b})$ respectively, but $(\texttt{alice-complete}, \mathsf{id}^{\mathsf{mul}}, \cdot, \cdot)$ does not exist in memory, and if $\mathbf{a} \in \mathbb{Z}_q$ and $\mathbf{z}_{\mathsf{A}} \in \mathbb{Z}_q$ then:

1) Compute

$$\mathbf{z}_{\mathsf{B}} := \left\{\mathbf{a}_i \cdot \mathbf{b}_i - \mathbf{z}_{\mathsf{A},i}\right\}_{i\in[1,\ell]}$$

2) Send $(\texttt{output}, \mathsf{id}^{\mathsf{mul}}, \mathbf{z}_{\mathsf{B}})$ to Bob.
3) Store $(\texttt{alice-complete}, \mathsf{id}^{\mathsf{mul}}, \mathbf{a}, \mathbf{z}_{\mathsf{B}})$ in memory.

We now present a two-party multiplication protocol that realizes the above functionality, which is based upon Oblivious Transfer, and specifically OT-extensions. The multiplication protocol of Doerner et al. specifies that Bob's OT choice bits comprise a high-entropy encoding of his input, and that Alice's OT correlations are determined by her input, and that each party's share of the product can be calculated simply by summing the outputs of the OT. The round complexity of their protocol is determined by the round complexity of OT-extensions, which is two when instantiated with the protocol of Keller et al. [29]. We abandon this approach, and instead specify that the two parties use oblivious transfer to perform a *randomized* multiplication (which corresponds to the preprocessing phase in the $\mathcal{F}^{\ell}_{\mathsf{2PMul}}$ functionality), and adjust their output shares after the fact (which corresponds to the input phase). While the randomized multiplication in our multiplier requires two rounds on its own, and the adjustment step a third, it is possible for many multipliers with data dependencies to evaluate their randomized multiplications concurrently, reducing the round count overall.

Protocol 1. Two-party Multiplication $(\pi^{\ell}_{\mathsf{2PMul}})$:

This protocol is parameterized by a statistical security parameter s and the group $\mathbb{Z}_q$ over which multiplication is to be performed. Let $\kappa = |q|$ and for convenience let $\eta = \xi \cdot \ell$ where $\xi = \kappa + 2s$ is the number of random choice bits per element in a batch and ℓ is the multiplication batch size (a parameter). This protocol makes use of a public *gadget vector* $\mathbf{g} \leftarrow \mathbb{Z}_q^{\eta}$, and it invokes the Correlated Oblivious Transfer functionality $\mathcal{F}^{\eta}_{\mathsf{COTe}}$ and the random oracle H. For technical reasons, Bob supplies ℓ, but we assume that it is available as a common input to both parties. Alice supplies a vector input integers $\mathbf{a} \in \mathbb{Z}_q^{\ell}$ and Bob supplies another vector of input integers $\mathbf{b} \in \mathbb{Z}_q^{\ell}$. Alice and Bob receive as output vectors of integers $\mathbf{z}_{\mathsf{A}} \in \mathbb{Z}_q^{\ell}$ and $\mathbf{z}_{\mathsf{B}} \in \mathbb{Z}_q^{\ell}$, respectively, such that $\mathbf{z}_{\mathsf{A},i} + \mathbf{z}_{\mathsf{B},i} = \mathbf{a}_i \cdot \mathbf{b}_i$ for all indices $i \in [1, \ell]$.

Init:

1) Alice and Bob initialize their OT extensions by transmitting $(\texttt{init})$ to $\mathcal{F}^{\eta}_{\mathsf{COTe}}$.

Encoding:

2) Bob samples a set of random OT choice bits, and uses them to calculate his one-time pads $\tilde{\mathbf{b}}$

$$\boldsymbol{\beta} \leftarrow \{0,1\}^{\ell\cdot\xi}$$
$$\tilde{\mathbf{b}} := \left\{\left\langle \mathbf{g}, \{\boldsymbol{\beta}_j\}_{j\in[i\cdot\xi+1,(i+1)\cdot\xi]}\right\rangle\right\}_{i\in[1,\ell]}$$

3) Alice samples her one-time pads $\tilde{\mathbf{a}} \leftarrow \mathbb{Z}_q^{\ell}$ and a set of check values $\hat{\mathbf{a}} \leftarrow \mathbb{Z}_q^{\ell}$ and sets $\boldsymbol{\alpha} \in \mathbb{Z}_q^{\xi\cdot\ell}$ as

$$\boldsymbol{\alpha} := \{\tilde{\mathbf{a}}_1 \| \hat{\mathbf{a}}_1\}_{j\in[1,\xi]} \,\|\, \ldots \,\|\, \{\tilde{\mathbf{a}}_n \| \hat{\mathbf{a}}_n\}_{j\in[1,\xi]}$$

Multiplication:

4) Alice and Bob access the $\mathcal{F}^{\eta}_{\mathsf{COTe}}$ functionality, supplying $\eta = \xi \cdot \ell$ as the OT-extension batch size. Alice plays the sender, supplying $\boldsymbol{\alpha}$ as her input, and Bob, the receiver, supplies $\boldsymbol{\beta}$. They receive as outputs, respectively, the arrays $\boldsymbol{\omega}_{\mathsf{A}} \in \mathbb{Z}_q^{\eta}$ and $\boldsymbol{\omega}_{\mathsf{B}} \in \mathbb{Z}_q^{\eta}$, which they interpret as

$$\left\{\tilde{\mathbf{z}}_{\mathsf{A},j} \middle\| \hat{\mathbf{z}}_{\mathsf{A},j}\right\}_{j\in[1,\eta]} = \boldsymbol{\omega}_{\mathsf{A}}$$
$$\left\{\tilde{\mathbf{z}}_{\mathsf{B},j} \middle\| \hat{\mathbf{z}}_{\mathsf{B},j}\right\}_{j\in[1,\eta]} = \boldsymbol{\omega}_{\mathsf{B}}$$

That is, $\tilde{\mathbf{z}}_{\mathsf{A}}$ is a vector wherein each element contains the first half of the corresponding element in Alice's output from $\mathcal{F}^{\eta}_{\mathsf{COTe}}$, and $\hat{\mathbf{z}}_{\mathsf{A}}$ is a vector wherein each element contains the second half. $\tilde{\mathbf{z}}_{\mathsf{B}}$ and $\hat{\mathbf{z}}_{\mathsf{B}}$ play identical roles for Bob. The steps in the protocol up to this point correspond to the Bob-preprocess phase in $\mathcal{F}^{\ell}_{\mathsf{2PMul}}$.

5) Alice and Bob generate 2ℓ shared, random values by calling the random oracle. As input they use the shared components of the transcript of the protocol that implements $\mathcal{F}^{\eta}_{\mathsf{COTe}}$, in order the ensure that these values have a temporal dependency on the completion of the previous step. In our proofs, we abstract this step as a coin tossing protocol.

$$\tilde{\boldsymbol{\chi}} \leftarrow H^{\ell}(1\|\mathsf{transcript})$$
$$\hat{\boldsymbol{\chi}} \leftarrow H^{\ell}(2\|\mathsf{transcript})$$

6) Alice computes

$$\mathbf{r} := \left\{\sum_{i\in[1,\ell]} \tilde{\boldsymbol{\chi}}_i \cdot \tilde{\mathbf{z}}_{\mathsf{A},i\cdot\xi+j} + \hat{\boldsymbol{\chi}}_i \cdot \hat{\mathbf{z}}_{\mathsf{A},i\cdot\xi+j}\right\}_{j\in[1,\xi]}$$
$$\mathbf{u} := \{\tilde{\boldsymbol{\chi}}_i \cdot \tilde{\mathbf{a}}_i + \hat{\boldsymbol{\chi}}_i \cdot \hat{\mathbf{a}}_i\}_{i\in[1,\ell]}$$

and sends $\mathbf{r}$ and $\mathbf{u}$ to Bob.

7) Bob aborts if

$$\bigvee_{j\in[1,\xi]} \left(\begin{array}{r} \mathbf{r}_j + \displaystyle\sum_{i\in[1,\ell]} \tilde{\boldsymbol{\chi}}_i \cdot \tilde{\mathbf{z}}_{\mathsf{B},i\cdot\xi+j} + \hat{\boldsymbol{\chi}}_i \cdot \hat{\mathbf{z}}_{\mathsf{B},i\cdot\xi+j} \\ \neq \displaystyle\sum_{i\in[1,\ell]} \boldsymbol{\beta}_{i\cdot\xi+j} \cdot \mathbf{u}_i \end{array}\right)$$

Note that steps 5, 6, and 7 correspond to the Alice-preprocess phase in $\mathcal{F}^{\ell}_{\mathsf{2PMul}}$.

8) Alice computes

$$\gamma_{\mathsf{A}} := \{\mathbf{a}_i - \tilde{\mathbf{a}}_i\}_{i\in[1,\ell]}$$

and sends γ_{A} to Bob. Meanwhile, Bob computes

$$\gamma_{\mathsf{B}} := \left\{\mathbf{b}_i - \tilde{\mathbf{b}}_i\right\}_{i\in[1,\ell]}$$

and sends γ_{B} to Alice.

9) Alice and Bob compute their output shares

$$\mathbf{z}_{\mathsf{A}} := \left\{\mathbf{a}_i \cdot \gamma_{\mathsf{B},i} + \sum_{j\in[1,\xi]} \mathbf{g}_j \cdot \tilde{\mathbf{z}}_{\mathsf{A},i\cdot\xi+j}\right\}_{i\in[1,\ell]}$$

$$\mathbf{z}_{\mathsf{B}} := \left\{\tilde{\mathbf{b}}_i \cdot \gamma_{\mathsf{A},i} + \sum_{j\in[1,\xi]} \mathbf{g}_j \cdot \tilde{\mathbf{z}}_{\mathsf{B},i\cdot\xi+j}\right\}_{i\in[1,\ell]}$$

Note that this step and step 8 correspond to the Alice-input, Bob-input, and Alice-input-rush phase in $\mathcal{F}^{\ell}_{\mathsf{2PMul}}$.

Theorem III.1. *The protocol $\pi^{\ell}_{\mathsf{2PMul}}$ UC-realizes the functionality $\mathcal{F}^{\ell}_{\mathsf{2PMul}}$ for a κ-bit field $\mathbb{Z}_q$ with s bits of statistical security in the $\mathcal{F}^{\eta}_{\mathsf{COTe}}$-hybrid Random Oracle Model, in the presence of a malicious adversary statically corrupting either party.*

Rushing Adversaries. In both $\mathcal{F}^{\ell}_{\mathsf{2PMul}}$ and $\pi^{\ell}_{\mathsf{2PMul}}$ we specify that during the adjustment process, either Alice or Bob may adjust their value first. We do this to ensure that both adjustments can occur in a single round, without assuming simultaneous message transmission: in the real world, one party will likely transmit slightly before the other, but neither party will know the transmission order until after both messages are sent. Due to the asymmetry in the equations that the parties use to calculate their output shares in step 9, however, this pseudo-simultaneous transmission opens up an opportunity for a rushing adversary to deprive the simulator of information that it requires to produce the γ_{B} message, which necessitates the addition of the previously-mentioned Alice-input-rush phase in $\mathcal{F}^{\ell}_{\mathsf{2PMul}}$.

Consider a similar functionality that lacked the final phase (always using the Alice-input phase when Alice adjusts her input), and imagine the procedure of the simulator $\mathcal{S}^{\mathbf{P}^*}_{\mathsf{Mul}}$ that simulates against Alice and plays the role of the ideal adversary for $\mathcal{F}^{\ell}_{\mathsf{2PMul}}$. If Bob adjusts his output first, then $\mathcal{F}^{\ell}_{\mathsf{2PMul}}$ will communicate Alice's output $\mathbf{z}_{\mathsf{A}}$ to $\mathcal{S}^{\mathbf{P}^*}_{\mathsf{Mul}}$. $\mathcal{S}^{\mathbf{P}^*}_{\mathsf{Mul}}$ must then calculate an adjustment value γ_{B} that causes the output in her view to be equal to the value $\mathbf{z}_{\mathsf{A}}$ returned by $\mathcal{F}^{\ell}_{\mathsf{2PMul}}$. In other words, γ_{B} must satisfy

$$\gamma_{\mathsf{B}} = \left\{\frac{\mathbf{z}_{\mathsf{A},i} - \sum_{j\in[1,\xi]} \mathbf{g}_j \cdot \tilde{\mathbf{z}}_{\mathsf{A},i\cdot\xi+j}}{\mathbf{a}_i}\right\}_{i\in[1,\ell]}$$

At this point, $\mathbf{z}_{\mathsf{A}}$ and $\tilde{\mathbf{z}}_{\mathsf{A}}$ should be known to the simulator, but since Alice has not yet transmitted her adjustment message, the simulator should not know $\mathbf{a}$, and consequently the correct value of γ_{B} cannot be calculated. We remedy this by compelling Alice to determine her own output $\mathbf{z}_{\mathsf{A}}$ in the functionality if and only if she performs her adjustment second. Consequently $\mathcal{S}^{\mathbf{P}^*}_{\mathsf{Mul}}$ can choose γ_{B} uniformly, and Alice's adjustment message γ_{A} subsequently fixes her output $\mathbf{z}_{\mathsf{A}}$ and thereby allows her input to be extracted via the above equation. Note that when simulating against Bob, and equivalent problem does not occur, since the equation Bob uses to adjust his output value does not involve his input value. We formalize the intuition presented here in the full version of this paper.

Round Count. As we have mentioned, our multiplication protocol $\pi^{\ell}_{\mathsf{2PMul}}$ requires an additional round relative to the protocol of Doerner et al. This third round is necessitated by the proof of security and not the protocol per se: the adjustment messages γ_{A} and γ_{B} have no data dependency upon the random multiplication that precedes them. However, an adversary with some knowledge of their counterparty's input could potentially use that knowledge in combination with the adjustment messages to compromise the random multiplication in some way, were the adjustment messages sent before the random multiplication is complete. In the context of the multiparty ECDSA signing protocol that we present in Section V, the parties' multiplication inputs are information-theoretically hidden prior to the multiplications themselves, and consequently, we can optimize the multiplication process slightly by sending the adjustment messages simultaneously with the randomized multiplication, saving one round.

Cost Comparison to Prior Work. Our multiplication protocol incurs a cost of $\kappa + 2s$ OT invocations per batched input, or $\ell \cdot (\kappa + 2s)$ for a batch of size ℓ. On the other hand, the encoding scheme used by Doerner et al.'s multiplication protocol specifies codewords of size $2\kappa + 2s$, which implies a cost of $2\kappa + 2s$ OT instances per multiplication. In practice, it is reasonable to choose $\kappa = 256$ and $s = 80$, under which parameterization our protocol yields a savings of more than 38% in terms of OT instances.

IV. Multiparty Multiplication

In this section, we compose multiple instances of the two-party multiplication functionality $\mathcal{F}^{\ell}_{\mathsf{2PMul}}$ in order to form a t-party multiplication protocol. Although we are aware of no previous papers that describe it specifically, the general technique of composing two-party multipliers to form multiparty multipliers has been in the folklore of MPC since time out of mind. Nevertheless, we give a full account of the functionality and protocol, and in the full version of this paper we give a proof of security.

Specifically, the ability of a malicious party playing the role of Alice to define its own output by rushing while interacting with $\mathcal{F}^{\ell}_{\mathsf{2PMul}}$ implies that an adversary that can do the same in the t-party setting, so long as at least one corrupted party plays the role of Alice. For simplicity, our functionality will assume that this is always the case and unconditionally allow corrupted parties to define their own output. In addition, we make the assumption in our functionality that the adversary can withhold or delay the output to any honest party (by, for example, refusing to engage in a required instance of $\mathcal{F}^{\ell}_{\mathsf{2PMul}}$ with that party). This is a simplifying assumption, because our protocol does not grant an adversary quite so much granularity.

We discuss this issue further in the full version of this paper. Finally, we note the simulator, which we will describe in the full version of this paper, cannot extract inputs from corrupted parties individually, but must instead extract the product of all corrupted parties' inputs. Consequently, we specify that a single ideal adversary (the simulator $\mathcal{S}^{\mathbf{P}^*}_{\mathsf{Mul}}$) interacts with the functionality in the corrupted parties' stead.

In both our functionality and protocol that follows it, a group of n parties run the setup phase, and any subgroup of t parties may subsequently compute an additive sharing of a product.

Functionality 5. $\mathcal{F}^{\ell,t,n}_{\mathsf{Mul}}$**:**

This functionality is parameterized by group $\mathbb{Z}_q$ over which multiplication is to be performed, the party count n, the threshold size t, and the batch size ℓ. The Init phase runs once with a group of parties $\{\mathcal{P}_i\}_{i\in[1,n]}$, and the Multiplication and Output phases may be run many times between any (varying) subgroup of parties indexed by $\mathbf{P} \subseteq [1,n]$ such that $|\mathbf{P}| = t$. An ideal adversary, denoted $\mathcal{S}^{\mathbf{P}^*}_{\mathsf{Mul}}$, statically corrupts the parties indexed by the set $\mathbf{P}^* \subset [1,n]$ such that $|\mathbf{P}^*| < t$. Inputs from corrupt parties are provided directly to the functionality by $\mathcal{S}^{\mathbf{P}^*}_{\mathsf{Mul}}$ as a single, combined value.

Init: Wait for message $(\texttt{init})$ from $\{\mathcal{P}_i\}_{i\in[1,n]\setminus\mathbf{P}^*}$ and from $\mathcal{S}^{\mathbf{P}^*}_{\mathsf{Mul}}$. Store $(\texttt{init-complete})$ in memory and send $(\texttt{init-complete})$ to $\{\mathcal{P}_i\}_{i\in[1,n]\setminus\mathbf{P}^*}$ and to $\mathcal{S}^{\mathbf{P}^*}_{\mathsf{Mul}}$.

Multiplication: Receive $(\texttt{mult}, \mathsf{id}^{\mathsf{mul}}, \mathbf{P}, \mathbf{a}_i)$ from each party $\mathcal{P}_i$ for $i \in \mathbf{P} \setminus \mathbf{P}^*$, and receive $(\texttt{mult}, \mathsf{id}^{\mathsf{mul}}, \mathbf{P}, \mathbf{a}_{\mathbf{P}^*}, \mathbf{z}_{\mathbf{P}^*})$ from $\mathcal{S}^{\mathbf{P}^*}_{\mathsf{Mul}}$. If $(\texttt{init-complete})$ exists in memory but $(\texttt{output}, \mathsf{id}^{\mathsf{mul}}, \cdot)$ does not exist in memory, and if $\mathbf{a}_{\mathbf{P}^*} \in \mathbb{Z}_q^\ell$ and $\mathbf{a}_i \in \mathbb{Z}_q^\ell$ for all $i \in \mathbf{P} \setminus \mathbf{P}^*$, and all parties agree to the same set $\mathbf{P}$, then sample

$$\{\mathbf{z}_i\}_{i\in\mathbf{P}\setminus\mathbf{P}^*} \leftarrow \mathbb{Z}_q^{|\mathbf{P}\setminus\mathbf{P}^*|\times\ell}$$

uniformly subject to

$$\left\{\mathbf{z}_{\mathbf{P}^*,j} + \sum_{i\in\mathbf{P}\setminus\mathbf{P}^*} \mathbf{z}_{i,j}\right\}_{j\in[1,\ell]} = \left\{\mathbf{a}_{\mathbf{P}^*,j} \cdot \prod_{i\in\mathbf{P}\setminus\mathbf{P}^*} \mathbf{a}_{i,j}\right\}_{j\in[1,\ell]}$$

and store $(\texttt{output}, \mathsf{id}^{\mathsf{mul}}, \mathbf{z})$ in memory.

Output: On receiving $(\texttt{release}, \mathsf{id}^{\mathsf{mul}}, i)$ from $\mathcal{S}^{\mathbf{P}^*}_{\mathsf{Mul}}$, if $(\texttt{output}, \mathsf{id}^{\mathsf{mul}}, \mathbf{z})$ exists in memory but $(\texttt{complete}, \mathsf{id}^{\mathsf{mul}}, i)$ does not, and if $i \in \mathbf{P} \setminus \mathbf{P}^*$, then send $(\texttt{output}, \mathsf{id}^{\mathsf{mul}}, \mathbf{z}_i)$ to $\mathcal{P}_i$, and store $(\texttt{complete}, \mathsf{id}^{\mathsf{mul}}, i)$ in memory.

We now give a protocol for t-party multiplication that realizes $\mathcal{F}^{\ell,t,n}_{\mathsf{Mul}}$. In our protocol specification, we will abstract away the fact that $\mathcal{F}^{\ell}_{\mathsf{2PMul}}$ is asymmetric, with designated Alice and Bob roles for participating parties. We will instead use phrasing such as "Parties $\mathcal{P}_i$ and $\mathcal{P}_j$ access $\mathcal{F}^{\ell}_{\mathsf{2PMul}}$ with inputs a and b" as shorthand to indicate that $\mathcal{P}_i$ plays the role of Alice and $\mathcal{P}_j$ that of Bob, with inputs as specified. We will continue to use this style throughout the rest of the paper. We illustrate the pattern of interaction during an instance of the protocol for the specific case of 8 parties as a wiring diagram in Figure 1.

Protocol 2. t-Party Multiplication $\left(\pi^{\ell,t,n}_{\mathsf{Mul}}\right)$**:**

This protocol is parameterized by the statistical security parameter s and the group $\mathbb{Z}_q$ over which multiplication is to be performed, and by the party count n, the threshold size t, and the batch size ℓ. It invokes the Two-party Multiplication functionality $\mathcal{F}^{\ell}_{\mathsf{2PMul}}$. The Init phase is run once with the entire group of parties $\{\mathcal{P}_i\}_{i\in[1,n]}$, and the Multiplication phase can be run repeatedly, each time with a unique $\mathsf{id}^{\mathsf{mul}}$ and a varying subset of parties parties $\mathbf{P} \subseteq [1,n]$ such that $|\mathbf{P}| = t$. During each multiplication, every party $\mathcal{P}_i$ for $i \in \mathbf{P}$ supplies an input vector $\mathbf{a}_i \in \mathbb{Z}_q^\ell$ and the unique index $\mathsf{id}^{\mathsf{mul}}$ and receives an output $\mathbf{z}_i \in \mathbb{Z}_q^\ell$, such that the outputs for all parties in $\mathbf{P}$ form an additive sharing of the element-wise product of the inputs.

Init: Each pair of parties $\mathcal{P}_i, \mathcal{P}_j$ for $i, j \in [1,n]$ such that $i < j$ initialize their multiplication oracle by sending $(\texttt{init})$ to their shared $\mathcal{F}^{\ell}_{\mathsf{2PMul}}$ instance.

Multiplication:

1) Each party $\mathcal{P}_i$ has input $\mathbf{a}_i$, and sets $\boldsymbol{\zeta}_i^0 := \mathbf{a}_i$.
2) For each pair of parties $\mathcal{P}_i, \mathcal{P}_j$ such that $i < j$:
   a) $\mathcal{P}_j$, acting as Bob, sends $(\texttt{preprocess}, \mathsf{id}^{\mathsf{mul}}_{i,j})$ to $\mathcal{F}^{\ell}_{\mathsf{2PMul}}$, where $\mathsf{id}^{\mathsf{mul}}_{i,j}$ is a unique, agreed upon index.
   b) On receiving $(\texttt{bob-ready}, \mathsf{id}^{\mathsf{mul}}_{i,j})$ from $\mathcal{F}^{\ell}_{\mathsf{2PMul}}$, $\mathcal{P}_i$, as Alice, sends $(\texttt{preprocess}, \mathsf{id}^{\mathsf{mul}}_{i,j})$ to $\mathcal{F}^{\ell}_{\mathsf{2PMul}}$.
   c) $\mathcal{P}_j$ receives $(\texttt{alice-ready}, \mathsf{id}^{\mathsf{mul}}_{i,j})$ from $\mathcal{F}^{\ell}_{\mathsf{2PMul}}$.
3) For $\rho \in [1, \log_2(t)]$:
   a) For each pair of parties $\mathcal{P}_i, \mathcal{P}_j$ in each contiguous non-overlapping subgroup of 2^ρ parties from $\mathbf{P}$, if $\mathcal{P}_i$ and $\mathcal{P}_j$ have not previously interacted during the course of this invocation of $\pi^{\ell,t,n}_{\mathsf{Mul}}$, then they send $(\texttt{input}, \mathsf{id}^{\mathsf{mul}}_{i,j}, \boldsymbol{\zeta}_i^{\rho-1})$ and $(\texttt{input}, \mathsf{id}^{\mathsf{mul}}_{i,j}, \boldsymbol{\zeta}_j^{\rho-1})$ to $\mathcal{F}^{\ell}_{\mathsf{2PMul}}$, respectively, and receive $(\texttt{output}, \mathsf{id}^{\mathsf{mul}}_{i,j}, \boldsymbol{\zeta}_i^{\rho,j})$ and $(\texttt{output}, \mathsf{id}^{\mathsf{mul}}_{i,j}, \boldsymbol{\zeta}_j^{\rho,i})$. If the party playing the role of Alice goes second, then it samples a random output and uses the rushing phase of $\mathcal{F}^{\ell}_{\mathsf{2PMul}}$.
   b) Each party $\mathcal{P}_i$ privately computes $\boldsymbol{\zeta}_i^\rho$ to be the element-wise sum of its output shares for round ρ:

$$\boldsymbol{\zeta}_i^\rho := \left\{\sum_{j\in\mathbf{P}^{\rho,i}} \boldsymbol{\zeta}_{i,l}^{\rho,j}\right\}_{l\in[1,\ell]}$$

   where $\mathbf{P}^{\rho,i} \subset \mathbf{P}$ such that $|\mathbf{P}^{\rho,i}| = 2^\rho - 1$ is the subgroup with whom $\mathcal{P}_i$ interacted in round ρ.
4) Each party $\mathcal{P}_i$ takes $\mathbf{z}_i := \boldsymbol{\zeta}_i^{\log_2(t)}$ to be their output.

Theorem IV.1. *The protocol $\pi^{\ell,t,n}_{\mathsf{Mul}}$ UC-realizes the functionality $\mathcal{F}^{\ell,t,n}_{\mathsf{Mul}}$ for a κ-bit field $\mathbb{Z}_q$ in the $\mathcal{F}^{\ell}_{\mathsf{2PMul}}$-hybrid Random Oracle Model, in the presence of a malicious adversary statically corrupting up to $t-1$ parties.*

Round Count. The protocol $\pi^{\ell,t,n}_{\mathsf{Mul}}$ requires each party to engage in t instances of the $\mathcal{F}^{\ell}_{\mathsf{2PMul}}$ functionality. The preprocessing phases of these instances are evaluated in parallel, but due

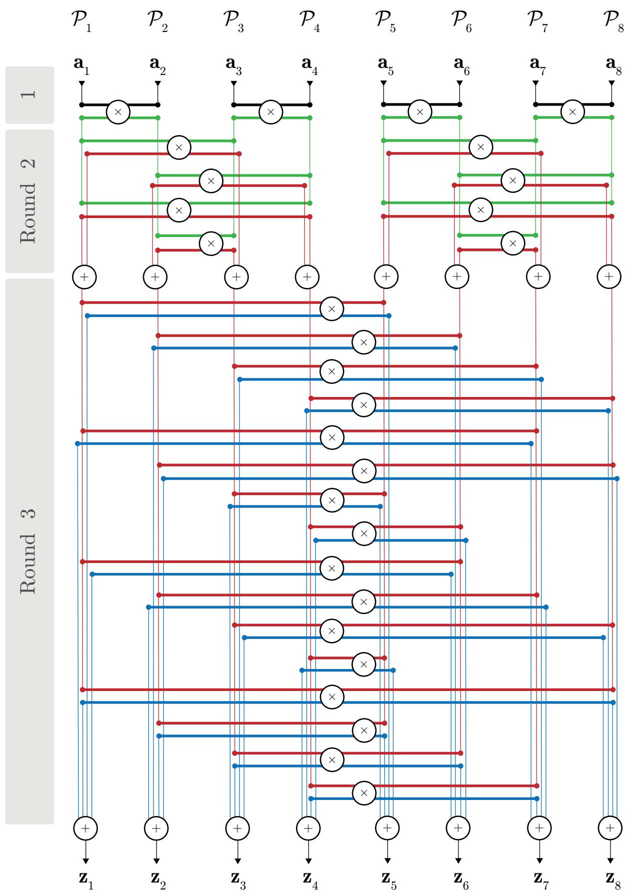
$\mathcal{P}_1$ $\mathcal{P}_2$ $\mathcal{P}_3$ $\mathcal{P}_4$ $\mathcal{P}_5$ $\mathcal{P}_6$ $\mathcal{P}_7$ $\mathcal{P}_8$
$\mathbf{a}_1$ $\mathbf{a}_2$ $\mathbf{a}_3$ $\mathbf{a}_4$ $\mathbf{a}_5$ $\mathbf{a}_6$ $\mathbf{a}_7$ $\mathbf{a}_8$
1
Round 2
Round 3
×
+
$\mathbf{z}_1$ $\mathbf{z}_2$ $\mathbf{z}_3$ $\mathbf{z}_4$ $\mathbf{z}_5$ $\mathbf{z}_6$ $\mathbf{z}_7$ $\mathbf{z}_8$

Fig. 1: **Illustration of a t-Party Multiplication among 8 parties**. We use × to denote an instance of the two-party multiplication functionality $\mathcal{F}^{\ell}_{\mathsf{2PMul}}$, and + to denote a local sum. Note that in each round all individual instances of $\mathcal{F}^{\ell}_{\mathsf{2PMul}}$ are invoked in parallel. Outputs from the first round are shown as green wires, from the second round as red wires, and from the third round as blue wires.

to data dependencies, the input-adjustment phases must be evaluated in $\log(t)$ sequential groups. Thus, when $\mathcal{F}^{\ell}_{\mathsf{2PMul}}$ is realized by $\pi^{\ell}_{\mathsf{2PMul}}$, $\pi^{\ell,t,n}_{\mathsf{Mul}}$ will require $\log(t)+2$ rounds in the general case. However, in our use-case we can apply the optimization discussed at the end of Section III to $\pi^{\ell}_{\mathsf{2PMul}}$, in light of which the round count of $\pi^{\ell,t,n}_{\mathsf{Mul}}$ is reduced to $\log(t)+1$.

V. Threshold ECDSA

In this section, we describe the threshold ECDSA functionality that our protocol realizes, followed by the protocol itself, which is broken into two parts: a setup protocol and a signing protocol. The former protocol is run once among a group of n parties, and its output may be reused many times by various subgroups of t parties engaging in the latter.

A. *The t-of-n ECDSA Functionality*

We have made no attempt to formulate a general signature functionality, but instead have modeled ECDSA in the threshold setting directly, much as previous works [11], [12] have done. Unlike the functionality of Doerner et al. [1], ours does not allow malicious parties the ability to bias the instance key. As is typical of dishonest-majority protocols, the adversary will have the ability to deprive honest parties of their output. We model this in our functionality by allowing any party to specify a vector $\mathbf{B} \subseteq \mathbf{P}$ of parties to block. We note that ECDSA makes use of a hash function, and that the standard specifies this function to be SHA-256. As we will discuss in Section VII, we use SHA-256 to instantiate the Random Oracle in our implementation. However, when our functionality makes use of the function H, it refers not to the Random Oracle but to SHA-256 specifically.

Functionality 6. $\mathcal{F}^{t,n}_{\mathsf{ECDSA}}$:

This functionality is parameterized by the Elliptic curve $(\mathbb{G}, G, q)$, as well as a hash function H. The setup phase runs once with a group of parties $\{\mathcal{P}_i\}_{i\in[1,n]}$, and the signing phase may be run many times between any (varying) subgroup of parties indexed by $\mathbf{P} \subseteq [1,n]$ such that $|\mathbf{P}| = t$.

Setup: On receiving $(\texttt{init})$ from all parties:

1) Sample and store the joint secret key, $\mathsf{sk} \leftarrow \mathbb{Z}_q$.
2) Compute and store the joint public key, $\mathsf{pk} := \mathsf{sk} \cdot G$.
3) Send $(\texttt{public-key}, \mathsf{pk})$ to all parties.
4) Store $(\texttt{ready})$ in memory.

Signing: On receiving $(\texttt{sign}, \mathsf{id}^{\mathsf{sig}}, \mathbf{P}, m)$ from each party $\mathcal{P}_i$ for $i \in \mathbf{P}$, where $\mathbf{P} \subseteq [1,n]$ such that $|\mathbf{P}| = t$ is the list of parties participating in this signature, if $(\texttt{ready})$ exists in memory but $(\texttt{complete}, \mathsf{id}^{\mathsf{sig}})$ does not exist in memory:

1) Sample $k \leftarrow \mathbb{Z}_q$ and store it as the instance key.
2) Wait for $(\texttt{get-instance-key}, \mathsf{id}^{\mathsf{sig}})$ from all parties $\mathbf{P}$.
3) Compute $(r_x, r_y) = R := k \cdot G$ and send $(\texttt{instance-key}, \mathsf{id}^{\mathsf{sig}}, R)$ to all parties.
4) Wait for $(\texttt{proceed}, \mathsf{id}^{\mathsf{sig}}, \mathbf{B}_i)$ from every party $\mathcal{P}_i$ for $i \in \mathbf{P}$. If some party sends $(\texttt{abort}, \mathsf{id}^{\mathsf{sig}})$, then halt.
5) Compute
$$\mathsf{sig} := \frac{H(m) + \mathsf{sk} \cdot r_x}{k}$$
6) Collect the signature, $\sigma := (\mathsf{sig} \mod q, r_x \mod q)$.
7) Compute
$$\mathbf{B} := \bigcup_{i\in\mathbf{P}} \mathbf{B}_i$$
8) Send $(\texttt{signature}, \mathsf{id}^{\mathsf{sig}}, \sigma)$ to each $\mathcal{P}_i$ for $i \in \mathbf{P} \setminus \mathbf{B}$
9) Store $(\texttt{complete}, \mathsf{id}^{\mathsf{sig}})$ in memory.

B. *Threshold Setup*

Our setup protocol is derived from the 2-of-n setup protocol of Doerner et al. [1]. Like their scheme, it uses simple techniques to produce and verify an n-party Shamir secret sharing [21] of a joint secret key sk, from which any t parties can derive a t-party additive sharing of sk with no further interaction; unlike their scheme, we use a proof of knowledge to ensure security against a dishonest majority. Their protocol sampled the public/private key pair as an n-party additive sharing, and then converted it to a Shamir sharing; we make a small improvement by sampling the Shamir sharing directly. Specifically, each party locally samples a random polynomial of degree degree-$(t-1)$ and distributes points at predetermined locations on this polynomial to the other parties. The parties sum the points they receive to construct a Shamir sharing of a single degree-$(t-1)$ polynomial. The parties then multiply their points on the shared polynomial by the elliptic curve generator G, broadcast the result, and verify that all subsets of their shares represent the same polynomial by homomorphically evaluating the polynomial in the curve group. For degree-2 polynomials, Doerner et al. required a number of evaluations quadratic in n, whereas we require only a linear number regardless of the polynomial degree. Since the homomorphic evaluation of the polynomial is equal to pk, an adversary can learn nothing more from the protocol than could be learned from *any* protocol that realizes the same functionality.

Protocol 3. Setup $(\pi^{t,n}_{\mathsf{ECDSA\text{-}Setup}})$**:**

This protocol is parameterized by the Elliptic curve $(\mathbb{G}, G, q)$, and invokes the $\mathcal{F}^{\ell}_{\mathsf{2PMul}}$, $\mathcal{F}^{\ell,t,n}_{\mathsf{Mul}}$, and $\mathcal{F}^{R_{\mathsf{DL}},n}_{\mathsf{Com\text{-}ZK}}$ functionalities. It runs among a group of parties $\{\mathcal{P}_i\}_{i\in[1,n]}$, taking no input, and yielding to each party $\mathcal{P}_i$ a point $p(i)$ on the polynomial p, and the joint public key pk.

Public Key Generation:

1) Each party $\mathcal{P}_i$ samples a random degree polynomial p_i of degree $t-1$.
2) For all pairs of parties $\mathcal{P}_i$ and $\mathcal{P}_j$, $\mathcal{P}_i$ sends $p_i(j)$ to $\mathcal{P}_j$ and receives $p_j(i)$ in return.
3) Each party $\mathcal{P}_i$ computes its point
$$p(i) := \sum_{j\in[1,n]} p_j(i)$$
4) Each party $\mathcal{P}_i$ computes $T_i := p(i)\cdot G$ and sends $(\texttt{com-proof}, \mathsf{id}^{\mathsf{com\text{-}zk}}_i, p(i), T_i)$ to $\mathcal{F}^{R_{\mathsf{DL}},n}_{\mathsf{Com\text{-}ZK}}$, using a fresh, unique value for $\mathsf{id}^{\mathsf{com\text{-}zk}}_i$.
5) Upon being notified of all other parties' commitments, each party $\mathcal{P}_i$ releases its proof by sending $(\texttt{decom-proof}, \mathsf{id}^{\mathsf{com\text{-}zk}}_i)$ to $\mathcal{F}^{R_{\mathsf{DL}},n}_{\mathsf{Com\text{-}ZK}}$.
6) Each party $\mathcal{P}_i$ receives $(\texttt{accept}, \mathsf{id}^{\mathsf{com\text{-}zk}}_j, T_j)$ from $\mathcal{F}^{R_{\mathsf{DL}},n}_{\mathsf{Com\text{-}ZK}}$ for each $j\in[1,n]\setminus\{i\}$ if $\mathcal{P}_j$'s proof of knowledge is valid. $\mathcal{P}_i$ aborts if it receives $(\texttt{fail}, \mathsf{id}^{\mathsf{com\text{-}zk}}_j)$ instead for any proof, or if there exists an index $x\in[1,n-t-1]$ such that $\mathbf{J}^x = [x, x+t]$ and $\mathbf{J}^{x+1} = [x+1, x+t+1]$ and
$$\sum_{j\in\mathbf{J}^x}\lambda^{\mathbf{J}^x}_j\cdot T_j \neq \sum_{j\in\mathbf{J}^{x+1}}\lambda^{\mathbf{J}^{x+1}}_j\cdot T_j$$
where $\lambda^{\mathbf{J}^x}_j$ and $\lambda^{\mathbf{J}^{x+1}}_j$ are party $\mathcal{P}_j$'s Lagrange coefficients for Shamir reconstruction with the sets of parties indexed by $\mathbf{J}^x$ and $\mathbf{J}^{x+1}$ respectively.
7) The parties compute the shared public key using any subset $\mathbf{J}\subseteq[1,n]$ such that $|\mathbf{J}| = t$
$$\mathsf{pk} := \sum_{j\in\mathbf{J}}\lambda^{\mathbf{J}}_j\cdot T_j$$

Auxiliary Setup:

8) Every party sends the $(\texttt{init})$ message to the $\mathcal{F}^{\ell,t,n}_{\mathsf{Mul}}$ functionality.
9) Every pair of parties $\mathcal{P}_i$ and $\mathcal{P}_j$ such that $i<j$ sends the $(\texttt{init})$ message to the $\mathcal{F}^{\ell}_{\mathsf{2PMul}}$ functionality.

Round Count. The Public Key Generation portion of $\pi^{t,n}_{\mathsf{ECDSA\text{-}Setup}}$ requires three broadcast rounds in total, but the initialization procedures in the Auxiliary Setup phase require five, when $\mathcal{F}^{n}_{\mathsf{COTe}}$ instantiated with Keller et al.'s OT-extension [29] and the VSOT protocol [1], as we intend. Since Auxiliary Setup is independent of Key Generation, these phases can be run concurrently, and the round count can be as low as five, concretely. Our implementation, however, runs them in sequence, yielding eight concrete rounds.

C. Threshold Signing

Finally, we give our protocol for arbitrary-threshold ECDSA signing. It follows the same general plan as that of Doerner et al. [1], being broken down into four distinct stages. Note, however, that unlike their protocol, the roles of all parties are symmetric, and all parties receive the final signature at the end (subject to the adversary's approval). The parties begin by sampling multiplicative shares of the instance key k, from which they can locally compute multiplicative sharings of $1/k$ and sk_i/k for $i\in\mathbf{P}$, where $\mathbf{P}$ is the set of signing parties and sk_i is the i^{th} party's additive share of sk. The multipliers discussed in prior sections are then used to convert these multiplicative sharings into additive sharings, and a consistency check ensures that they are all consistent with one another. Finally, each party creates a linear share of the signature using the information known to it, and the parties exchange shares.

Protocol 4. Signing $(\pi^{t,n}_{\mathsf{ECDSA\text{-}Sign}})$**:**

This protocol is parameterized by the Elliptic curve $(\mathbb{G}, G, q)$ and the statistical security parameter s, and invokes the $\mathcal{F}^{\ell}_{\mathsf{2PMul}}$, $\mathcal{F}^{\ell,t,n}_{\mathsf{Mul}}$, and $\mathcal{F}^{R_{\mathsf{DL}},n}_{\mathsf{Com\text{-}ZK}}$ functionalities. It runs among a group of parties $\mathbf{P}\subseteq[1,n]$ such that $|\mathbf{P}| = t$, taking as input the public key pk, the message m, and the signature index $\mathsf{id}^{\mathsf{sig}}$ (which is used to generate other unique indices as required) from each party $\mathcal{P}_i$, along with a point $p(i)$ on the polynomial that encodes the secret key, and yielding to each party a copy of the signature σ.

Instance Key Multiplication:

1) Each party $\mathcal{P}_i$ for $i\in\mathbf{P}$ samples their multiplicative share of the instance key $k_i\leftarrow\mathbb{Z}_q$ and a uniform pad value $\phi_i\leftarrow\mathbb{Z}_q$, and commits to the pad by sending $(\texttt{commit}, \mathsf{id}^{\mathsf{com}}_{i,1}, \phi_i, \mathbf{P})$ to $\mathcal{F}^{n}_{\mathsf{Com}}$, using a fresh value for $\mathsf{id}^{\mathsf{com}}_{i,1}$. All other parties are notified of $\mathcal{P}_i$'s commitment.
2) Each party $\mathcal{P}_i$ invokes $\mathcal{F}^{\ell,t,n}_{\mathsf{Mul}}$ with $\ell = 2$, supplying $\{k_i, \phi_i/k_i\}$ as its input along with a fresh, agreed-upon multiplication index, and receiving as output $\{u_i, v_i\}$. We elide the specific messages exchanged with the

functionality in this process, but note that

$$\sum_{i\in\mathbf{P}} u_i = \prod_{i\in\mathbf{P}} k_i = k \quad\text{and}\quad \sum_{i\in\mathbf{P}} v_i = \prod_{i\in\mathbf{P}} \frac{\phi_i}{k_i} = \frac{\phi}{k}$$

Secret Key Multiplication:

3) Each party $\mathcal{P}_i$ computes $\lambda_i^{\mathbf{P}}$, its Lagrange coefficient given that it is reconstructing sk with the parties in $\mathbf{P}$. $\mathcal{P}_i$ then computes sk_i, its additive share of the secret key for this group of parties
$$\mathsf{sk}_i := \lambda_i^{\mathbf{P}} \cdot p(i)$$
4) Each pair of parties, $\mathcal{P}_i$ and $\mathcal{P}_j$ invoke $\mathcal{F}^{\ell}_{\mathsf{2PMul}}$ with $\ell = 2$. The party with the lower index plays the role of Alice and the other Bob, and they use a fresh, agreed-upon multiplication index. The parties run the multiplication preprocessing and input phases, with $\mathcal{P}_i$ supplying as input $\{\mathsf{sk}_i, v_i\}$ and $\mathcal{P}_j$ supplying $\{v_j, \mathsf{sk}_j\}$. As outputs they receive $\{w_i^{j,1}, w_i^{j,2}\}$ and $\{w_j^{i,1}, w_j^{i,2}\}$, respectively. We again elide the specific messages involved in this process, but note that
$$w_i^{j,1} + w_j^{i,1} = \mathsf{sk}_i \cdot v_j \quad\text{and}\quad w_i^{j,2} + w_j^{i,2} = \mathsf{sk}_j \cdot v_i$$
5) Each party $\mathcal{P}_i$ sets
$$w_i := \mathsf{sk}_i \cdot v_i + \sum_{j\in\mathbf{P}\setminus\{i\}} \left(w_i^{j,1} + w_i^{j,2} \right)$$

Consistency Check:

6) Each party $\mathcal{P}_i$ computes $R_i := u_i \cdot G$ and commits to a proof of knowledge of discrete logarithm for this value by sending $(\texttt{com-proof}, \mathsf{id}_i^{\mathsf{com\text{-}zk}}, u_i, R_i, \mathbf{P})$ to $\mathcal{F}^{R_{\mathsf{DL}},n}_{\mathsf{Com\text{-}ZK}}$, using a fresh value for $\mathsf{id}_i^{\mathsf{com\text{-}zk}}$.
7) Upon being notified of all other parties' commitments, each party $\mathcal{P}_i$ releases the previous proof by sending $(\texttt{decom-proof}, \mathsf{id}_i^{\mathsf{com\text{-}zk}})$ to $\mathcal{F}^{R_{\mathsf{DL}},n}_{\mathsf{Com\text{-}ZK}}$
8) Each party $\mathcal{P}_i$ receives $(\texttt{accept}, \mathsf{id}_j^{\mathsf{com\text{-}zk}}, R_j)$ from $\mathcal{F}^{R_{\mathsf{DL}},n}_{\mathsf{Com\text{-}ZK}}$ for each $j \in \mathbf{P}\setminus\{i\}$ if $\mathcal{P}_j$'s proof of knowledge is valid. Once these messages are received, $\mathcal{P}_i$ computes
$$R := \sum_{j\in\mathbf{P}} R_j$$
If $\mathcal{P}_i$ instead receives $(\texttt{fail}, \mathsf{id}_j^{\mathsf{com\text{-}zk}})$ for any proof, then it aborts.
9) Each party $\mathcal{P}_i$ calculates
$$\begin{aligned} \Gamma_i^1 &:= v_i \cdot R \\ \Gamma_i^2 &:= v_i \cdot \mathsf{pk} - w_i \cdot G \\ \Gamma_i^3 &:= w_i \cdot R \end{aligned}$$
and commits to all three values simultaneously by sending $(\texttt{commit}, \mathsf{id}_{i,2}^{\mathsf{com}}, (\Gamma_i^1, \Gamma_i^2, \Gamma_i^3), \mathbf{P})$ to $\mathcal{F}^n_{\mathsf{Com}}$, using a fresh value for $\mathsf{id}_{i,2}^{\mathsf{com}}$.
10) Upon being notified of all other parties' commitments, $\mathcal{P}_i$ sends $(\texttt{decommit}, \mathsf{id}_{i,1}^{\mathsf{com}})$ and $(\texttt{decommit}, \mathsf{id}_{i,2}^{\mathsf{com}})$ to $\mathcal{F}^n_{\mathsf{Com}}$ and collects $\{(\phi_j, \Gamma_j^1, \Gamma_j^2, \Gamma_j^3)\}_{j\in\mathbf{P}\setminus\{i\}}$ as the other parties do the same.
11) Each party $\mathcal{P}_i$ computes
$$\phi := \prod_{j\in\mathbf{P}} \phi_j$$
and aborts if
$$\sum_{j\in\mathbf{P}} \Gamma_j^1 \neq \phi \cdot G \ \vee\ \phi = 0$$
$$\vee\ \sum_{j\in\mathbf{P}} \Gamma_j^2 \neq 0 \ \vee\ \sum_{j\in\mathbf{P}} \Gamma_j^3 \neq \phi \cdot \mathsf{pk}$$

Signing:

12) Each party $\mathcal{P}_i$ calculates
$$\mathsf{sig}_i := \frac{H(m) \cdot v_i + r_x \cdot w_i}{\phi}$$
and broadcasts sig_i.
13) Each party computes
$$\mathsf{sig} := \sum_{i\in\mathbf{P}} \mathsf{sig}_i \quad\text{and}\quad \sigma := (\mathsf{sig}, r_x)$$
where $(r_x, r_y) = R$, and outputs σ if $\mathsf{Verify}(\mathsf{pk}, \sigma) = 1$.

Theorem V.1. *The protocols $\pi^{t,n}_{\mathsf{ECDSA\text{-}Setup}}$ and $\pi^{t,n}_{\mathsf{ECDSA\text{-}Sign}}$ UC-realize the functionality $\mathcal{F}^{t,n}_{\mathsf{ECDSA}}$ for the elliptic curve group $(\mathbb{G}, G, q)$ in the $(\mathcal{F}^{\ell,t,n}_{\mathsf{Mul}}, \mathcal{F}^{\ell}_{\mathsf{2PMul}}, \mathcal{F}^n_{\mathsf{Com}}, \mathcal{F}^{R_{\mathsf{DL}},n}_{\mathsf{Com\text{-}ZK}})$-hybrid Random Oracle Model, in the presence of a malicious adversary statically corrupting up to $t-1$ parties, if the Computational Diffie-Hellman problem is hard in $\mathbb{G}$.*

Security. In the following paragraphs we informally discuss the security properties of $\pi^{t,n}_{\mathsf{ECDSA\text{-}Sign}}$, in order to give the reader an intuitive notion of the attacks that are possible when a malicious adversary corrupts a majority of parties, and the way in which the consistency check acts to prevent them. This section serves only to develop an intuition; we provide a formal discussion and a proof of Theorem V.1 in the full version of this paper. We make the simplifying assumption that the adversary always corrupts $t-1$ parties (that is, all but one), which are indexed by the vector $\mathbf{P}^* \subset \mathbf{P}$, but note that our discussion applies equally well to weaker adversaries. We designate the remaining honest party as $\mathcal{P}_h$ with index h.

As our protocol is built atop $\mathcal{F}^{\ell,t,n}_{\mathsf{Mul}}$, $\mathcal{F}^{\ell}_{\mathsf{2PMul}}$, $\mathcal{F}^n_{\mathsf{Com}}$, and $\mathcal{F}^{R_{\mathsf{DL}},n}_{\mathsf{Com\text{-}ZK}}$, we assume secure instantiations of those functionalities are available, and analyze the remaining space of attacks, which involve corrupted parties supplying inconsistent inputs to these functionalities and thereby either producing a signature for an unexpected message, or learning some information about the honest parties' secrets. Specifically, our task will be to argue that if inconsistent inputs are supplied to the various instances of these functionalities, then the consistency checks fail and the remaining honest party aborts with overwhelming probability. To this end, we can divide the class of attacks into two main subclasses: inconsistent inputs to the instance key exchange, and inconsistent inputs to the secret key multiplication.

- *Inconsistent inputs to Instance Key Multiplication.* Recall that in their interactions with $\mathcal{F}^{\ell,t,n}_{\mathsf{Mul}}$, the pool of corrupted parties are represented by an ideal adversary $\mathcal{S}^{\mathbf{P}^*}_{\mathsf{Mul}}$, which submits to the functionality a single unified input for the entire pool. Suppose we define $k_{\mathbf{P}^*}$ to be the first value in the batch supplied by $\mathcal{S}^{\mathbf{P}^*}_{\mathsf{Mul}}$ to $\mathcal{F}^{\ell,t,n}_{\mathsf{Mul}}$ in step 2 of $\pi^{t,n}_{\mathsf{ECDSA\text{-}Sign}}$, and $\phi_{\mathbf{P}^*}$ to likewise be the product of the values submitted by the corrupt parties to $\mathcal{F}^{n}_{\mathsf{Com}}$ in step 1. We can then define the error value e_a such that $\phi_{\mathbf{P}^*}/k_{\mathbf{P}^*}+e_a$ is the second input in the batch supplied by $\mathcal{S}^{\mathbf{P}^*}_{\mathsf{Mul}}$ to $\mathcal{F}^{\ell,t,n}_{\mathsf{Mul}}$ in step 2. Let $u_{\mathbf{P}^*}$ and $v_{\mathbf{P}^*}$ be the unified (that is, summed) outputs corresponding to the first and second inputs in this step, respectively. This yields

$$u_h + u_{\mathbf{P}^*} = k_h \cdot k_{\mathbf{P}^*}$$
$$v_h + v_{\mathbf{P}^*} = \frac{\phi_h \cdot \phi_{\mathbf{P}^*}}{k_h \cdot k_{\mathbf{P}^*}} + \phi_h \cdot e_a$$

In order for the first consistency check to pass, it must be the case that $v_h \cdot R + \Gamma^1_{\mathbf{P}^*} = \phi \cdot G$. If and only if $e_a = 0$, then the adversary can compute the check message $\Gamma^1_{\mathbf{P}^*} := v_{\mathbf{P}^*} \cdot R$ such that this is true. Otherwise, the adversary must compute $\Gamma^1_{\mathbf{P}^*} := v_{\mathbf{P}^*} \cdot R - \phi_h \cdot e_a \cdot R$, where ϕ_h is uniform, and therefore $\phi_h \cdot e_a$ is uniform as well. ϕ_h is revealed to the adversary only after $\Gamma^1_{\mathbf{P}^*}$ must be committed, and therefore the adversary can do no better than to guess, and with overwhelming probability the check message will not verify, causing $\mathcal{P}_h$ to abort.
- *Inconsistent inputs to Secret Key Multiplication.* Suppose that all values are defined as above, and that sk_i is defined as the correct value given $\mathcal{P}_i$'s Shamir share $p(i)$ (itself defined to be the output of the setup protocol). Suppose furthermore that without loss of generality some individual corrupted party $\mathcal{P}^*_i$ supplies as a batched input $\{\mathsf{sk}_i+e_{\mathsf{sk}}, v_i+e_v\}$ when invoking $\mathcal{F}^{\ell}_{\mathsf{2PMul}}$ with $\mathcal{P}_h$ in step 4 of $\pi^{t,n}_{\mathsf{ECDSA\text{-}Sign}}$, receiving as output $\{w^{h,1}_i, w^{h,2}_i\}$. For simplicity, we assume only one corrupted party induces an offset in this way, though the following argument applies equally well when this is not the case. Regardless, we now have

$$w_{\mathbf{P}^*} := \mathsf{sk}_{\mathbf{P}^*} \cdot v_{\mathbf{P}^*} + \sum_{i \in \mathbf{P}^*} w^{h,1}_i + w^{h,2}_i$$

where $w_{\mathbf{P}^*}$ is a unified (i.e. summed over the corrupt parties) version of the output specified in step 5 of $\pi^{t,n}_{\mathsf{ECDSA\text{-}Sign}}$. This yields the relation

$$w_h + w_{\mathbf{P}^*} = \frac{\mathsf{sk} \cdot \phi}{k} + e_{\mathsf{sk}} \cdot \frac{\phi_h}{k_h} + e_v \cdot \mathsf{sk}_h$$

In the case that $e_{\mathsf{sk}} \neq 0$, the adversary can only pass the second consistency check by computing the corrupted parties' values such that their sum $\Gamma^2_{\mathbf{P}^*}$ is

$$\Gamma^2_{\mathbf{P}^*} = v_{\mathbf{P}^*} \cdot \mathsf{pk} + w_{\mathbf{P}^*} \cdot G - e_{\mathsf{sk}} \cdot \frac{\phi_h}{k_h} \cdot G$$

where the adversary knows $R_h = k_h \cdot G$, but neither k_h itself, nor ϕ_h, and both k_h and ϕ_h are uniform. As before, ϕ_h is revealed to the adversary only after $\Gamma^2_{\mathbf{P}^*}$ must be committed, and therefore $\mathcal{P}_h$ will abort with overwhelming probability when $e_{\mathsf{sk}} \neq 0$. In the case that $e_v \neq 0$, the adversary can only pass the third consistency check by computing the corrupted parties' values such that their sum $\Gamma^3_{\mathbf{P}^*}$ is

$$\Gamma^3_{\mathbf{P}^*} = w_{\mathbf{P}^*} \cdot R - e_v \cdot \mathsf{sk}_h \cdot R$$

where the adversary knows

$$\mathsf{sk}_h \cdot G = \mathsf{pk} - \sum_{i \in \mathbf{P}^*} \mathsf{sk}_i \cdot G$$

but the adversary does not know sk_h itself. Computing $\mathsf{sk}_h \cdot R$ given only R and $\mathsf{sk}_h \cdot G$ is a direct violation of the Computational Diffie-Hellman Assumption, and therefore under that assumption $\mathcal{P}_h$ will abort with overwhelming probability when $e_v \neq 0$.

This covers every opportunity available to an adversary for supplying inconsistent inputs, and so we conclude that if the adversary does such a thing, then it will fail to pass at least one of the checks, assuming that the Computational Diffie-Hellman Problem is hard in the elliptic curve group $\mathbb{G}$. A formal treatment of security can be found in the full version.

Round Count. For readability, we expressed the protocol $\pi^{t,n}_{\mathsf{ECDSA\text{-}Sign}}$ in individual steps, but many of these can be collapsed together in practice to reduce the number of rounds. In particular, the process of committing to ϕ_i for $i \in \mathbf{P}$ is independent of and can be performed simultaneously with the first round of preprocessing for $\pi^{\ell,t,n}_{\mathsf{Mul}}$ (which realizes $\mathcal{F}^{\ell,t,n}_{\mathsf{Mul}}$), and the preprocessing for all instances of $\pi^{\ell}_{\mathsf{2PMul}}$ (which realizes $\mathcal{F}^{\ell}_{\mathsf{2PMul}}$) can be moved forward to occur at the same time. The round-count optimization originally described in Section III can be applied, reducing the remaining rounds required by $\pi^{\ell}_{\mathsf{2PMul}}$ to $\log(t)$. Following this, a single round is required to complete all instances of $\pi^{\ell}_{\mathsf{2PMul}}$ simultaneously. The process of committing to R_i and an associated proof of knowledge for $i \in \mathbf{P}$ is independent of the secret key multiplication, and thus can it can also be performed immediately after $\pi^{\ell}_{\mathsf{2PMul}}$ completes. Another round is required to decommit, and two more to commit to and then release the check messages. Finally, the last round is used to swap shares of the signature. Thus the total round count for ECDSA signing comes to $\log(t) + 6$.

VI. Cost Analysis

In Table I we provide an accounting of the communication costs for our ECDSA setup and signing protocols, and in Table II we account for the costs of our multiplication protocols. Round counts for our protocols are discussed at length in the relevant sections. Our equations assume that elements from $\mathbb{Z}_q$ are represented in κ bits, and that curve points are transmitted with point compression and thus are represented in $\kappa + 1$ bits. We assume that commitments require transmission of a single element from $\mathbb{Z}_q$, that decommitments consist simply of the committed values, and that zero-knowledge proofs of knowledge of discrete logarithm comprise two curve points

Phase	Communication (Bits)
Setup	$\frac{n^2-n}{2} \cdot (5\kappa^2 + 6\kappa + 2) + 4\kappa \cdot n + 2n$
Signing	$\frac{t^2-t}{2} \cdot (9\kappa^2 + 18\kappa \cdot s + \kappa \cdot \kappa^{\mathsf{OT}} + 30\kappa + 10)$

TABLE I: **Overall Communication Cost Equations for ECDSA**. The costs assume that the $\mathcal{F}^{\eta}_{\mathsf{COTe}}$ functionality is realized via the protocol of Keller et al. [29] (which introduces an additional security parameter, κ^{OT}) with the VSOT protocol [1] supplying base-OTs.

	Offline (Setup) Costs	
Protocol	Rounds	Communication (Bits)
$\pi^{\ell}_{\mathsf{2PMul}}$	5	$\kappa \cdot (5\kappa + 4) + 2$
$\pi^{\ell,t,n}_{\mathsf{Mul}}$	5	$\frac{n^2-n}{2} \cdot \kappa \cdot (5\kappa + 4) + 2$
	Online Costs	
$\pi^{\ell}_{\mathsf{2PMul}}$	3	$\kappa \cdot (2\xi \cdot \ell + \xi + 3\ell + \kappa^{\mathsf{OT}} + 2)$
$\pi^{\ell,t,n}_{\mathsf{Mul}}$	$\log(t)$+ 2	$\frac{t^2-t}{2} \cdot \kappa \cdot (2\xi \cdot \ell + \xi + 3\ell + \kappa^{\mathsf{OT}} + 2)$

TABLE II: **Communication Cost Equations for Subprotocols**. The costs assume that the $\mathcal{F}^{\eta}_{\mathsf{COTe}}$ functionality is realized via the protocol of Keller et al. [29] with the VSOT protocol [1] supplying base-OTs. In this table we do not consider the round-reducing optimization for $\pi^{\ell}_{\mathsf{2PMul}}$. Note that ℓ is the multiplication batch size, and ξ is the size of the encoding used by Bob, in bits.

and a single element from $\mathbb{Z}_q$, along with the point for which knowledge of discrete logarithm is to be proven.

The signing protocol $\pi^{t,n}_{\mathsf{ECDSA\text{-}Sign}}$ contains one execution of the $\pi^{\ell}_{\mathsf{2PMul}}$ protocol for each pair of parties with $\ell = 4$; see Section VII for a discussion of the optimization that allows this, as opposed to the two executions each with $\ell = 2$ that would be suggested by the protocols as previously described. In addition, each party that participates in the sigature must broadcast commitments and decommitments to ϕ_i, R_i, a proof of knowledge of discrete logarithm for R_i, and check messages Γ^1_i, Γ^2_i, and Γ^3_i. Finally, each party must broadcast its signature share sig_i. The commitments are coalesced such that only three calls to $\mathcal{F}^{n}_{\mathsf{Com}}$ are required, each adding κ bits of communication to the cost of the value committed.

For the setup protocol $\pi^{t,n}_{\mathsf{ECDSA\text{-}Setup}}$, the bulk of the communication cost comes from the initialization of OT-extensions, which will later be used by $\pi^{\ell}_{\mathsf{2PMul}}$ and (by proxy) $\pi^{\ell,t,n}_{\mathsf{Mul}}$. The only other elements transmitted during setup are polynomial points $p_j(i)$ from every party $\mathcal{P}_j$ to every $\mathcal{P}_i$, and for every $\mathcal{P}_i$ a commitment and decommiment to the curve point T_i and a corresponding proof of knowledge of T_i's discrete logarithm.

Concretely, for $\kappa = 256, s = 80$, and $\kappa^{\mathsf{OT}} = 128 + s$ (an additional security parameter for OT-extension [29]), the communication cost for signing is roughly $64.7 \cdot t \cdot (t-1)$ kilobytes, and for setup roughly $20.5 \cdot n \cdot (n-1) + 0.1 \cdot n$ kilobytes. As an example, for $n = 16$ and $t = 8$, setup requires 15291 KB of communication and signing requires 3571 KB.

VII. Implementation

We created proof-of-concept implementations of our t-of-n setup and signing protocols in the Rust language, which are derived from the open source 2-of-n implementations of Doerner et al. [1]. Our implementation uses the secp256k1 curve, as standardized by NIST [5]. Thus, for all benchmarks, $\kappa = 256$; additionally, we chose $s = 80$. We instantiated the $\mathcal{F}^{\eta}_{\mathsf{COTe}}$ functionality using the protocol of Keller et al. [29] and set the OT-extensions security parameter $\kappa^{\mathsf{OT}} = 128 + s$, following their analysis. We chose, as Doerner et al. did, to instantiate $\mathcal{F}^{R_{\mathsf{DL}},n}_{\mathsf{Com\text{-}ZK}}$ via the Fiat-Shamir Heuristic (though we note that this transform is *not* UC-secure), and to instantiate the PRG, the random oracle H, and the commitment functionality $\mathcal{F}^{n}_{\mathsf{Com}}$ via SHA-256. Consequently, our protocol uses the same concrete hash function as specified in the ECDSA standard.

We note that while the folkloric hash-based instantiation of $\mathcal{F}^{n}_{\mathsf{Com}}$ (i.e. $H(m\|r)$ where m is the message, and $r \leftarrow \{0,1\}^{\kappa}$) requires a random nonce to be appended to the message in order to hide the message regardless of its distribution, in our protocol all committed messages have sufficient entropy that the nonce can be omitted.

Unlike Doerner et al., we do not parallelize vectors of hashing operations. Instead, each party parallelizes its interactions with its counterparties (and the computations that they require), using a number of threads equal to the number of parties, or a specified maximum, whichever is smaller. Additionally, the pairwise OT-extension initialization required by our setup protocol is parallelized among an number of threads equal to the number of parties. While we have assumed throughout this paper that the setup protocol can parallelize key-generation and OT-extension initialization, our implementation runs these two phases sequentially, and thus the practical round count is increased from five to eight.

In our signing protocol, as described in Section V, the parties instantiate both the $\mathcal{F}^{\ell}_{\mathsf{2PMul}}$ and $\mathcal{F}^{\ell,t,n}_{\mathsf{Mul}}$ functionalities with batch sizes of $\ell = 2$. Within the $\pi^{\ell,t,n}_{\mathsf{Mul}}$ protocol that realizes $\mathcal{F}^{\ell,t,n}_{\mathsf{Mul}}$, $\mathcal{F}^{\ell}_{\mathsf{2PMul}}$ is instantiated a second time by all pairs of parties, again with a batch size $\ell = 2$. Observe that the preprocessing for both sets of $\mathcal{F}^{\ell}_{\mathsf{2PMul}}$ instances can be performed simultaneously, and that when $\pi^{\ell}_{\mathsf{2PMul}}$ is used to realize $\mathcal{F}^{\ell}_{\mathsf{2PMul}}$ it is feasible for the parties to provide inputs and produce outputs for each element in a batch independently. In our implementation, we combine the batches and use only a single instance of the $\pi^{\ell}_{\mathsf{2PMul}}$ protocol for each pair of parties, with a batch size $\ell = 4$. This allows us to perform only one OT-extension operation, and thereby save the overhead associated with a second.

We benchmarked our implementation using a set of Google Cloud Platform `n1-highcpu-8` nodes, each running Ubuntu 18.04 with kernel 4.15.0. Each node of this type has four physical cores clocked at 2.0 GHz, and is capable of executing eight threads simultaneously. These machines are slightly slower than those used by Doerner et al. [1], and thus the timings we report for their protocol are slightly slower than they report themselves. Each party participating in a benchmark was

n/t Range	n/t Step	Samples (Signing)	Samples (Setup)
$[2, 8]$	1	16000	2000
$(8, 16]$	2	8000	1000
$(16, 32]$	4	4000	500
$(32, 64]$	8	2000	250
$(64, 128]$	16	1000	125
$(128, 256]$	32	500	62

TABLE III: **LAN Benchmark Parameters**. For signing we varied t according to these parameters, and for setup we varied n, fixing $t = \lfloor (n+1)/2 \rfloor$.

allocated one node, and the parties communicated via Google's internal network. We compiled our code using the nightly version of Rust 1.28, with the default level of optimization. Parallelism was provided by the Rayon crate and, as each node can execute eight threads simultaneously, we limited the number of threads used in signing to ten (having arrived at this number empirically). Our hash function implementations were written in C using compiler intrinsics, and were compiled with GCC 8.2.0. Our benchmarking programs were designed to establish insecure connections among the parties one time only, and then run a batch of setup or signing operations, measuring the wall clock time for the entire batch. Thus, they record overhead due to latency and bandwidth constraints, but they do not record overhead due to private or authenticated channels.

A. LAN Benchmarks

For benchmarks in the LAN setting, we created a set of 256 nodes in Google's South Carolina datacenter. Among these nodes, we measured the bandwidth to be generally between 5 and 10 Gbits/sec, and the round-trip latency to be approximately 0.3 ms. Using these nodes, we collected data for both our setup and signing protocols using combinations of parameters as specified in Table III. For signing benchmarks, all costs are independent of n, the number of parties in the larger group from whom the signing parties are selected. Consequently, we varied only t, the number of parties actually participating in signing. For setup, only computation costs depend upon t, and not bandwidth; consequently we varied n and set $t = \lfloor (n+1)/2 \rfloor$, which we determined to be the most expensive value relative to a particular choice of n. Our aim in choosing sample counts was to ensure each benchmark took five to ten minutes in total, in order to smooth out artifacts due to transient network conditions. Our results for setup are reported in Figure 2, and our results for signing are reported in Figure 3.

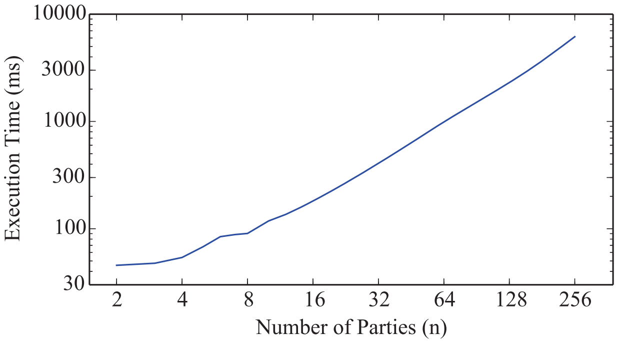

Fig. 2: **Wall Clock Times for n-Party Setup over LAN**. Note that all parties reside on individual machines in the same datacenter, and latency is on the order of a few tenths of a millisecond.

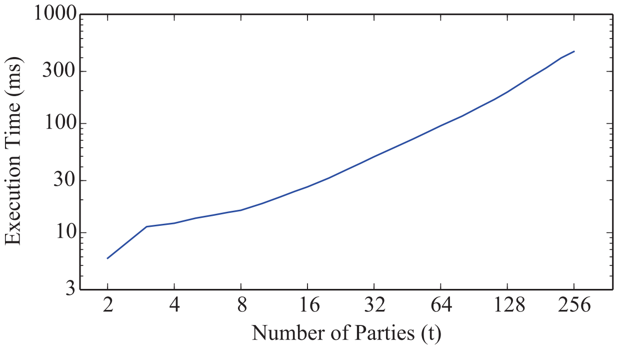

Fig. 3: **Wall Clock Times for t-Party Signing over LAN**. Note that all parties reside on individual machines in the same datacenter, and latency is on the order of a few tenths of a millisecond.

We note that our method only slightly underperforms that of Doerner et al. [1] for 2-of-n signing in this setting, in spite of the fact that our protocol implements a somewhat stronger functionality. Specifically, we require 9.52 ms, whereas an evaluation of their protocol (with no parallelism) in our benchmarking environment requires 5.83 ms. In a similar benchmark environment, but without parallelism, the 2-of-2 protocol of Lindell [12] was reported to require 36.8 ms to sign with only two parties. Allowing parallelism, our protocol is capable of signing with 24 parties in 37.6 ms, or roughly the same time envelope. The most efficient prior works for threshold ECDSA signing with arbitrary thresholds are those of Gennaro et al. [10] and Boneh et al. [11] (who provide an improved implementation Gennaro et al.'s protocol in addition to developing new techniques). As with Lindell's protocol, we did not benchmark their protocols in our environment, and so no truly fair comparison is possible. However, Boneh et al. provide benchmarks for both protocols among groups of parties ranging in size from 2 to 20, with each party residing on a single four-core, eight-thread machine, and no network costs recorded. Gennaro et al.'s protocol is the more efficient of the two in the case of 2-of-n signing, and requires roughly 350 ms. For 20-of-n signing, Boneh et al.'s protocol is the more efficient of the two, requiring roughly 1.5 seconds. While it is true that their benchmark environments differs from ours, our results are factors of roughly 40 and 50 better than theirs. We do not believe that environmental differences account for this.

Among the prior works, only Lindell reports on setup performance. In the 2-of-2 case, his protocol requires 2435 ms, whereas in the 2-of-n case our protocol requires only 45 ms. Even in the 128-of-n case, our setup protocol requires only

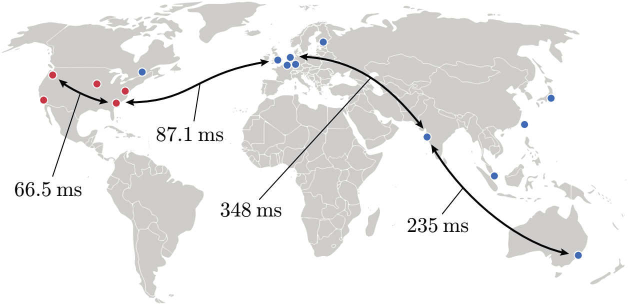

Fig. 4: **Map of Datacenter Locations used for WAN Benchmarks**, with latency figures along a few of the longer routes. The subgroup of five zones inside the US are highlighted in red.

Parties/Zones	Signing Rounds	Signing Time	Setup Time
5/1	9	13.6	67.9
5/5	9	288	328
16/1	10	26.3	181
16/16	10	3045	1676
40/1	12	60.8	539
40/5	12	592	743
128/1	13	193.2	2300
128/16	13	4118	3424

TABLE IV: **Wall-clock Times in Milliseconds over WAN**. The benchmark configurations used are described in Section VII-B. For signing we varied t according to these parameters, and for setup we varied n, fixing $t = \lfloor(n+1)/2\rfloor$. Benchmarks involving only a single zone are LAN benchmarks, for comparison.

2299 ms. It seems that the performance of the setup protocols of Gennaro et al. and Boneh et al. has never been reported, but Lindell [12] conjectures that they would require many minutes.

B. WAN Benchmarks

As we have previously noted, our protocol is at a disadvantage relative to prior works in terms of round count. In order to demonstrate the practical implications of this fact, we ran an additional benchmark in the WAN setting. We chose 16 Google datacenters (otherwise known as *zones*) that offer instances with current-generation CPUs; these are located on a map in Figure 4. Five were located inside the United States, in South Carolina, Virginia, Oregon, California, and Iowa. Among these, the longest leg was between Oregon and South Carolina, with a round-trip latency of 66.5 ms and bandwidth of 353 Mbits/sec. The remaining 11 were located in Montreál, London, Frankfurt, Belgium, the Netherlands, Finland, Sydney, Taiwan, Tokyo, Mumbai, and Singapore. Among the complete set, the longest leg was between Belgium and Mumbai, with a round-trip latency of 348 ms and a bandwidth of 53.4 MBits/sec. We tested two configurations: one with only the five US datacenters participating, and another with all 16. For each configuration, we performed benchmarks with one party in each participating datacenter, and with eight parties in each participating datacenter. In all cases, we collected 125 samples. Results are reported in Table IV, along with comparative data from our LAN benchmarks.

It is worth noting that Wang et al. [31] recently made the claim that they performed the largest-scale demonstration of multiparty computation to date. Their benchmark involves 128 parties split among eight datacenters around the world, who jointly compute an AES circuit using the actively-secure multiparty garbling protocol that they developed. Our WAN benchmark involves 128 parties split among 16 datacenters, and thus we assert that we have also evaluated one of the largest secure multiparty protocols to date, at least so far as party count and geographic distribution are concerned. We also note that the in the clear setting, AES is generally considered to have a much lower circuit complexity than ECDSA; this is reflected in the significantly lower computation time for a single AES operation as compared to signing a single message using ECDSA. Interestingly, in the context of evaluating these primitives securely among multiple parties, our protocol for realizing $\mathcal{F}^{t,n}_{\mathsf{ECDSA}}$ performs considerably better than Wang et al.'s realization of $\mathcal{F}^{n}_{\mathsf{AES}}$. In the LAN setting with 128 parties (each much more powerful than the ones we employ), they report a 17-second wall clock time, including preprocessing, and in the global WAN setting with 128 parties, their protocol requires 2.5 minutes. When the setup and signing costs are combined for our protocol, it requires 2.5 seconds and 7.5 seconds with 128 parties in the LAN and global WAN settings, respectively. We believe that this serves to demonstrate that there are multiparty functionalities for which specially tailored protocols are warranted in practice, as opposed to the blind use of generic MPC for all tasks.

C. Low-power Benchmarks

Finally, we performed a set of benchmarks on a group of three Raspberry Pi model 3B+ single-board computers in order to demonstrate the feasibility of evaluating our protocol (and the protocols of Doerner et al. [1]) on small, low-powered devices. Each board has a single, quad-core ARM-based processor clocked at 1.4 GHz. The boards were loaded with Raspbian Linux (kernel 4.14) and connected to one another via ethernet. As an optimization for the embedded setting, we abandoned SHA-256 (except where required by ECDSA) in favor of the BLAKE2 hash function [32], using assembly implementations provided by the BLAKE2 authors. To simulate the setting wherein an embedded device signs with a more powerful one, we used a 2013 15" Macbook Pro running Mac OS 10.13 (i.e. one author's laptop). This machine was engaged in other tasks at the time of benchmarking, and no attempt was made to prevent this. We benchmarked 2-of-2 signing and setup between the Macbook and a single Raspberry Pi, and t-of-n setup and signing among the group of Pis, with n set as 3 and t as both 2 and 3. When $n = 2$, we used the slightly more efficient protocols of Doerner et al. [1] without modification, and when $t = 3$ we used the protocols presented in this paper. For setup, we collected 50 samples, and for signing, we collected 250. Results are presented in Table V. We observe that in spite of

Configuration	Benchmark	Setup Time	Signing Time
Macbook/RPi	2-of-2	1419	52.6
2×RPi	2-of-2	1960	58.5
2×RPi	2-of-n	–	69.8
3×RPi	3-of-3	2277	162

TABLE V: **Wall-clock Times in Milliseconds for Raspberry Pi**. The benchmark configurations used are described in Section VII-C.

the limitations of the hardware on which these benchmarks were run, the signing time remains much less than a second, and setup requires only a few seconds. Thus we expect our protocol to be computationally efficient enough to run even on embedded devices such as hardware tokens or smartwatches, and certainly on more powerful mobile devices such as phones.

VIII. Code and Full Version

Our implementation is available under the three-clause BSD license from https://gitlab.com/neucrypt/mpecdsa. For the full version of this paper, please visit http://neucrypt.org.

IX. Acknowledgments

We thank Dennis Giese for providing the hardware used in our low-power device benchmark.

References

[1] J. Doerner, Y. Kondi, E. Lee, and a. shelat, “Secure two-party threshold ecdsa from ecdsa assumptions,” in *IEEE S&P*, 2018.

[2] Y. Desmedt, “Society and group oriented cryptography: A new concept,” in *CRYPTO*, 1987.

[3] National Institute of Standards and Technology, “FIPS PUB 186-4: Digital Signature Standard (DSS),” http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.186-4.pdf, 2013.

[4] American National Standards Institute, “X9.62: Public Key Cryptography For The Financial Services Industry: The Elliptic Curve Digital Signature Algorithm (ECDSA),” 2005.

[5] D. R. L. Brown, “Sec 2: Recommended elliptic curve domain parameters,” 2010. [Online]. Available: http://www.secg.org/sec2-v2.pdf

[6] D. Kravitz, “Digital signature algorithm,” jul 1993, uS Patent 5,231,668.

[7] Bitcoin Wiki, “Transaction,” https://en.bitcoin.it/wiki/Transaction, 2017, accessed Oct 22, 2017.

[8] G. Wood, “Ethereum: A secure decentralised generalised transaction ledger,” 2017. [Online]. Available: https://ethereum.github.io/yellowpaper/paper.pdf

[9] P. MacKenzie and M. K. Reiter, “Two-party generation of dsa signatures,” in *CRYPTO*, 2001.

[10] R. Gennaro, S. Goldfeder, and A. Narayanan, *Threshold-Optimal DSA/ECDSA Signatures and an Application to Bitcoin Wallet Security*, 2016.

[11] D. Boneh, R. Gennaro, and S. Goldfeder, “Using level-1 homomorphic encryption to improve threshold dsa signatures for bitcoin wallet security,” in *LATINCRYPT*, 2017.

[12] Y. Lindell, “Fast secure two-party ecdsa signing,” in *CRYPTO*, 2017.

[13] P. Paillier, “Public-key cryptosystems based on composite degree residuosity classes,” in *EUROCRYPT*, 1999.

[14] N. Gilboa, “Two party rsa key generation,” in *CRYPTO*, 1999.

[15] W. Diffie and M. Hellman, “New directions in cryptography,” *IEEE Trans. Inf. Theor.*, 1976.

[16] V. Shoup, “Lower bounds for discrete logarithms and related problems,” in *EUROCRYPT*, 1997.

[17] D. R. L. Brown, “Generic groups, collision resistance, and ECDSA,” *Des. Codes Cryptography*, 2005.

[18] D. Boneh and M. Zhandry, “Multiparty key exchange, efficient traitor tracing, and more from indistinguishability obfuscation,” in *CRYPTO*, 2014.

[19] A. Joux, “A one round protocol for tripartite diffie-hellman,” *J. Cryptol.*, 2004.

[20] G. J. Simmons, “The prisoners’ problem and the subliminal channel,” in *CRYPTO*, 1983.

[21] A. Shamir, “How to share a secret,” *Commun. ACM*, 1979.

[22] O. Goldreich, S. Micali, and A. Wigderson, “How to play any mental game,” in *STOC*, 1987.

[23] J. Katz and Y. Lindell, *Introduction to Modern Cryptography, Second Edition*. Chapman & Hall/CRC, 2015, ch. Digital Signature Schemes, pp. 443–486.

[24] R. Canetti, “Universally composable security: A new paradigm for cryptographic protocols,” in *FOCS*, 2001.

[25] C.-P. Schnorr, “Efficient identification and signatures for smart cards,” in *CRYPTO*, 1989.

[26] A. Fiat and A. Shamir, “How to prove yourself: Practical solutions to identification and signature problems,” in *CRYPTO*, 1986.

[27] M. Fischlin, “Communication-efficient non-interactive proofs of knowledge with online extractors,” in *CRYPTO*, 2005.

[28] D. Beaver, “Correlated pseudorandomness and the complexity of private computations,” in *STOC*, 1996.

[29] M. Keller, E. Orsini, and P. Scholl, “Actively secure OT extension with optimal overhead,” in *CRYPTO*, 2015.

[30] T. Chou and C. Orlandi, “The simplest protocol for oblivious transfer,” in *LATINCRYPT*, 2015.

[31] X. Wang, S. Ranellucci, and J. Katz, “Global-scale secure multiparty computation,” in *CCS*, 2017.

[32] J.-P. Aumasson, S. Neves, Z. Wilcox-O’Hearn, and C. Winnerlein, “Blake2: simpler, smaller, fast as md5,” https://blake2.net/blake2.pdf, 2013.