Secure Two-party Threshold ECDSA from ECDSA Assumptions

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—**The Elliptic Curve Digital Signature Algorithm (ECDSA) is one of the most widely used schemes in deployed cryptography. Through its applications in code and binary authentication, web security, and cryptocurrency, it is likely one of the few cryptographic algorithms encountered on a daily basis by the average person. However, its design is such that executing multi-party or threshold signatures in a secure manner is challenging: unlike other, less widespread signature schemes, secure multi-party ECDSA requires custom protocols, which has heretofore implied reliance upon additional cryptographic assumptions such as the Paillier encryption scheme.**

We propose new protocols for multi-party ECDSA key-generation and signing with a threshold of two, which we prove secure against malicious adversaries in the random oracle model using only the Computational Diffie-Hellman Assumption and the assumptions already implied by ECDSA itself. Our scheme requires only two messages, and via implementation we find that it outperforms the best prior results in practice by a factor of 55 for key generation and 16 for signing, coming to within a factor of 12 of local signatures. Concretely, two parties can jointly sign a message in just over two milliseconds.

I. Introduction

Threshold Digital Signature Schemes are a classic notion in the field of Cryptography [1], which 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 size. Though they are extensively studied, these types of signatures are seldom used in practice, in part because bespoke threshold schemes are incompatible with familiar, widely-accepted signature schemes, and, on the other hand, because threshold techniques for standard signatures tend to be highly inefficient, reliant upon unacceptable assumptions, or otherwise undesirable.

Consider the specific case of the Elliptic Curve Digital Signature Algorithm (ECDSA), perhaps the most widespread of signatures schemes: all existing threshold techniques for generating ECDSA signatures require the invocation of heavy cryptographic primitives such as Paillier encryption [2]–[4]. This leads to both poor performance and to reliance upon assumptions that are foreign to the mathematics on which ECDSA is based. This is troublesome, because performance concerns and avoidance of certain assumptions often motivate the use of ECDSA in the first place. We address this shortcoming by devising the first threshold signing algorithm for ECDSA that is based solely upon Elliptic Curves and the assumptions that the ECDSA signature scheme itself already makes. Furthermore, we improve upon the performance of previous works by a factor of sixteen or more.

Notionally introduced by Diffie and Hellman [5] and first formulated and proven by Goldwasser *et al.* [6], Digital Signature Schemes allow one party (the signer) who holds a *secret key* to convince anyone who holds the matching *public key* that a message is authentic (i.e. that it cannot have been altered since it was signed) and non-repudiable (i.e. that no one other than the signer could have signed it). Signature schemes achieve this through the property of *existential unforgeability* against adaptive chosen-message attacks. That is, an adversary is allowed to choose any number of messages for which it may request a signature, but we require that it can never produce a valid signature for a new message on its own unless it has access to the secret key.

ECDSA is a standardized [7]–[9] derivative of the earlier Digital Signature Algorithm (DSA), devised by David Kravitz [10]. Where DSA is based upon arithmetic modulo a prime, ECDSA uses elliptic curve operations over finite fields. Compared to its predecessor, it has the advantage of being more efficient and requiring much shorter key lengths for the same level of security. In addition to the typical use cases of authenticated messaging, code and binary signing, remote login, &c., ECDSA has been eagerly adopted where high efficiency is important. For example, it is used by TLS [11], DNSSec [12], and many cryptocurrencies, including Bitcoin [13] and Ethereum [14].

A t-of-n threshold signature scheme is a set of protocols which allow n parties to jointly generate a single public key, along with n private shares of a joint secret key, and then privately sign messages if and only if t (some predetermined number) of those parties participate in the signing operation. In addition to satisfying the standard properties of signature schemes, it is necessary that threshold signature schemes be secure in a similar sense to other protocols for multi-party computation. That is, it is necessary that no malicious party can subvert the protocols to extract another party's share of the secret key, and that no subset of fewer than t parties can collude to generate signatures.

The concept of threshold signatures originates with the work of Yvo Desmedt [1], who proposed that multi-party and threshold cryptographic protocols could be designed to mirror societal structures, and thus cryptography could take on a new role, replacing organizational policy and social convention with mathematical assurance. Although this laid the motivational groundwork, it was the subsequent work of Desmedt and Frankel [15] that introduced the first true threshold encryption and signature schemes. These are based upon a combination of the well-known ElGamal [16] and Shamir Secret-Sharing [17] primitives, and carry the disadvantage that they require a trusted party to distribute private keys. Pedersen [18] later removed the need for a trusted third party.

The earliest threshold signature schemes were formulated as was convenient for achieving threshold properties; Desmedt and Frankel [15] recognized the difficulties inherent in designing threshold systems for standard signature schemes. Nevertheless, they later returned to the problem [19], proposing a non-interactive threshold system for RSA signatures [20]. This was subsequently improved and proven secure in a series of works [21]–[24]. Threshold schemes were also developed for Schnorr [25], [26] and DSA [27]–[29] signatures. Many of these schemes were too inefficient to be practical, however.

The efficiency and widespread acceptance of the ECDSA signature scheme make it a natural target for similar work, and indeed threshold ECDSA signatures are such a useful primitive that many cryptocurrencies are already implementing a similar concept in an ad-hoc manner [30]. Unfortunately, the design of the ECDSA algorithm poses a unique problem: the fact that it uses its nonce in a multiplicative fashion frustrates attempts to use typical linear secret sharing systems as primitives. The recent works of Gennaro *et al.* [3] and Lindell [2] solve this problem by using *multiplicative* sharing in combination with homomorphic Paillier encryption [31]; the former focuses on the general t-of-n threshold case, with an emphasis on the honest-majority setting, while the latter focuses on the difficult 2-of-2 case specifically. The resulting schemes (and the latter in particular) are very efficient in comparison to previous threshold schemes for plain DSA signatures: Lindell reports that his scheme requires only 37ms (including communication) per signature over the standard P-256 [9] curve.

Unfortunately, both Lindell and Gennaro *et al.*'s schemes depend upon the Paillier cryptosystem, and thus their security relies upon the Decisional Composite Residuosity Assumption. In some applications (crypto-currencies, for example), the choice of ECDSA is made carefully in consideration of the required assumptions, and thus using a threshold scheme that requires new assumptions may not be acceptable. Additionally, if it is to be proven secure via simulation, Lindell's scheme requires a *new* (though reasonable) assumption about the Paillier cryptosystem to be accepted. Furthermore, the Paillier cryptosystem is so computationally expensive that even a single Paillier operation represents a significant cost relative to typical Elliptic Curve operations. Thus in this work we ask whether an efficient, secure, multi-party ECDSA signing scheme can be constructed using only elliptic curve primitives and elliptic curve assumptions, and find the answer in the affirmative.

A. Our Technique

Lindell observes that the problem of securely computing an ECDSA signature among two parties under a public key pk can be reduced to that of securely computing just *two* secure multiplications over the integers modulo the ECDSA curve order q ($\mathbb{Z}_q$). Lindell uses multiplicative shares of the secret key and nonce (hereafter called the instance key), and computes the signature using the Paillier additive homomorphic encryption scheme. We propose a new method to share the products which eliminates the need for homomorphic encryption.

Recall the signing equation for ECDSA,

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

where m is the message, H is a hash function, sk is the secret key, k is the instance key, and r_x is the x-coordinate of the elliptic curve point $R = k \cdot G$ (G being the generator for the curve). Suppose that $k = k_{\mathsf{A}} \cdot k_{\mathsf{B}}$ such that k_{A} and k_{B} are randomly chosen by Alice and Bob respectively, and $R = (k_{\mathsf{A}} \cdot k_{\mathsf{B}}) \cdot G$, and suppose that $\mathsf{sk} = \mathsf{sk}_{\mathsf{A}} \cdot \mathsf{sk}_{\mathsf{B}}$. Alice and Bob can learn R (and thus r_x) securely via Diffie-Hellman exchange, and they receive m as input. Rearranging, we have

$$\mathsf{sig} = H(m)\left(\frac{1}{k_{\mathsf{A}}} \cdot \frac{1}{k_{\mathsf{B}}}\right) + r_x\left(\frac{\mathsf{sk}_{\mathsf{A}}}{k_{\mathsf{A}}} \cdot \frac{\mathsf{sk}_{\mathsf{B}}}{k_{\mathsf{B}}}\right)$$

which identifies the two multiplications on private inputs that are necessary. In our scheme, the results of of these multiplications are returned as *additive* secret shares to Alice and Bob. Since the rest of the equation is distributive over these shares, Alice and Bob can assemble shares of the signature without further interaction. Alice sends her share to Bob, who reconstructs sig and checks that it verifies.

To compute these multiplications, one could apply generic multi-party computation over arithmetic circuits, but generic MPC techniques incur large practical costs in order to achieve malicious security. Instead, we construct a new two-party multiplication protocol, based upon the semi-honest Oblivious-Transfer (OT) multiplication technique of Gilboa [32], which we harden to tolerate malicious adversaries using the structure of the signature scheme itself. Note that even if the Gilboa multiplication protocol is instantiated with a malicious-secure OT protocol, it is vulnerable to a simple selective failure attack whereby the OT sender (Alice) can learn one or more bits of the secret input of the OT receiver (Bob). We mitigate this attack by encoding the Bob's input randomly, such that Alice must learn more than a statistical security parameter number of bits in order to determine his unencoded input.

Unfortunately Bob may also cheat and learn something about Alice's secrets by using inconsistent inputs in the two different multiplication protocols, or by using inconsistent inputs between the multiplications and the Diffie-Hellman exchange. In order to mitigate this issue, we introduce a simple *consistency check* which ensures that Bob's inputs correspond to his shares of the established secret key and instance key. In essence, Alice and Bob combine their shares with the secret key and instance key *in the exponent*, such that if the shares are consistent then they evaluate to a constant value. This check is a novel and critical element of our protocol, and we conjecture that it can be applied to other domains.

Our signing protocol can easily be adapted for threshold signing among n parties with a threshold of two. This requires the addition of a special n-party setup protocol, and the modification of the signing protocol to allow the parties to provide additive shares of their joint secret key rather than multiplicative shares. Surprisingly, however, this modification incurs an overhead equivalent to roughly half of an ordinary multiplication.

B. *Our Contributions*

1) We present an efficient n-party ECDSA key generation protocol and prove it secure in the Random Oracle model under the Computational Diffie-Hellman assumption.
2) We present an efficient two-party, two-round ECDSA signing protocol that is secure under the Computational Diffie-Hellman assumption and the assumption that the resulting signature is itself secure. Since CDH is implied by the Generic Group Model, under which ECDSA is proven secure, we require no additional assumptions relative to ECDSA itself.
3) We formulate a new ideal functionality for multi-party ECDSA signing that permits our signing protocol to achieve much better practical efficiency than it could if it were required to adhere to the standard functionality. We reduce the security of our functionality to the security of the classic signature game in the Generic Group Model.
4) In service of our main protocol, we devise a variant of Gilboa's multiplication by oblivious transfer technique [32] that may be of independent interest. It uses randomized input-encoding along with input commitments to avoid explicit correctness and consistency checks while maintaining security against malicious adversaries.
5) Our multiplication protocol has at its core an oblivious transfer scheme based upon the Simplest OT [33] and KOS [34] OT-extension protocols. We introduce a new check system to avoid the issues that have recently cast doubt on the UC-security of Simplest OT [35].
6) We provide an implementation of our protocol in Rust, and demonstrate its efficiency under real-world conditions. In benchmarks, we find our implementation can produce roughly 475 signatures per second on commodity hardware without parallelism.

C. *Organization*

The remainder of this document is organized as follows. In Section II we review essential concepts and definitions, and in Section III we discuss the ideal functionality that our protocols will realize. In Section IV we specify a basic two-party protocol, which we extend to support 2-of-n threshold signing in Section V. In Section VI we describe the OT and multiplication primitives that we use. In Section VII we present a comparative analysis of our protocols. In Section VIII, we describe our implementation and present benchmark results. In the full version of this paper we prove our protocol secure.

II. Preliminaries and Definitions

A. *Notation and Conventions*

We denote curve points with capitalized variables and scalars with lower case. Vectors are given in bold and indexed by subscripts, while matrices are denoted by bold capitals, with subscripts and superscripts representing row indices and column indices respectively. We use $=$ to denote equality, $:=$ for assignment, and $\leftarrow$ for sampling an instance from a distribution. We use $\overset{c}{\equiv}$ to denote computational indistinguishability, $\overset{s}{\equiv}$ to denote statistical indistinguishability, and for statistical equivalence, we use $\equiv$. Throughout this document, we use κ to represent the security parameter of the elliptic curve over which our equations are evaluated. Likewise we use s for the statistical security parameter.

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, and we assume that the functionality aborts if a party tries to reuse a session ID, send messages out of order, &c. 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. So, 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 [36])**.**
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: $b := \mathsf{Verify}_{\mathsf{pk}}(m, \sigma)$

A Digital Signature Scheme satisfies two properties:

1) (Correctness) With overwhelmingly high probability, all valid signatures must verify. Formally, we require that over $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{Gen}(1^\kappa)$ and all messages m in the message space,
$$\Pr_{\mathsf{pk},\mathsf{sk},m}\left[\mathsf{Verify}_{\mathsf{pk}}(m, \mathsf{Sign}_{\mathsf{sk}}(m)) = 1\right] > 1 - \mathrm{negl}(\kappa)$$
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,
$$\Pr_{\mathsf{pk},\mathsf{sk}}\left[\begin{array}{r}\mathsf{Verify}_{\mathsf{pk}}(m, \sigma) = 1 \land m \notin \mathbf{Q} : \\ (m, \sigma) \leftarrow \mathcal{A}^{\mathsf{Sign}_{\mathsf{sk}}(\cdot)}(\mathsf{pk})\end{array}\right] < \mathrm{negl}(\kappa)$$

C. *ECDSA*

The ECDSA algorithm 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. The algorithm makes use of a hash function $H : \{0,1\}^* \mapsto \mathbb{Z}_q$. Curve coordinates and scalars are represented in $\kappa = \log_2(q)$ bits, which is also the security parameter. A number of standard curves with various security parameters have been promulgated [9]. Assuming a curve has been fixed, the ECDSA algorithms are as follows [36]:

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

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

The initial publication of the ECDSA algorithm did not include a rigorous proof of security; this proof was later provided by Brown [37] in the Generic Group Model, based upon the hardness of discrete logarithms and the assumption that the hash function H is collision resistant and uniform. Vaudenay [38] surveys this and other ECDSA security results, and Koblitz and Menezes provide some analysis and critique of the proof technique [39]. In this work, we simply assume that ECDSA is secure as specified in Definition 1.

D. Oblivious Transfer

Our construction uses a 1-of-2 Oblivious Transfer (OT) system, which is a cryptographic protocol evaluated by two parties: a sender and a receiver. The sender submits as input two private messages, m_0 and m_1; the receiver submits a single bit b, indicating its choice between those two. At the end of the protocol, the receiver learns the message m_b, and the sender learns nothing. In particular, the sender does not learn the value of the bit b, and the receiver does not learn the value of the message $m_{\bar{b}}$. 1-of-2 OT was introduced by Evan *et al.* [40], and is distinct from the earlier Rabin-style OT [41], [42]. For a complete formal definition, we refer the reader to Naor and Pinkas [43]. Beaver [44] later introduced the notion of OT-extension, by which a few instances of Oblivious Transfer can be extended to transfer polynomially many messages using only symmetric-key primitives. For reasons of efficiency, many modern protocols (including our own) use OT-extension rather than plain OT.

III. Two Functionalities

As our scheme is a multi-party computation protocol in the malicious security model, its security will be defined relative to an ideal functionality. Prior works on threshold ECDSA [2], [3] present a functionality $\mathcal{F}_{\mathsf{ECDSA}}$ (Functionality 1) that applies the threshold model directly to the original ECDSA algorithms. The ECDSA Gen algorithm becomes the first phase of $\mathcal{F}_{\mathsf{ECDSA}}$, and the ECDSA Sign algorithm becomes the second.

Functionality 1. $\mathcal{F}_{\mathsf{ECDSA}}$**:**

This functionality is parameterized by a group $\mathbb{G}$ of order q (represented in κ bits) generated by G, as well as hash function $H : \{0,1\}^* \mapsto \mathbb{Z}_q$. The setup phase runs once with a group of parties $\mathbf{P}$ such that $|\mathbf{P}| = n$, and the signing phase may be run many times between any two specific parties from this group, Alice and Bob.

Setup (2-of-*n*): On receiving $(\texttt{init})$ from all parties in $\mathbf{P}$:

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

Signing: On receiving $(\texttt{sign}, \mathsf{sig_{id}}, \mathsf{B}, m)$ from Alice and $(\texttt{sign}, \mathsf{sig_{id}}, \mathsf{A}, m)$ from Bob, if $(\texttt{ready})$ exists in memory but $(\texttt{complete}, \mathsf{sig_{id}})$ does not exist in memory:

```
1) Sample $k \leftarrow \mathbb{Z}_q$ and store it as the instance key.
2) Compute $(r_x, r_y) = R \coloneqq k \cdot G$
3) Compute
   $$\mathsf{sig} \coloneqq \frac{H(m) + \mathsf{sk} \cdot r_x}{k}$$
4) Collect the signature, $\sigma \coloneqq (\mathsf{sig} \mod q, r_x \mod q)$
5) Send $(\texttt{signature}, \mathsf{sig_{id}}, \sigma)$ to Bob.
6) Store $(\texttt{complete}, \mathsf{sig_{id}})$ in memory.
```

Our scheme does not realize $\mathcal{F}_{\mathsf{ECDSA}}$, but instead a new functionality $\mathcal{F}_{\mathsf{SampledECDSA}}$ (Functionality 2), which we have formulated to allow us to build a protocol that requires only two rounds. Of course, it is well known that generic Multi-party Computation can compute any function in two rounds [45], [46] (or even one round, with a complex setup procedure), but the challenge is to do so efficiently. It is natural to use a Diffie-Hellman exchange to compute R, which would otherwise require expensive secure point multiplication techniques, but this precludes either a two-round protocol or use of the standard functionality for an intuitive reason: in the (basic) Diffie-Hellman exchange, Bob sends $D_{\mathsf{B}} \coloneqq k_{\mathsf{B}} \cdot G$ to Alice, who replies to Bob with $D_{\mathsf{A}} \coloneqq k_{\mathsf{A}} \cdot G$. Both Alice and Bob can compute $R \coloneqq k_{\mathsf{A}} \cdot k_{\mathsf{B}} \cdot G$. While Alice cannot learn the discrete logarithm of R, she does have the power to determine R itself due to the fact that she chooses k_{A} after having seen D_{B}. This conflicts with Functionality 1, which requires that the functionality pick R. It is not obvious how to solve this without adding rounds or using a much more expensive primitive, though we conjecture that a more elaborate one-time setup procedure may provide a resolution.

Instead, we have devised $\mathcal{F}_{\mathsf{SampledECDSA}}$. Relative to the previous variant, we divide the signing phase of the functionality into three parts, allowing the parties to abort between them. In the first two parts, Alice and Bob initiate a new signature for a message m, and a random instance key k is chosen by the functionality, along with $R = k \cdot G$, which is returned to Alice. Alice is permitted to request a new sampling of R from the functionality arbitrarily many times (with a negligible chance of receiving a favorable value), and to choose from the sampled set one value under which the signature will be performed. If neither party aborts, then in the third part the functionality will return a signature under the chosen R. This accounts for Alice's ability to manipulate the Diffie-Hellman exchange, and yet it ensures that she does not know the discrete logarithm of the value that is eventually chosen, and that the value is uniform over $\mathbb{G}$.

In Appendix B we prove in the Generic Group Model [47] that $\mathcal{F}_{\mathsf{SampledECDSA}}$ is no less secure than ECDSA itself. However, if reliance on the GGM is undesirable (ECDSA's own reliance notwithstanding) we believe it possible that a three-round variant of our protocol can realize the $\mathcal{F}_{\mathsf{ECDSA}}$ functionality directly.

Functionality 2. $\mathcal{F}_{\mathsf{SampledECDSA}}$**:**

This functionality is parametrized in a manner identical to Functionality 1. Note that Alice may engage in the Offset Determination phase as many times as she wishes.

Setup (2-of-n): On receiving $(\texttt{init})$ from all parties in $\mathbf{P}$:

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 in $\mathbf{P}$.
4) Store $(\texttt{ready})$ in memory.

Instance Key Agreement: On receiving $(\texttt{new}, \mathsf{sig_{id}}, m, \mathsf{B})$ from Alice and $(\texttt{new}, \mathsf{sig_{id}}, m, \mathsf{A})$ from Bob, if $(\texttt{ready})$ exists in memory, and if $(\texttt{message}, \mathsf{sig_{id}}, \cdot, \cdot)$ does not exist in memory, and if Alice and Bob both supply the same message m and each indicate the other as their counterparty, then:

1) Sample $k_{\mathsf{B}} \leftarrow \mathbb{Z}_q$.
2) Store $(\texttt{message}, \mathsf{sig_{id}}, m, k_{\mathsf{B}})$ in memory.
3) Send $(\texttt{nonce-shard}, \mathsf{sig_{id}}, D_{\mathsf{B}} := k_{\mathsf{B}} \cdot G)$ to Alice.

Offset Determination: On receiving $(\texttt{nonce}, \mathsf{sig_{id}}, i, R_i)$ from Alice, if $(\texttt{message}, \mathsf{sig_{id}}, m, k_{\mathsf{B}})$ exists in memory, but $(\texttt{nonce}, \mathsf{sig_{id}}, j, \cdot)$ for $j = i$ does not exist in memory:

4) Sample $k_i^{\Delta} \leftarrow \mathbb{Z}_q$.
5) Store $\left(\texttt{nonce}, \mathsf{sig_{id}}, i, R_i, k_i^{\Delta}\right)$ in memory.
6) Compute $k_{i,\mathsf{A}}^{\Delta} = k_i^{\Delta}/k_{\mathsf{B}}$ and send $(\texttt{offset}, \mathsf{sig_{id}}, k_{i,\mathsf{A}}^{\Delta})$ to Alice.

Signing: On receiving $(\texttt{sign}, \mathsf{sig_{id}}, i, k_{\mathsf{A}})$ from Alice and $(\texttt{sign}, \mathsf{sig_{id}})$ from Bob, if $(\texttt{message}, \mathsf{sig_{id}}, m, k_{\mathsf{B}})$ exists in memory and $\left(\texttt{nonce}, \mathsf{sig_{id}}, j, R_i, k_i^{\Delta}\right)$ for $j = i$ exists in memory, but $(\texttt{complete}, \mathsf{sig_{id}})$ does not exist in memory:

7) Abort if $k_{\mathsf{A}} \cdot k_{\mathsf{B}} \cdot G \neq R_i$.
8) Set $k := k_{\mathsf{A}} \cdot k_{\mathsf{B}} + k_i^{\Delta}$ and store $(r_x, r_y) = R := k \cdot G$.
9) Compute
$$\mathsf{sig} := \frac{H(m) + \mathsf{sk} \cdot r_x}{k}$$
10) Collect the signature, $\sigma := (\mathsf{sig} \mod q, r_x \mod q)$
11) Send $\left(\texttt{signature}, \mathsf{sig_{id}}, R, k_i^{\Delta}, \sigma\right)$ to Bob.
12) Store $(\texttt{complete}, \mathsf{sig_{id}})$ in memory.

IV. A Basic 2-of-2 Scheme

We describe a simplified 2-of-2 version of our scheme initially, abstracting away the multiplication protocols for the sake of clarity. In Section V we extend our scheme to support 2-of-n threshold signing. The fundamental structure of our 2-of-2 scheme is similar to that of Lindell [2] in that the signing protocol ingests multiplicative shares of both the private key and the instance key from each party.

A. Signing

Alice and Bob begin with m, the message to be signed, and multiplicative shares of a secret key ($\mathsf{sk_A}$ and $\mathsf{sk_B}$ respectively), as well as a public key pk that is consistent with those shares. The protocol is divided into four logical steps:

1) **Multiplication:** The parties transform their multiplicative shares of the instance key into additive shares. A second multiplication converts multiplicative shares of the secret key divided by the instance key into additive shares. Due to the presence of the consistency check, the multiplication protocols employed are not required to enforce correctness or consistency of inputs. Although many multiplication protocols are valid candidates, we use the custom OT-based multiplication protocol that we describe in Section VI-C, referred to here as π_{Mul}.
2) **Instance Key Exchange:** The parties calculate $R = k \cdot G$ using a modified Diffie-Hellman exchange.
3) **Consistency Check:** The parties verify that the first multiplication uses inputs consistent with the Instance Key Exchange. This is achieved by adding a random pad ϕ to Alice's input, and then combining the pad with the multiplication output and the known value R in such a way that Bob can retrieve the pad only if he acted honestly. A second check ensures that the multiplications are consistent with each other and with the public key, by combining the multiplication outputs with the public key in the exponent.
4) **Signature and Verification:** The parties reconstruct the signature, which is given to Bob. Bob verifies the signature in the usual way, and, if the signature verifies, then he outputs it.

The instance key exchange component implements the second and third phases of the $\mathcal{F}_{\mathsf{SampledECDSA}}$ functionality (Functionality 2), and the multiplication, consistency check, and verification components implement the fourth phase. Although we make a logical distinction between these four components, in the actual protocol they are intertwined. In particular, we reorder the messages such that all messages from Bob to Alice come first, followed by all messages from Alice to Bob, which results in a two-message protocol. Additionally, rather than perform the consistency check directly, we use its associated message as a key to encrypt all subsequent communications, so that the protocol can only be completed if the consistency check passes. We give the protocol below, and in Figure 1 we provide an illustration, along with annotations indicating the logical component associated with each step.

Protocol 1. Two-party Signing $(\pi^{\mathsf{Sign}}_{\mathsf{2P\text{-}ECDSA}})$:

This protocol is parameterized by the Elliptic curve $(\mathbb{G}, G, q)$ and the hash function $H : \{0,1\}^* \mapsto \mathbb{Z}_q$. It relies upon the subprotocol π_{Mul}. Alice and Bob provide their multiplicative secret key shares $\mathsf{sk}_{\mathsf{A}}, \mathsf{sk}_{\mathsf{B}}$ as input, along with identical copies of the message m, and Bob receives as output a signature σ.

Multiplication and Instance Key Exchange:

1) Bob chooses his secret instance key, $k_{\mathsf{B}} \leftarrow \mathbb{Z}_q$, and Alice chooses her instance key seed, $k'_{\mathsf{A}} \leftarrow \mathbb{Z}_q$. Bob computes

$$D_{\mathsf{B}} := k_{\mathsf{B}} \cdot G$$

and sends D_{B} to Alice.

2) Alice computes

$$R' := k'_{\mathsf{A}} \cdot D_{\mathsf{B}}$$
$$k_{\mathsf{A}} := H(R') + k'_{\mathsf{A}}$$
$$R := k_{\mathsf{A}} \cdot D_{\mathsf{B}}$$

3) Alice chooses a pad $\phi \leftarrow \mathbb{Z}_q$, and then Alice and Bob run the π_{Mul} subprotocol with inputs $\phi + 1/k_{\mathsf{A}}$ and $1/k_{\mathsf{B}}$ respectively, and receive shares t^1_{A} and t^1_{B} of their padded joint inverse instance key

$$t^1_{\mathsf{A}} + t^1_{\mathsf{B}} = \frac{\phi}{k_{\mathsf{B}}} + \frac{1}{k_{\mathsf{A}} \cdot k_{\mathsf{B}}}$$

Alice and Bob also run the π_{Mul} subprotocol with inputs $\mathsf{sk}_{\mathsf{A}}/k_{\mathsf{A}}$ and $\mathsf{sk}_{\mathsf{B}}/k_{\mathsf{B}}$ respectively (that is, their secret key shares multiplied by their inverse instance key shares). They receive shares t^2_{A} and t^2_{B} of their joint secret key over their joint instance key

$$t^2_{\mathsf{A}} + t^2_{\mathsf{B}} = \frac{\mathsf{sk}_{\mathsf{A}} \cdot \mathsf{sk}_{\mathsf{B}}}{k_{\mathsf{A}} \cdot k_{\mathsf{B}}}$$

These two protocol instances are interleaved such that the messages from Bob to Alice are transmitted first, followed by the messages from Alice to Bob.

4) Alice transmits R' to Bob, who computes

$$R := H(R') \cdot D_{\mathsf{B}} + R'$$

For both Alice and Bob let $(r_x, r_y) = R$.

Consistency Check, Signature, and Verification:

5) Alice and Bob both compute $m' = H(m)$.

6) Alice computes the first check value Γ^1, encrypts her pad ϕ with this value, and then transmits the encryption η^ϕ to Bob.

$$\Gamma^1 := G + \phi \cdot k_{\mathsf{A}} \cdot G - t^1_{\mathsf{A}} \cdot R$$
$$\eta^\phi := H(\Gamma^1) + \phi$$

7) Alice computes her share of the signature $\mathsf{sig}_{\mathsf{A}}$ and the second check value Γ^2. She encrypts $\mathsf{sig}_{\mathsf{A}}$ with the second check value and then transmits the encryption η^{sig} to Bob

$$\mathsf{sig}_{\mathsf{A}} := (m' \cdot t^1_{\mathsf{A}}) + (r_x \cdot t^2_{\mathsf{A}})$$
$$\Gamma^2 := (t^1_{\mathsf{A}} \cdot \mathsf{pk}) - (t^2_{\mathsf{A}} \cdot G)$$
$$\eta^{\mathsf{sig}} := H(\Gamma^2) + \mathsf{sig}_{\mathsf{A}}$$

8) Bob computes the check values and reconstructs the signature

$$\Gamma^1 := t^1_{\mathsf{B}} \cdot R$$
$$\phi := \eta^\phi - H(\Gamma^1)$$
$$\theta := t^1_{\mathsf{B}} - \phi/k_{\mathsf{B}}$$
$$\mathsf{sig}_{\mathsf{B}} := (m' \cdot \theta) + (r_x \cdot t^2_{\mathsf{B}})$$
$$\Gamma^2 := (t^2_{\mathsf{B}} \cdot G) - (\theta \cdot \mathsf{pk})$$
$$\mathsf{sig} := \mathsf{sig}_{\mathsf{B}} + \eta^{\mathsf{sig}} - H(\Gamma^2)$$

9) Bob uses the public key pk to verify that $\sigma := (\mathsf{sig}, r_x)$ is a valid signature on message m. If the verification fails, Bob aborts. If it succeeds, he outputs σ.

On the Structure of the Consistency Check: Because the consistency check mechanism is non-obvious, we present an informal justification for it here. In the full version of this paper, we prove the mechanism formally secure. Suppose that we reorganized our protocol to omit Alice's pad ϕ. Then we would have

$$\hat{t^1_{\mathsf{A}}} + \hat{t^1_{\mathsf{B}}} = \frac{1}{k_{\mathsf{A}} \cdot k_{\mathsf{B}}} \qquad t^2_{\mathsf{A}} + t^2_{\mathsf{B}} = \frac{\mathsf{sk}_{\mathsf{A}} \cdot \mathsf{sk}_{\mathsf{B}}}{k_{\mathsf{A}} \cdot k_{\mathsf{B}}}$$
$$(\hat{t^1_{\mathsf{A}}} + \hat{t^1_{\mathsf{B}}}) \cdot \mathsf{pk} = (t^2_{\mathsf{A}} + t^2_{\mathsf{B}}) \cdot G$$

If Bob behaves honestly, he should use $1/k_{\mathsf{B}}$ and $\mathsf{sk}_{\mathsf{B}}/k_{\mathsf{B}}$ as his inputs to the two multiplications. Suppose Bob cheats by using different inputs; without loss of generality, we can interpret his cheating as using inputs $x + 1/k_{\mathsf{B}}$ and $\mathsf{sk}_{\mathsf{B}}/k_{\mathsf{B}}$, in essence offsetting his input for the first multiplication by some value x relative to his input for the second multiplication:

$$\hat{t^1_{\mathsf{A}}} + \hat{t^1_{\mathsf{B}}} = 1/k + x/k_{\mathsf{A}}$$
$$(\hat{t^1_{\mathsf{A}}} + \hat{t^1_{\mathsf{B}}}) \cdot \mathsf{pk} = (t^2_{\mathsf{A}} + t^2_{\mathsf{B}}) \cdot G + x \cdot \mathsf{pk}/k_{\mathsf{A}}$$

and in order to pass the consistency check, Bob would need to calculate $\mathsf{pk}/k_{\mathsf{A}}$, for which the information in his view is not sufficient.

It is tempting to take advantage of the fact that $(\hat{t^1_{\mathsf{A}}} + \hat{t^1_{\mathsf{B}}}) \cdot R = G$ to design a similar mechanism to verify that the first multiplication is consistent with the instance key exchange, but a check based upon this principle is insecure. Again, if we suppose that Bob cheats by offsetting his input for the multiplication by some value x relative to his input for the Diffie-Hellman exchange that produces R, then

$$\hat{t^1_{\mathsf{A}}} + \hat{t^1_{\mathsf{B}}} = 1/k + x/k_{\mathsf{A}}$$

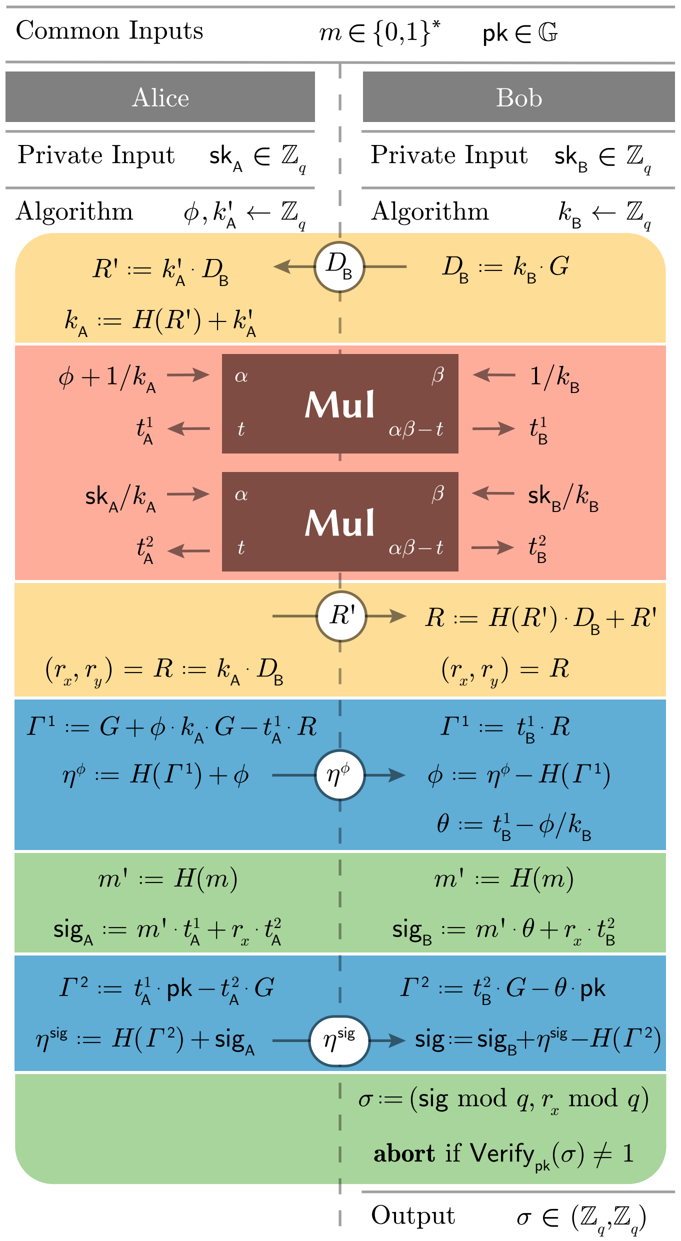

Common Inputs $m \in \{0,1\}^*$ $\mathsf{pk} \in \mathbb{G}$
Alice
Bob
Private Input $\mathsf{sk}_A \in \mathbb{Z}_q$
Private Input $\mathsf{sk}_B \in \mathbb{Z}_q$
Algorithm $\phi, k'_A \leftarrow \mathbb{Z}_q$
Algorithm $k_B \leftarrow \mathbb{Z}_q$
$R' := k'_A \cdot D_B$
D_B
$D_B := k_B \cdot G$
$k_A := H(R') + k'_A$
$\phi + 1/k_A$
α
Mul
β
$1/k_B$
t^1_A
t
$\alpha\beta - t$
t^1_B
sk_A/k_A
α
Mul
β
sk_B/k_B
t^2_A
t
$\alpha\beta - t$
t^2_B
R'
$R := H(R') \cdot D_B + R'$
$(r_x, r_y) = R := k_A \cdot D_B$
$(r_x, r_y) = R$
$\Gamma^1 := G + \phi \cdot k_A \cdot G - t^1_A \cdot R$
$\Gamma^1 := t^1_B \cdot R$
$\eta^\phi := H(\Gamma^1) + \phi$
η^ϕ
$\phi := \eta^\phi - H(\Gamma^1)$
$\theta := t^1_B - \phi/k_B$
$m' := H(m)$
$m' := H(m)$
$\mathsf{sig}_A := m' \cdot t^1_A + r_x \cdot t^2_A$
$\mathsf{sig}_B := m' \cdot \theta + r_x \cdot t^2_B$
$\Gamma^2 := t^1_A \cdot \mathsf{pk} - t^2_A \cdot G$
$\Gamma^2 := t^2_B \cdot G - \theta \cdot \mathsf{pk}$
$\eta^{\mathsf{sig}} := H(\Gamma^2) + \mathsf{sig}_A$
η^{sig}
$\mathsf{sig} := \mathsf{sig}_B + \eta^{\mathsf{sig}} - H(\Gamma^2)$
$\sigma := (\mathsf{sig} \bmod q, r_x \bmod q)$
abort if $\mathsf{Verify}_{\mathsf{pk}}(\sigma) \neq 1$
Output $\sigma \in (\mathbb{Z}_q, \mathbb{Z}_q)$

Fig. 1: **Illustrated Two-party Signing Scheme**. Operations are color-coded according to the logical component with which they are associated: Multiplication, Instance Key Exchange, Consistency Check, and Verification/Signing. We specify how to instantiate the multiplication subprotocol (π_{Mul}) in Section VI-C.

$$(\hat{t^1_A} + \hat{t^1_B}) \cdot R = G + x \cdot k_B \cdot G$$

Unfortunately, the offset produced is made up entirely of elements known to Bob. We rectify this by introducing into the equation a term that Bob cannot predict. Alice intentionally offsets her input to the multiplication using a pad ϕ. If Bob is honest, then

$$t^1_A + t^1_B = 1/k + \phi/k_B$$
$$t^1_B \cdot R = G + \phi \cdot k_A \cdot G - t^1_A \cdot R$$

which implies that both Alice and Bob can compute $t^1_B \cdot R$. On the other hand, if Bob is dishonest, then

$$t^1_A + t^1_B = 1/k + \phi/k_B + x/k_A + x \cdot \phi$$
$$t^1_B \cdot R = G + \phi \cdot k_A \cdot G + x \cdot k_B \cdot G + x \cdot \phi \cdot R - t^1_A \cdot R$$

Because x is unknown to Alice and ϕ is unknown to Bob, neither party is capable of calculating the offset that has been induced. Consequently, if Alice masks ϕ using the value of $t^1_B \cdot R$ that she *expects* Bob to have, then he will be able to remove the mask and retrieve ϕ if and only if he has behaved honestly. Without knowledge of ϕ, he will not be able to pass the second consistency check or reconstruct the signature. We note that there is an assumption of circular security in this construction, which is resolved in our proofs via use of the Random Oracle Model.

B. Setup

We now present a simplified setup protocol for two parties. This protocol does not implement the setup phase of the $\mathcal{F}_{\mathsf{SampledECDSA}}$ functionality, as it does not support threshold signing, but it does provide a similar functionality to the setup protocol of Lindell [2]. In short, it implements the ECDSA Gen algorithm, combining multiplicative secret key shares via a simple Diffie-Hellman [5] key exchange. Proofs of knowledge are necessary in order to ensure that if the protocol completes then the parties are capable of signing, and thus the protocol makes use of both a direct zero-knowledge proof-of-knowledge-of-discrete-logarithm functionality $\mathcal{F}^{R_{\mathsf{DL}}}_{\mathsf{ZK}}$, and a commit-and-prove variant $\mathcal{F}^{R_{\mathsf{DL}}}_{\mathsf{Com\text{-}ZK}}$. These can be concretely instantiated by Schnorr proofs [25] and the Fiat-Shamir [48] or Fischlin [49] transforms. Finally, the protocol initializes the OT-extensions, a process modeled by notifying the $\mathcal{F}^{\ell}_{\mathsf{COTe}}$ functionality that the parties are ready, and implemented using the $\pi^{\mathsf{Setup}}_{\mathsf{KOS}}$ subprotocol. To sign successfully, Alice and Bob must remember the state associated with the OT-extensions and their secret keys.

Protocol 2. Two-party Setup $(\pi^{\mathsf{Setup}}_{\mathsf{2P\text{-}ECDSA}})$**:**

This protocol is parameterized by the Elliptic curve $(\mathbb{G}, G, q)$, and relies upon the $\mathcal{F}^{\ell}_{\mathsf{COTe}}$, $\mathcal{F}^{R_{\mathsf{DL}}}_{\mathsf{ZK}}$, and $\mathcal{F}^{R_{\mathsf{DL}}}_{\mathsf{Com\text{-}ZK}}$ functionalities. It takes no input and yields the joint public key pk along with a secret key share sk_A to Alice, and to Bob a secret key share sk_B along with pk.

Public Key Generation:

1) Alice and Bob sample $\mathsf{sk}_A \leftarrow \mathbb{Z}_q$ and $\mathsf{sk}_B \leftarrow \mathbb{Z}_q$, respectively.
2) Alice calculates $\mathsf{pk}_A := \mathsf{sk}_A \cdot G$ and Bob calculates $\mathsf{pk}_B := \mathsf{sk}_B \cdot G$.
3) Alice submits $(\mathsf{sk}_A, \mathsf{pk}_A)$ to the functionality $\mathcal{F}^{R_{\mathsf{DL}}}_{\mathsf{Com\text{-}ZK}}$, and Bob becomes aware of Alice's commitments.
4) Bob submits $(\mathsf{sk}_B, \mathsf{pk}_B)$ to the $\mathcal{F}^{R_{\mathsf{DL}}}_{\mathsf{ZK}}$ functionality, and Alice receives pk_B as output, along with a bit indicating that the proof was sound. If it was not, Alice aborts.
5) Alice instructs the $\mathcal{F}^{R_{\mathsf{DL}}}_{\mathsf{Com\text{-}ZK}}$ functionality to release the proof associated with her previous commitment. Bob receives pk_A as output, along with a bit indicating that

the proof was sound. If it was not, Bob aborts.

6) Alice and Bob compute the public key

$$\mathsf{pk} := \mathsf{sk}_{\mathsf{A}} \cdot \mathsf{pk}_{\mathsf{B}} = \mathsf{sk}_{\mathsf{B}} \cdot \mathsf{pk}_{\mathsf{A}}$$

Auxilliary Setup:

7) Alice and Bob both send the $(\texttt{ready})$ messages to the $\mathcal{F}^{\ell}_{\mathsf{COTe}}$ Functionality to initialize OT-extensions.

V. 2-of-n Threshold Signing

We now demonstrate a simple extension of our two-party ECDSA protocol for performing threshold signatures among n parties, with a threshold of two. In Protocol 2, Alice and Bob supplied individual secret keys $\mathsf{sk}_{\mathsf{A}}, \mathsf{sk}_{\mathsf{B}}$, which became multiplicative shares of their joint secret key. In the threshold setting we will be working with a set of parties $\mathbf{P}$ of size n, each party i with a secret key share sk_i, and we demand that if the setup does not abort then any pair of parties can sign under the joint sk.

In order to achieve this, we specify that in the threshold setting, the joint secret key sk is calculated as the *sum* of the parties' contributions, rather than as the product:

$$\mathsf{sk} := \sum_{i\in[1,n]} \mathsf{sk}_i$$

In other words, the parties' individual secret keys represent an n-of-n sharing of sk. It is natural to use a threshold secret sharing scheme to convert these into a 2-of-n sharing. Specifically, we use Shamir Secret Sharing [17], and a simple consistency check allows us to guarantee security against malicious adversaries.

From Shamir shares, any two parties can generate additive shares of the joint secret key. However, our 2-of-2 signing protocol (Protocol 1) required multiplicative shares as its input. We will need to modify the signing protocol slightly to account for the change. First, we present our 2-of-n setup procedure.

A. Setup

Protocol 3. 2-of-n Setup $(\pi^{\mathsf{2P\text{-}Setup}}_{n\mathsf{P\text{-}ECDSA}})$:

This protocol is parameterized by the Elliptic curve $(\mathbb{G}, G, q)$, and relies $\mathcal{F}^{\ell}_{\mathsf{COTe}}$ and $\mathcal{F}^{R_{\mathsf{DL}}}_{\mathsf{Com\text{-}ZK}}$ functionalities. It runs among a group of parties $\mathbf{P}$ of size n, from whom it takes no input. It yields as output for all parties a joint public key pk, and for each individual party $\mathbf{P}_i$ a point $p(i)$ on the polynomial p and a secret key share sk_i.

Public Key Generation:

1) For all $i \in [1,n]$, Party $\mathbf{P}_i$ samples $\mathsf{sk}_i \leftarrow \mathbb{Z}_q$.
2) For all $i \in [1,n]$, Party $\mathbf{P}_i$ calculates $\mathsf{pk}_i := \mathsf{sk}_i \cdot G$ and submits $(\mathsf{sk}_i, \mathsf{pk}_i)$ to the $\mathcal{F}^{R_{\mathsf{DL}}}_{\mathsf{Com\text{-}ZK}}$ functionality, which notifies all other parties that $\mathbf{P}_i$ is committed. When $\mathbf{P}_i$ becomes aware of all other parties' commitments, it instructs $\mathcal{F}^{R_{\mathsf{DL}}}_{\mathsf{Com\text{-}ZK}}$ to release its proof to the others. If any party's proof fails to verify, then all parties abort.
3) All parties compute the shared public key

$$\mathsf{pk} := \sum_{i\in[1,n]} \mathsf{pk}_i$$

4) For all $i \in [1,n]$, $\mathbf{P}_i$ chooses a random line given by the degree-1 polynomial $p_i(x)$, such that $p_i(0) = \mathsf{sk}_i$. For all $j \in [1,n]$, $\mathbf{P}_i$ sends $p_i(j)$ to $\mathbf{P}_j$ and receives $p_j(i)$.
5) For all $i \in [1,n]$, $\mathbf{P}_i$ computes its point on the joint polynomial p,

$$p(i) := \sum_{j\in[1,n]} p_j(i)$$

It also computes a commitment to its share of the secret key, $T_i := p(i) \cdot G$, and broadcasts T_i to all other parties.

6) All parties abort if $\exists i \in [2,n]$ such that

$$\lambda_{(i-1),i} \cdot T_{i-1} + \lambda_{i,(i-1)} \cdot T_i \neq \mathsf{pk}$$

where $\lambda_{(i-1),i}$ and $\lambda_{i,(i-1)}$ are the appropriate Lagrange coefficients for Shamir-reconstruction between $\mathbf{P}_{i-1}$ and $\mathbf{P}_i$. If any party holds a point $p(i)$ that is inconsistent with the polynomial held by the other parties, then this check will fail.

Auxilliary Setup:

7) Every pair of parties $\mathbf{P}_i$ and $\mathbf{P}_j$ such that $i < j$ send the $(\texttt{ready})$ message to the $\mathcal{F}^{\ell}_{\mathsf{COTe}}$ functionality to initialize OT-extensions between themselves.

A Note on General Thresholds: We note that a slight generalization of the $\pi^{\mathsf{2P\text{-}Setup}}_{n\mathsf{P\text{-}ECDSA}}$ protocol allows it to perform setup for any threshold t such that $t \leq n$. The only required changes are the use of polynomials of the appropriate degree (as in Shamir Secret Sharing), and the evaluation of the consistency check in step 6 over contiguous threshold-sized groups of parties. However, our signing protocol is not so easily generalized, and therefore we leave general threshold signing to future work.

B. Signing

Once the setup is complete, suppose two parties from the set $\mathbf{P}$ (we will resume referring to them as Alice and Bob) wish to sign. They can use Lagrange interpolation [50] to construct additive shares $t^0_{\mathsf{A}}, t^0_{\mathsf{B}}$ of the secret key, but the signing algorithm we have previously described requires *multiplicative* shares. To account for this, we modify our signing algorithm in the following intuitive way: originally, the second invocation of π_{Mul} took $\mathsf{sk}_{\mathsf{A}}/k_{\mathsf{A}}$ from Alice and $\mathsf{sk}_{\mathsf{B}}/k_{\mathsf{B}}$ from Bob and computed additive shares of the product

$$\frac{\mathsf{sk}_{\mathsf{A}} \cdot \mathsf{sk}_{\mathsf{B}}}{k_{\mathsf{A}} \cdot k_{\mathsf{B}}}$$

We replace this with two invocations of π_{Mul} that calculate

$$\frac{t^0_{\mathsf{A}}}{k_{\mathsf{A}} \cdot k_{\mathsf{B}}} \quad \text{and} \quad \frac{t^0_{\mathsf{B}}}{k_{\mathsf{A}} \cdot k_{\mathsf{B}}}$$

respectively. Alice and Bob can then locally sum their outputs from these two multiplications to yield shares of

$$\frac{t^0_{\mathsf{A}} + t^0_{\mathsf{B}}}{k_{\mathsf{A}} \cdot k_{\mathsf{B}}} = \frac{\mathsf{sk}}{k}$$

In Figure 2, we illustrate our 2-of-n signing protocol, along with the optimized triple-multiplication protocol that we describe in Section VI-D.

Protocol 4. 2-of-n Signing $\left(\pi^{\text{2P-Sign}}_{n\text{P-ECDSA}}\right)$:

This protocol is parameterized identically to Protocol 1, except that Alice and Bob provide Shamir-shares $p(\mathsf{A}), p(\mathsf{B})$ of sk as input, rather than multiplicative shares.

Key Share Reconstruction:

1) Alice locally calculates the correct Lagrange coefficient $\lambda_{\mathsf{A,B}}$ for Shamir-reconstruction with Bob. Bob likewise calculates $\lambda_{\mathsf{B,A}}$. They then use their respective points $p(\mathsf{A}), p(\mathsf{B})$ on the polynomial p to calculate additive shares of the secret key

$$t^0_{\mathsf{A}} := \lambda_{\mathsf{A,B}} \cdot p(\mathsf{A}) \qquad t^0_{\mathsf{B}} := \lambda_{\mathsf{B,A}} \cdot p(\mathsf{B})$$

Multiplication and Instance Key Exchange:

2) Bob chooses his secret instance key, $k_{\mathsf{B}} \leftarrow \mathbb{Z}_q$, and Alice chooses her instance key seed, $k'_{\mathsf{A}} \leftarrow \mathbb{Z}_q$. Bob computes

$$D_{\mathsf{B}} := k_{\mathsf{B}} \cdot G$$

and sends D_{B} to Alice.

3) Alice computes

$$\begin{aligned} R' &:= k'_{\mathsf{A}} \cdot D_{\mathsf{B}} \\ k_{\mathsf{A}} &:= H(R') + k'_{\mathsf{A}} \\ R &:= k_{\mathsf{A}} \cdot D_{\mathsf{B}} \end{aligned}$$

4) Alice chooses a pad $\phi \leftarrow \mathbb{Z}_q$, and then Alice and Bob run the π_{Mul} subprotocol with inputs $\phi + 1/k_{\mathsf{A}}$ and $1/k_{\mathsf{B}}$ respectively, and receive shares t^1_{A} and t^1_{B} of their padded joint inverse instance key.

5) Alice and Bob run the π_{Mul} subprotocol with inputs $t^0_{\mathsf{A}}/k_{\mathsf{A}}$ and $1/k_{\mathsf{B}}$ respectively. They receive shares $t^{2\text{a}}_{\mathsf{A}}, t^{2\text{a}}_{\mathsf{B}}$ of Alice's secret key share over their joint instance key

$$t^{2\text{a}}_{\mathsf{A}} + t^{2\text{a}}_{\mathsf{B}} = \frac{t^0_{\mathsf{A}}}{k_{\mathsf{A}} \cdot k_{\mathsf{B}}}$$

6) Alice and Bob run the π_{Mul} subprotocol with inputs $1/k_{\mathsf{A}}$ and $t^0_{\mathsf{B}}/k_{\mathsf{B}}$ respectively. They receive shares $t^{2\text{b}}_{\mathsf{A}}, t^{2\text{b}}_{\mathsf{B}}$ of Bob's secret key share over their joint instance key

$$t^{2\text{b}}_{\mathsf{A}} + t^{2\text{b}}_{\mathsf{B}} = \frac{t^0_{\mathsf{B}}}{k_{\mathsf{A}} \cdot k_{\mathsf{B}}}$$

7) Alice and Bob merge their respective shares

$$t^2_{\mathsf{A}} := t^{2\text{a}}_{\mathsf{A}} + t^{2\text{b}}_{\mathsf{A}} \qquad t^2_{\mathsf{B}} := t^{2\text{a}}_{\mathsf{B}} + t^{2\text{b}}_{\mathsf{B}}$$

8) Alice transmits R' to Bob, who computes

$$R := H(R') \cdot D_{\mathsf{B}} + R'$$

For both Alice and Bob let $(r_x, r_y) = R$.

Consistency Check, Signature, and Verification:

As in Protocol 1 $\left(\pi^{\text{Sign}}_{\text{2P-ECDSA}}\right)$

VI. Multiplication with OT Extensions

The Bulk of both the complexity and the practical cost of our scheme arises from the OT-extension protocols which we use to perform multiplication. We augment Simplest OT [33] with a verification procedure and refer to the new primitive as Verified Simplest OT (VSOT). VSOT is used as the basis for a lightly optimized instantiation of the KOS [34] OT-extension protocol, which is used in turn to build the OT-multiplication primitive required by our main signing protocol.

If we did not desire simulation-based malicious security, then it would be sufficient to use the Simplest OT scheme without modification. In composing the protocol to build a larger simulation-sound malicious protocol however, there is a complication. The security proof relies upon the fact that the protocol's hash queries are modeled as calls to a Random Oracle, and uses those queries to extract the receiver's inputs. However, the queries need not occur before the receiver has sent its last message, and so there is no guarantee that a malicious receiver will actually query the oracle. When Simplest OT is composed, it may be the case that the receiver's inputs are required for simulation before they are required by the receiver itself, in which case the protocol will be unsimulatable. This flaw has recently been noticed by a number of authors, including Byali *et al.* [51], who discuss it in more detail, and it seems to affect other OT protocols as well [35], [52]. Barreto *et al.* [52] propose to solve the problem by adding a public-key verification process in the Random Oracle model. Rather than using expensive public-key operations, however, we specify that the receiver must prove knowledge of its output using only symmetric-key operations, ensuring that it does in fact hold that output, and therefore that its input is extractable. As a consequence, our protocol is able to realize only an OT functionality ($\mathcal{F}_{\mathsf{SF\text{-}OT}}$) that allows for selective failure by the sender, but we show that this is sufficient for our purposes.

A. *Verified Simplest OT*

We begin by describing the VSOT protocol. Because Alice and Bob participate in this protocol with their roles reversed, relative to the usual arrangement, we refer to the participants simply as the sender and receiver in this section. The protocol comprises four phases. In the first, the sender generates a private/public key pair, and sends the public key to the receiver. In the second phase, the receiver encodes its choice bit and the sender generates two random pads based upon the encoded choice bit in such a way that the receiver can only recover one. The third phase is a verification, which is necessary to ensure that the protocol is simulatable. Finally, the pads are used by the sender to mask its messages for transmission to the receiver in the fourth phase. This protocol realizes the $\mathcal{F}_{\mathsf{SF\text{-}OT}}$ functionality, which is given as Functionality 3 in Appendix A.

Protocol 5. Verified Simplest OT (π_{VSOT}):

This protocol is parameterized by the Elliptic curve $(\mathbb{G}, G, q)$, and symmetric security parameter $\kappa = |q|$, and a hash function $H : \{0,1\}^* \mapsto \mathbb{Z}_q$. It relies upon the $\mathcal{F}^{R_{\mathsf{DL}}}_{\mathsf{ZK}}$ functionality. It takes as input a choice bit $\omega \in \{0,1\}$ from the receiver, and two messages $\alpha^0, \alpha^1 \in \mathbb{Z}_q$ from the sender. It outputs one message $\alpha^\omega \in \mathbb{Z}_q$ to the receiver, and nothing to the sender.

Public Key:

1) The sender samples $b \leftarrow \mathbb{Z}_q$ and computes $B := b \cdot G$.

2) The sender submits (B, b) to the $\mathcal{F}_{\mathsf{ZK}}^{R_{\mathsf{DL}}}$ functionality, and the receiver receives B along with a bit indicating whether the proof was sound. If it was not, the receiver aborts.

Pad Transfer:

3) The receiver samples $a \leftarrow \mathbb{Z}_q$, and then computes its encoded choice bit A and the pad ρ^ω

$$A \coloneqq a \cdot G + \omega \cdot B$$
$$\rho^\omega \coloneqq H(a \cdot B)$$

and sends A to the sender.

4) The sender computes two pads

$$\rho^0 \coloneqq H(b \cdot A)$$
$$\rho^1 \coloneqq H(b \cdot (A - B))$$

Verification:

5) The sender computes a challenge

$$\xi \coloneqq H(H(\rho^0)) \oplus H(H(\rho^1))$$

and sends the challenge ξ to the receiver.

6) The receiver computes a response

$$\rho' \coloneqq H(H(\rho^\omega)) \oplus (\omega \cdot \xi)$$

and sends ρ' to the sender.

7) The sender aborts if $\rho' \neq H(H(\rho^0))$. Otherwise, it opens its challenge by sending $H(\rho^0)$ and $H(\rho^1)$ to the receiver.

8) The receiver aborts if the value of $H(\rho^\omega)$ it received from the sender does not match the one it calculated itself, or if

$$\xi \neq H(H(\rho^0)) \oplus H(H(\rho^1))$$

Message Transfer:

9) The sender pads its two messages α^0, α^1, and transmits the padded messages $\hat{\alpha}^0, \hat{\alpha}^1$ to the receiver

$$\hat{\alpha}^0 \coloneqq \alpha^0 + \rho^0$$
$$\hat{\alpha}^1 \coloneqq \alpha^1 + \rho^1$$

10) The receiver removes the pad from its chosen message

$$\alpha^\omega = \hat{\alpha}^\omega - \rho^\omega$$

For simplicity, we describe VSOT as requiring one complete protocol evaluation per OT instance. However, if (public) nonces are used in each of the hash invocations, then the Public Key phase can be run once and the resulting (single) public key B can be reused in as many Transfer and Verification phases as required without sacrificing security. Further note that if the messages transmitted by the sender are specified to be uniform, then the sender can actually omit the Message Transfer phase entirely and treat the pads ρ^0, ρ^1 as messages, receiving them as output instead of supplying them as input. Likewise, the receiver treats its one pad ρ^ω as its output. This effectively transforms VSOT into a Random OT protocol. We make use of both of these optimizations in our implementation.

B. Correlated OT-extension with KOS

Our multiplication protocol requires the use of a large number of OT instances where the correlation between messages is specified, but the messages must otherwise be random. Therefore, rather than using VSOT directly, we layer a Correlated OT-extension (COTe) protocol atop it. This is essentially an instantiation the KOS protocol; thus we include a protocol description here for completeness, but refer the reader to Keller *et al.* [34] for a more thorough discussion. Being a Correlated OT protocol, it allows the sender to define a correlation between the two messages, but does not allow the sender to determine the messages specifically. As with all OT-extension systems, it is divided into a setup protocol, which uses some base OT system to generate correlated secrets between the two parties, and an extension protocol, which uses these correlated secrets to efficiently perform additional OTs. These protocols realize the Correlated Oblivious Transfer functionality $\mathcal{F}_{\mathsf{COTe}}^{\ell}$, which is given as Functionality 4 in Appendix A.

Protocol 6. KOS Setup $(\pi_{\mathsf{KOS}}^{\mathsf{Setup}})$:

This protocol is parameterized by the curve order q and the symmetric security parameter $\kappa = |q|$. It depends upon the OT Functionality $\mathcal{F}_{\mathsf{SF\text{-}OT}}$, and takes no input from either party. Alice receives as output a private OTe correlation $\boldsymbol{\nabla} \in \{0,1\}^\kappa$ and a vectors of seeds $\mathbf{s}^{\boldsymbol{\nabla}} \in \mathbb{Z}_q^\kappa$, and Bob receives two vectors of seeds $\mathbf{s}^0$ and $\mathbf{s}^1 \in \mathbb{Z}_q^\kappa$.

Setup:

1) Alice samples a correlation vector, $\boldsymbol{\nabla} \leftarrow \{0,1\}^\kappa$.
2) For each bit $\boldsymbol{\nabla}_i$ of the correlation vector, Alice and Bob access the $\mathcal{F}_{\mathsf{SF\text{-}OT}}$ functionality, with Alice acting as the receiver and using $\boldsymbol{\nabla}_i$ for her choice bit and Bob acting as the sender. Bob samples two random seed elements $\mathbf{s}_i^0 \leftarrow \mathbb{Z}_q$ and $\mathbf{s}_i^1 \leftarrow \mathbb{Z}_q$ and Alice receives as output a single seed element $\mathbf{s}_i^{\boldsymbol{\nabla}_i}$.
3) Alice and Bob collate their individual seed elements into vectors, $\mathbf{s}^{\boldsymbol{\nabla}}$ and $\mathbf{s}^0, \mathbf{s}^1$ respectively, and take these vectors as output.

Protocol 7. KOS Extension $(\pi_{\mathsf{KOS}}^{\mathsf{Extend}})$:

This protocol is parameterized by the OT batch size ℓ, the OT security parameter κ^{OT}, the curve order q, and the symmetric security parameter $\kappa = |q|$. For notational convenience, let $\ell' = \ell + \kappa^{\mathsf{OT}}$. It makes use of the pseudo-random generator $\mathsf{Prg}_{\mathbb{Z}} : \mathbb{Z}_q^\kappa \mapsto \mathbb{Z}_{2^{\ell'}}$, which expands its argument and then outputs the chunk of ℓ' bits indexed by the value given as a subscript, and it makes use use of the hash function $H : \{0,1\}^* \mapsto \mathbb{Z}_q$. The protocol also uses a fresh, public OT-extension index, $\mathsf{ext_{id}}$. Alice supplies a vector of input integers, $\boldsymbol{\alpha} \in \mathbb{Z}_q^\ell$, along with her private OTe correlation $\boldsymbol{\nabla} \in \{0,1\}^\kappa$ and seed $\mathbf{s}^{\boldsymbol{\nabla}} \in \mathbb{Z}_q^\kappa$, which she received during the KOS setup protocol. Bob supplies a vector of choice bits $\boldsymbol{\omega} \in \{0,1\}^\ell$ along with his seeds $\mathbf{s}^0$ and $\mathbf{s}^1 \in \mathbb{Z}_q^\kappa$ from the OT setup. Alice and Bob receive $\mathbf{t}_{\mathsf{A}}$ and $\mathbf{t}_{\mathsf{B}} \in \mathbb{Z}_q^\ell$ as output.

Extension:

1) Bob chooses $\boldsymbol{\gamma}^{\mathsf{ext}} \leftarrow \{0,1\}^{\kappa^{\mathsf{OT}}}$ and collates $\mathbf{w} := \boldsymbol{\omega} \| \boldsymbol{\gamma}^{\mathsf{ext}}$. We use w to indicate $\mathbf{w}$ interpreted as a single value in $\mathbb{Z}_{2^{\ell'}}$. That is, $\mathsf{Bits}(w) = \mathbf{w}$.
2) Bob computes two vectors of PRG expansions of his OT-extension seeds
$$\mathbf{v}^0 := \left\{\mathsf{Prg}_{\mathsf{ext}_{\mathsf{id}}}(\mathbf{s}_i^0)\right\}_{i\in[1,\kappa]}$$
$$\mathbf{v}^1 := \left\{\mathsf{Prg}_{\mathsf{ext}_{\mathsf{id}}}(\mathbf{s}_i^1)\right\}_{i\in[1,\kappa]}$$
and Alice computes a vector of expansions of her correlated seed
$$\mathbf{v}^{\boldsymbol{\nabla}} := \left\{\mathsf{Prg}_{\mathsf{ext}_{\mathsf{id}}}(\mathbf{s}_i^{\boldsymbol{\nabla}_i})\right\}_{i\in[1,\kappa]}$$
3) Bob collates the vector $\boldsymbol{\psi} \in \mathbb{Z}_q^{\ell'}$, which is the transpose of $\mathbf{v}^0$. That is, the first element of $\boldsymbol{\psi}$ is the concatenation of the first bits of all of the elements of $\mathbf{v}^0$, and so on. More formally if we define a matrix
$$\mathbf{V} \in \{0,1\}^{\kappa\times\ell'}$$
then the relationship is given by
$$\mathbf{V}^i = \mathsf{Bits}(\mathbf{v}_i^0) \quad \forall i \in [1,\kappa]$$
$$\mathbf{V}_j = \mathsf{Bits}(\boldsymbol{\psi}_j) \quad \forall j \in [1,\ell']$$
4) Bob computes the matrix
$$\mathbf{u} := \left\{\mathbf{v}_i^0 \oplus \mathbf{v}_i^1 \oplus w\right\}_{i\in[1,\kappa]}$$
and then he computes a matrix of pseudo-random elements from $\mathbb{Z}_q$
$$\boldsymbol{\chi} := \{H(j\|\mathbf{u})\}_{j\in[1,\ell']}$$
which he uses to create a linear sampling of $\mathbf{w}$ and $\boldsymbol{\psi}$
$$w' := \bigoplus_{j\in[1,\ell']} \mathbf{w}_j \cdot \boldsymbol{\chi}_j$$
$$v' := \bigoplus_{j\in[1,\ell']} \boldsymbol{\psi}_j \wedge \boldsymbol{\chi}_j$$
Finally, he sends w', v', and $\mathbf{u}$ to Alice.
5) Alice computes the vector
$$\mathbf{z} := \left\{\mathbf{v}_i^{\boldsymbol{\nabla}_i} \oplus (\boldsymbol{\nabla}_i \cdot \mathbf{u}_i)\right\}_{i\in[1,\kappa]}$$
and collates the vector $\boldsymbol{\zeta}$, which is the transpose of $\mathbf{z}$ in exactly the way that $\boldsymbol{\psi}$ is the transpose $\mathbf{v}^0$. She also calculates $\boldsymbol{\chi}$ in the same manner as Bob
$$\boldsymbol{\chi} := \{H(j\|\mathbf{u})\}_{j\in[1,\ell']}$$
Finally, she computes
$$z' := \bigoplus_{j\in[1,\ell']} \boldsymbol{\zeta}_j \wedge \boldsymbol{\chi}_j$$
and if $z' \neq v' \oplus (\nabla \wedge w')$, where ∇ is $\boldsymbol{\nabla}$ reinterpreted as an element in $\mathbb{Z}_{2^\kappa}$, then Alice aborts.

Transfer:

6) Alice computes
$$\mathbf{t}_{\mathsf{A}} := \left\{H(j\|\boldsymbol{\zeta}_j)\right\}_{j\in[1,\ell]}$$
$$\boldsymbol{\tau} := \left\{H(j\|(\boldsymbol{\zeta}_j \oplus \nabla)) - \mathbf{t}_{\mathsf{A}j} + \boldsymbol{\alpha}_j\right\}_{j\in[1,\ell]}$$
and sends $\boldsymbol{\tau}$ to Bob
7) Bob computes
$$\mathbf{t}_{\mathsf{B}} := \left\{\begin{cases} -H(j\|\boldsymbol{\psi}_j) & \text{if } \mathbf{w}_j = 0 \\ \boldsymbol{\tau}_j - H(j\|\boldsymbol{\psi}_j) & \text{if } \mathbf{w}_j = 1 \end{cases}\right\}_{j\in[1,\ell]}$$

C. Multiplication

In the context of our scheme, we are primarily interested in using OT-extension as a basis for two-party multiplication. The classic Gilboa [32] OT-multiplication takes an input from Alice and an input from Bob, and returns to them additive secret shares of the product of those two inputs. It works essentially by performing binary multiplication with a single oblivious transfer for each bit in Bob's input.

Unfortunately, this protocol is vulnerable to selective failure attacks in the malicious setting. Alice can corrupt one of the two messages during any single transfer, and in doing so learn the value of Bob's input bit for that transfer according to whether or not their outputs are correct. We address this by encoding Bob's input with enough redundancy that learning s (a statistical security parameter) of Bob's choice bits via selective failure does not leak information about the original input value. A consistency check ensures that the parties abort if the multiplication output is incorrect, and thus the probability that Alice succeeds in more than s selective failures is exponentially small. A proposition of Impagliazzo and Naor [53] gives us the following encoding scheme: for an input β of length κ, sample $\kappa + 2s$ random bits $\boldsymbol{\gamma}^{\mathsf{mul}} \leftarrow \{0,1\}^{\kappa+2s}$ and take the dot product with some public random vector $\mathbf{c}_{\mathsf{R}} \in \mathbb{Z}_q^{\kappa+2s}$. Use this dot product as a mask for the original input. The encoded input is
$$\mathsf{Bits}\left(\beta - \langle \mathbf{c}_{\mathsf{R}}, \boldsymbol{\gamma}^{\mathsf{mul}}\rangle\right) \| \boldsymbol{\gamma}^{\mathsf{mul}}$$
In the full version of this paper, we prove formally that this encoding scheme is secure against s selective failures.

Protocol 8. Multiplication (π_{Mul})**:**

This protocol is parameterized by the statistical security parameter s, the curve order q, and the symmetric security parameter $\kappa = |q|$. It also makes use of a coefficient vector $\mathbf{c} = \mathbf{c}_{\mathsf{G}} \| \mathbf{c}_{\mathsf{R}}$, where $\mathbf{c}_{\mathsf{G}} \in \mathbb{Z}_q^{\kappa}$ is a *gadget vector* such that $\mathbf{c}_{\mathsf{G}i} = 2^{i-1}$, and $\mathbf{c}_{\mathsf{R}} \leftarrow \mathbb{Z}_q^{\kappa+2s}$ is a public random vector. It requires access to the Correlated Oblivious Transfer functionality $\mathcal{F}^{\ell}_{\mathsf{COTe}}$. Alice supplies some input integer $\alpha \in \mathbb{Z}_q$, and Bob supplies some input integer $\beta \in \mathbb{Z}_q$. Alice and Bob receive t_{A} and $t_{\mathsf{B}} \in \mathbb{Z}_q$ as output, respectively, such that

$t_{\mathsf{A}} + t_{\mathsf{B}} = \alpha \cdot \beta$.

Encoding:

1) Bob chooses $\boldsymbol{\gamma}^{\mathsf{mul}} \leftarrow \{0,1\}^{\kappa+2s}$ and computes

$$\boldsymbol{\omega} := \mathsf{Bits}\left(\beta - \langle \mathbf{c}_{\mathsf{R}}, \boldsymbol{\gamma}^{\mathsf{mul}} \rangle\right) \| \boldsymbol{\gamma}^{\mathsf{mul}}$$

This is essentially a randomized encoding of β.

2) Alice sets

$$\boldsymbol{\alpha} := \{\mathbf{c}_j \cdot \alpha\}_{j \in [1, 2\kappa+2s]}$$

Multiplication:

3) Alice and Bob access the $\mathcal{F}^{\ell}_{\mathsf{COTe}}$ functionality, with $\ell := 2\kappa + 2s$. Alice plays the sender, supplying $\boldsymbol{\alpha}$ as her input, and Bob, the receiver, supplies $\boldsymbol{\omega}$. They receive $\mathbf{t}_{\mathsf{A}}$ and $\mathbf{t}_{\mathsf{B}}$ as outputs, respectively.

4) Alice and Bob compute their output shares

$$t_{\mathsf{A}} := \sum_{j \in [1, 2\kappa+2s]} \mathbf{t}_{\mathsf{A}j} \qquad t_{\mathsf{B}} := \sum_{j \in [1, 2\kappa+2s]} \mathbf{t}_{\mathsf{B}j}$$

D. Coalesced Multiplication

The multiplication protocol described in the foregoing section supports the multiplication of only a single integer α by a single integer β, and in our two-party and 2-of-n signing protocols (Protocols 1 and 4 respectively) we invoke the multiplication protocol two or three times. An optimization allows these multiple invocations to be combined at reduced cost, albeit by breaking some of our previous abstractions.

Consider first the case of two-party signing, wherein two multiplications must be performed. Each multiplication individually encodes its input, enlarging it by $\kappa + 2s$ bits to account for the encoding vector $\boldsymbol{\gamma}^{\mathsf{mul}}$, and then individually calls upon the $\mathcal{F}^{\ell}_{\mathsf{COTe}}$ correlated OT-extension functionality with batch size $\ell = 2\kappa + 2s$. The $\pi^{\mathsf{Extend}}_{\mathsf{KOS}}$ protocol that realizes this functionality comprises an Extension phase and a Transfer phase. In the latter, both computation and communication costs are proportionate to ℓ, but in the former, they are proportionate to $\ell' = \ell + \kappa^{\mathsf{OT}}$. Two multiplications performed in the naïve way incur twice the cost. However, we observe that two multiplication protocol instances can share a single invocation of $\pi^{\mathsf{Extend}}_{\mathsf{KOS}}$ simply by doubling the batch size, thereby reducing the extension cost by an amount proportionate to κ^{OT}. Furthermore, we show in the full version of this paper that our encoding scheme requires only $2\kappa + 2s$ random bits to encode two inputs of length κ when the inputs are combined into a single extension instance, rather than $2\kappa + 4s$ bits, as would be required if the inputs were encoded separately. Thus, we can construct an improved double-multiplication protocol as follows.

Suppose that Alice and Bob wish to compute the products $\alpha^1 \cdot \beta^1$ and $\alpha^2 \cdot \beta^2$ where the inputs are all of length κ. Bob chooses the encoding vectors $\boldsymbol{\gamma}^{\mathsf{mul1}}, \boldsymbol{\gamma}^{\mathsf{mul2}} \leftarrow \{0,1\}^{\kappa}$, $\boldsymbol{\gamma}^{\mathsf{mul3}} \leftarrow \{0,1\}^{2s}$ and computes a single choice bit vector

$$\begin{aligned}\boldsymbol{\omega} := \mathsf{Bits}&\left(\beta^1 - \langle \mathbf{c}_{\mathsf{R}}, \boldsymbol{\gamma}^{\mathsf{mul1}} \| \boldsymbol{\gamma}^{\mathsf{mul3}} \rangle\right) \| \boldsymbol{\gamma}^{\mathsf{mul1}} \\ \| \; \mathsf{Bits}&\left(\beta^2 - \langle \mathbf{c}_{\mathsf{R}}, \boldsymbol{\gamma}^{\mathsf{mul2}} \| \boldsymbol{\gamma}^{\mathsf{mul3}} \rangle\right) \| \boldsymbol{\gamma}^{\mathsf{mul2}} \| \boldsymbol{\gamma}^{\mathsf{mul3}}\end{aligned}$$

For her part, Alice calculates $\boldsymbol{\alpha}^1$ from α^1 and $\boldsymbol{\alpha}^2$ from α^2 using the ordinary coefficient vector $\mathbf{c}$. Alice and Bob then engage in the Extension phase of the $\pi^{\mathsf{Extend}}_{\mathsf{KOS}}$ protocol with $\ell = 4\kappa + 2s$, which produces $\mathbf{w} \in \{0,1\}^{\ell}$ and $\boldsymbol{\psi} \in \mathbb{Z}_q^{\ell}$ as output for Bob, and $\boldsymbol{\zeta} \in \mathbb{Z}_q^{\ell}$ as output for Alice. They then engage in a modified version of the $\pi^{\mathsf{Extend}}_{\mathsf{KOS}}$ Transfer phase. Specifically, when hashing the parts of $\boldsymbol{\zeta}$ and $\boldsymbol{\psi}$ that correspond to the encoding vector $\boldsymbol{\gamma}^{\mathsf{mul3}}$, Alice and Bob both use hash functions of the form $H^2 : \{0,1\}^* \mapsto \mathbb{Z}_q^2$, which produce two elements from $\mathbb{Z}_q$ as output rather than the usual one element. If we use $H_1^2(\cdot)$ and $H_2^2(\cdot)$ to indicate the first and second elements produced by a particular hash function invocation, then Alice computes her output and transfer vectors as

$$\begin{aligned}\mathbf{t}_{\mathsf{A}} := &\left\{H(j \| \boldsymbol{\zeta}_j)\right\}_{j \in [1, 4\kappa]} \\ &\| \left\{H_1^2(j \| \boldsymbol{\zeta}_j)\right\}_{j \in (4\kappa, 4\kappa+2s]} \\ &\| \left\{H_2^2(j \| \boldsymbol{\zeta}_j)\right\}_{j \in (4\kappa, 4\kappa+2s]} \\ \boldsymbol{\tau} := &\left\{H(j \| (\boldsymbol{\zeta}_j \oplus \nabla)) - H(j \| \boldsymbol{\zeta}_j) + \boldsymbol{\alpha}^1_j\right\}_{j \in [1, 2\kappa]} \\ &\| \left\{H(j \| (\boldsymbol{\zeta}_j \oplus \nabla)) - H(j \| \boldsymbol{\zeta}_j) + \boldsymbol{\alpha}^2_{j-2\kappa}\right\}_{j \in (2\kappa, 4\kappa]} \\ &\| \left\{H_1^2(j \| (\boldsymbol{\zeta}_j \oplus \nabla)) - H_1^2(j \| \boldsymbol{\zeta}_j) + \boldsymbol{\alpha}^1_{j-2\kappa}\right\}_{j \in (4\kappa, 4\kappa+2s]} \\ &\| \left\{H_2^2(j \| (\boldsymbol{\zeta}_j \oplus \nabla)) - H_2^2(j \| \boldsymbol{\zeta}_j) + \boldsymbol{\alpha}^2_{j-2\kappa}\right\}_{j \in (4\kappa, 4\kappa+2s]}\end{aligned}$$

Notice that when calculating $\boldsymbol{\tau}$, Alice masks $\boldsymbol{\alpha}^1$ with the lower halves of the outputs of H^2, and $\boldsymbol{\alpha}^2$ with the upper. Bob computes his output vector

$$\begin{aligned}\mathbf{t}_{\mathsf{B}} := &\left\{\begin{cases} -H(j \| \boldsymbol{\psi}_j) & \text{if } \mathbf{w}_j = 0 \\ \boldsymbol{\tau}_j - H(j \| \boldsymbol{\psi}_j) & \text{if } \mathbf{w}_j = 1 \end{cases}\right\}_{j \in [1, 4\kappa]} \\ &\| \left\{\begin{cases} -H_1^2(j \| \boldsymbol{\psi}_j) & \text{if } \mathbf{w}_j = 0 \\ \boldsymbol{\tau}_j - H_1^2(j \| \boldsymbol{\psi}_j) & \text{if } \mathbf{w}_j = 1 \end{cases}\right\}_{j \in (4\kappa, 4\kappa+2s]} \\ &\| \left\{\begin{cases} -H_2^2(j \| \boldsymbol{\psi}_j) & \text{if } \mathbf{w}_j = 0 \\ \boldsymbol{\tau}_{j+2s} - H_2^2(j \| \boldsymbol{\psi}_j) & \text{if } \mathbf{w}_j = 1 \end{cases}\right\}_{j \in (4\kappa, 4\kappa+2s]}\end{aligned}$$

Finally, Alice and Bob compute their output shares

$$t^1_{\mathsf{A}} := \sum_{\substack{j \in [1, 2\kappa] \\ \cup (4\kappa, 4\kappa+2s]}} \mathbf{t}_{\mathsf{A}j} \qquad t^1_{\mathsf{B}} := \sum_{\substack{j \in [1, 2\kappa] \\ \cup (4\kappa, 4\kappa+2s]}} \mathbf{t}_{\mathsf{B}j}$$

$$t^2_{\mathsf{A}} := \sum_{\substack{j \in (2\kappa, 4\kappa] \\ \cup (4\kappa+2s, 4\kappa+4s]}} \mathbf{t}_{\mathsf{A}j} \qquad t^2_{\mathsf{B}} := \sum_{\substack{j \in (2\kappa, 4\kappa] \\ \cup (4\kappa+2s, 4\kappa+4s]}} \mathbf{t}_{\mathsf{B}j}$$

Now Alice and Bob have shares of both products. Because they have achieved this only by extending the output lengths of certain hash function instances, the security of this double-multiplication protocol follows from the security of the original.

Further consider the case of 2-of-n signing, in which three multiplications are used to compute the products

$$\alpha_1 \cdot \beta_1 \qquad \alpha_2 \cdot \beta_2 \qquad \alpha_3 \cdot \beta_1$$

Notice that in the first and third multiplications, Bob's inputs are identical, while in the second it differs. We can compute the first and second products using the double-multiplication technique described previously, and make an additional modification in order to compute the third. Rather further enlarging the size of the OT-extension batch generated in the Extension phase of $\pi^{\mathsf{Extend}}_{\mathsf{KOS}}$, we can perform the Extension phase in exactly the same way as before, and modify only the Transfer phase. We define $H^3 : \{0,1\}^* \mapsto \mathbb{Z}_q^3$, which produces three elements from $\mathbb{Z}_q$. We use H^2 to compute the components of $\mathbf{t}_{\mathsf{A}}, \boldsymbol{\tau}$, and $\mathbf{t}_{\mathsf{B}}$ that correspond to the encoding of Bob's first input, and we use H^3 to compute the components that correspond to $\boldsymbol{\gamma}^{\mathsf{mul3}}$. Alice calculates and additional OT input vector $\boldsymbol{\alpha}^3$, and masks its elements using the additional hash outputs. The two parties then sum the additional entries in their $\mathbf{t}_{\mathsf{A}}$ and $\mathbf{t}_{\mathsf{B}}$ vectors to find shares of the third product, $\alpha_3 \cdot \beta_1$. Thus Alice and Bob can thus perform this additional multiplication simply by enlarging the hash outputs in the KOS transfer phase. In Figure 2, we illustrate this triple-multiplication technique in the context of the 2-of-n signing protocol.

To compute three products in the naïve way, $\kappa \cdot (3\kappa^{\mathsf{OT}} + 12\kappa + 12s + 6)$ bits must be transferred (with a proportionate amount of computation being performed). Concretely, if we use $\kappa = 256$, $s = 80$, and $\kappa^{\mathsf{OT}} = 128{+}s$ (following KOS [34]), then the total comes to 145.7 KiB. Using coalesced multiplication, only $\kappa \cdot (\kappa^{\mathsf{OT}} + 10\kappa + 8s + 2)$ bits must be transferred (again, with a proportionate amount of computation). Concretely, this amounts to 106.6 KiB, a savings of roughly one third.

VII. Cost Analysis

When all of the optimizations have been applied and all functionalities and sub-protocols have been collapsed, we find that our protocols have communication and computation costs as reported in Table I. Though we account completely for communications, we count only elliptic curve point multiplications and calls to the hash function H toward computation cost. We assume that both commitments and the PRG are implemented via the hash function H, and that proofs-of-knowledge-of-discrete-logarithm are implemented via Schnorr protocols with the Fiat-Shamir heuristic.

The 2-of-n setup protocol is somewhat more complex than Table I indicates. Over its course, each of the n parties commits to and then sends a single proof-of-knowledge-of-discrete-logarithm to all other parties in broadcast and then verifies the $n-1$ proofs that it receives. The parties then compute and send Lagrange coefficients to one another, which requires $O(n^2)$ (parallel) communication in total, and this pattern repeats for verification. Finally, each party evaluates a single KOS Setup instance with every other party, for $(n^2 - n)/2$ instances in total. The entire protocol requires four broadcast rounds, plus the messages required by the KOS Setup instances.

For ease of comparison, concrete communication costs for our signing protocol along with the signing protocols of Gennaro *et al.* [3], Boneh *et al.* [4], and Lindell [2] are listed in Table II. The former pair of schemes are related: Boneh *et al.* reduce the number of messages in Gennaro *et al.*'s signing protocol from six to four, with the goal of reducing the communication cost. Apart from requiring only two messages, our signing protocol requires roughly one twentieth of the communication incurred by either.

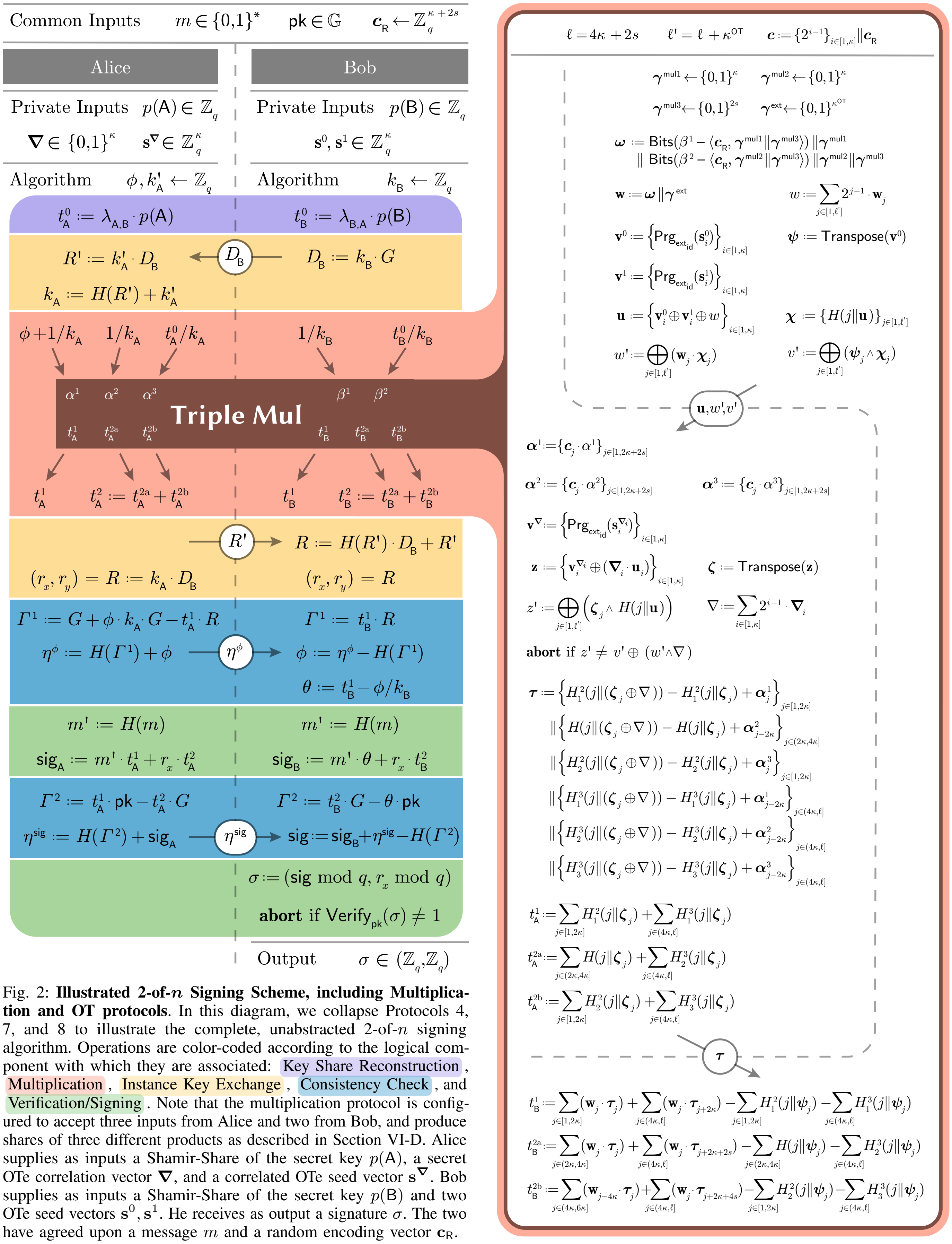
Lindell's signing scheme requires four messages and excels in terms of communication cost, only transferring a commitment, two curve points, two zero-knowledge proofs, and one Paillier ciphertext. However, the Paillier homomorphic operations it requires are quite expensive. Lindell's scheme requires one encryption, one homomorphic scalar multiplication, and one homomorphic addition with a Paillier modulus $N > 2q^4 + q^3$; concretely, a standard 2048-bit modulus is sufficient for a 256-bit curve. Gennaro *et al.* and Boneh *et al.*'s schemes both require one to three encryptions and three to five homomorphic additions and scalar multiplications per party, with $N > q^8$, which likewise implies that for 256-bit curves, a 2048-bit modulus is sufficient. In addition, Lindell's protocol requires 12 Elliptic Curve multiplications, while the protocols of the other two require roughly 100. These Paillier and group operations dominate the computation cost of the protocols.

VIII. Implementation

We created a proof-of-concept implementation of our 2-of-2 and 2-of-n setup and signing protocols in the Rust language. As a prerequisite, we also created an elliptic curve library in Rust. We use SHA-256 to instantiate the Hash function, per the ECDSA specification, and in addition we use it to instantiate the PRG. As a result, our protocol relies on both the same *theoretical* assumptions as ECDSA and the same *practical* assumption: that SHA-256 is secure. The SHA-256 implementation used in signing is capable of parallelizing vectors of hash operations, and the 2-of-n setup protocol is capable of parallelizing OT-extension initializations, but otherwise the code is strictly single-threaded. This approach has likely resulted in reduced performance relative to an optimized C implementation, but we believe that the safety afforded by Rust makes the trade worthwhile.

We benchmarked our implementation on a pair of Amazon `C5.2xlarge` instances from Amazon's Virginia datacenter, both running Ubuntu 16.04 with Linux kernel 4.4.0, and we compiled our code using Rust 1.25 with the default level of optimization. The bandwidth between our instances was measured to be be 5GBits/Second, and the round-trip latency to be 0.1ms. Our signatures were calculated over the secp256k1 curve, as standardized by NIST [7]. Thus $\kappa = 256$, and we chose $s = 80$ and $\kappa^{\mathsf{OT}} = 128 + s$, following the analysis of KOS [34]. We performed both strictly single-threaded benchmarks, and benchmarks allowing parallel hashing with three threads per party, collecting 10,000 samples for setup and 100,000 for signing. Note that signatures were not batched, and thus each sample was impacted individually by the full latency of the network. The average wall-clock times for both signing protocols and the 2-of-2 setup protocol are reported in Table III, along with results from previous works for comparison.

We benchmarked our 2-of-n setup algorithm using set of 20 Amazon `C5.2xlarge` instances from the Virginia datacenter,



Common Inputs $m \in \{0,1\}^*$ $\mathsf{pk} \in \mathbb{G}$ $\boldsymbol{c}_{\mathsf{R}} \leftarrow \mathbb{Z}_q^{\kappa+2s}$
Alice
Bob
Private Inputs $p(\mathsf{A}) \in \mathbb{Z}_q$
$\boldsymbol{\nabla} \in \{0,1\}^\kappa$ $\mathbf{s}^{\nabla} \in \mathbb{Z}_q^\kappa$
Private Inputs $p(\mathsf{B}) \in \mathbb{Z}_q$
$\mathbf{s}^0, \mathbf{s}^1 \in \mathbb{Z}_q^\kappa$
Algorithm $\phi, k'_{\mathsf{A}} \leftarrow \mathbb{Z}_q$
Algorithm $k_{\mathsf{B}} \leftarrow \mathbb{Z}_q$
$t^0_{\mathsf{A}} := \lambda_{\mathsf{A,B}} \cdot p(\mathsf{A})$
$t^0_{\mathsf{B}} := \lambda_{\mathsf{B,A}} \cdot p(\mathsf{B})$
$R' := k'_{\mathsf{A}} \cdot D_{\mathsf{B}}$
D_{B}
$D_{\mathsf{B}} := k_{\mathsf{B}} \cdot G$
$k_{\mathsf{A}} := H(R') + k'_{\mathsf{A}}$
$\phi + 1/k_{\mathsf{A}}$ $1/k_{\mathsf{A}}$ $t^0_{\mathsf{A}}/k_{\mathsf{A}}$
$1/k_{\mathsf{B}}$ $t^0_{\mathsf{B}}/k_{\mathsf{B}}$
α^1 α^2 α^3
Triple Mul
β^1 β^2
t^1_{A} t^{2a}_{A} t^{2b}_{A} t^1_{B} t^{2a}_{B} t^{2b}_{B}
t^1_{A} $t^2_{\mathsf{A}} := t^{2a}_{\mathsf{A}} + t^{2b}_{\mathsf{A}}$
t^1_{B} $t^2_{\mathsf{B}} := t^{2a}_{\mathsf{B}} + t^{2b}_{\mathsf{B}}$
R'
$R := H(R') \cdot D_{\mathsf{B}} + R'$
$(r_x, r_y) = R := k_{\mathsf{A}} \cdot D_{\mathsf{B}}$
$(r_x, r_y) = R$
$\Gamma^1 := G + \phi \cdot k_{\mathsf{A}} \cdot G - t^1_{\mathsf{A}} \cdot R$
$\Gamma^1 := t^1_{\mathsf{B}} \cdot R$
$\eta^\phi := H(\Gamma^1) + \phi$
η^ϕ
$\phi := \eta^\phi - H(\Gamma^1)$
$\theta := t^1_{\mathsf{B}} - \phi / k_{\mathsf{B}}$
$m' := H(m)$
$m' := H(m)$
$\mathsf{sig}_{\mathsf{A}} := m' \cdot t^1_{\mathsf{A}} + r_x \cdot t^2_{\mathsf{A}}$
$\mathsf{sig}_{\mathsf{B}} := m' \cdot \theta + r_x \cdot t^2_{\mathsf{B}}$
$\Gamma^2 := t^1_{\mathsf{A}} \cdot \mathsf{pk} - t^2_{\mathsf{A}} \cdot G$
$\Gamma^2 := t^2_{\mathsf{B}} \cdot G - \theta \cdot \mathsf{pk}$
$\eta^{\mathsf{sig}} := H(\Gamma^2) + \mathsf{sig}_{\mathsf{A}}$
η^{sig}
$\mathsf{sig} := \mathsf{sig}_{\mathsf{B}} + \eta^{\mathsf{sig}} - H(\Gamma^2)$
$\sigma := (\mathsf{sig} \bmod q, r_x \bmod q)$
abort if $\mathsf{Verify}_{\mathsf{pk}}(\sigma) \neq 1$
Output $\sigma \in (\mathbb{Z}_q, \mathbb{Z}_q)$
$\ell = 4\kappa + 2s$ $\ell' = \ell + \kappa^{\mathsf{OT}}$ $\boldsymbol{c} := \{2^{i-1}\}_{i \in [1,\kappa]} \| \boldsymbol{c}_{\mathsf{R}}$
$\boldsymbol{\gamma}^{\mathsf{mul1}} \leftarrow \{0,1\}^\kappa$ $\boldsymbol{\gamma}^{\mathsf{mul2}} \leftarrow \{0,1\}^\kappa$
$\boldsymbol{\gamma}^{\mathsf{mul3}} \leftarrow \{0,1\}^{2s}$ $\boldsymbol{\gamma}^{\mathsf{ext}} \leftarrow \{0,1\}^{\kappa^{\mathsf{OT}}}$
$\boldsymbol{\omega} := \mathsf{Bits}(\beta^1 - \langle \boldsymbol{c}_{\mathsf{R}}, \boldsymbol{\gamma}^{\mathsf{mul1}} \| \boldsymbol{\gamma}^{\mathsf{mul3}} \rangle) \| \boldsymbol{\gamma}^{\mathsf{mul1}}$
$\| \mathsf{Bits}(\beta^2 - \langle \boldsymbol{c}_{\mathsf{R}}, \boldsymbol{\gamma}^{\mathsf{mul2}} \| \boldsymbol{\gamma}^{\mathsf{mul3}} \rangle) \| \boldsymbol{\gamma}^{\mathsf{mul2}} \| \boldsymbol{\gamma}^{\mathsf{mul3}}$
$\mathbf{w} := \boldsymbol{\omega} \| \boldsymbol{\gamma}^{\mathsf{ext}}$ $w := \sum_{j \in [1,\ell']} 2^{j-1} \cdot \mathbf{w}_j$
$\mathbf{v}^0 := \{\mathsf{Prg}_{\mathsf{ext}_{\mathsf{id}}}(\mathbf{s}^0_i)\}_{i \in [1,\kappa]}$ $\boldsymbol{\psi} := \mathsf{Transpose}(\mathbf{v}^0)$
$\mathbf{v}^1 := \{\mathsf{Prg}_{\mathsf{ext}_{\mathsf{id}}}(\mathbf{s}^1_i)\}_{i \in [1,\kappa]}$
$\mathbf{u} := \{\mathbf{v}^0_i \oplus \mathbf{v}^1_i \oplus w\}_{i \in [1,\kappa]}$ $\boldsymbol{\chi} := \{H(j \| \mathbf{u})\}_{j \in [1,\ell']}$
$w' := \bigoplus_{j \in [1,\ell']} (\mathbf{w}_j \cdot \boldsymbol{\chi}_j)$ $v' := \bigoplus_{j \in [1,\ell']} (\boldsymbol{\psi}_j \wedge \boldsymbol{\chi}_j)$
$\mathbf{u}, w', v'$
$\boldsymbol{\alpha}^1 := \{\boldsymbol{c}_j \cdot \alpha^1\}_{j \in [1,2\kappa+2s]}$
$\boldsymbol{\alpha}^2 := \{\boldsymbol{c}_j \cdot \alpha^2\}_{j \in [1,2\kappa+2s]}$ $\boldsymbol{\alpha}^3 := \{\boldsymbol{c}_j \cdot \alpha^3\}_{j \in [1,2\kappa+2s]}$
$\mathbf{v}^{\nabla} := \{\mathsf{Prg}_{\mathsf{ext}_{\mathsf{id}}}(\mathbf{s}^{\nabla_i}_i)\}_{i \in [1,\kappa]}$
$\mathbf{z} := \{\mathbf{v}^{\nabla_i}_i \oplus (\boldsymbol{\nabla}_i \cdot \mathbf{u}_i)\}_{i \in [1,\kappa]}$ $\boldsymbol{\zeta} := \mathsf{Transpose}(\mathbf{z})$
$z' := \bigoplus_{j \in [1,\ell']} (\boldsymbol{\zeta}_j \wedge H(j \| \mathbf{u}))$ $\nabla := \sum_{i \in [1,\kappa]} 2^{i-1} \cdot \boldsymbol{\nabla}_i$
abort if $z' \neq v' \oplus (w' \wedge \nabla)$
$\boldsymbol{\tau} := \{H^2_1(j \| (\boldsymbol{\zeta}_j \oplus \nabla)) - H^2_1(j \| \boldsymbol{\zeta}_j) + \boldsymbol{\alpha}^1_j\}_{j \in [1,2\kappa]}$
$\| \{H(j \| (\boldsymbol{\zeta}_j \oplus \nabla)) - H(j \| \boldsymbol{\zeta}_j) + \boldsymbol{\alpha}^2_{j-2\kappa}\}_{j \in (2\kappa,4\kappa]}$
$\| \{H^2_2(j \| (\boldsymbol{\zeta}_j \oplus \nabla)) - H^2_2(j \| \boldsymbol{\zeta}_j) + \boldsymbol{\alpha}^3_j\}_{j \in [1,2\kappa]}$
$\| \{H^3_1(j \| (\boldsymbol{\zeta}_j \oplus \nabla)) - H^3_1(j \| \boldsymbol{\zeta}_j) + \boldsymbol{\alpha}^1_{j-2\kappa}\}_{j \in (4\kappa,\ell]}$
$\| \{H^3_2(j \| (\boldsymbol{\zeta}_j \oplus \nabla)) - H^3_2(j \| \boldsymbol{\zeta}_j) + \boldsymbol{\alpha}^2_{j-2\kappa}\}_{j \in (4\kappa,\ell]}$
$\| \{H^3_3(j \| (\boldsymbol{\zeta}_j \oplus \nabla)) - H^3_3(j \| \boldsymbol{\zeta}_j) + \boldsymbol{\alpha}^3_{j-2\kappa}\}_{j \in (4\kappa,\ell]}$
$t^1_{\mathsf{A}} := \sum_{j \in [1,2\kappa]} H^2_1(j \| \boldsymbol{\zeta}_j) + \sum_{j \in (4\kappa,\ell]} H^3_1(j \| \boldsymbol{\zeta}_j)$
$t^{2a}_{\mathsf{A}} := \sum_{j \in (2\kappa,4\kappa]} H(j \| \boldsymbol{\zeta}_j) + \sum_{j \in (4\kappa,\ell]} H^3_2(j \| \boldsymbol{\zeta}_j)$
$t^{2b}_{\mathsf{A}} := \sum_{j \in [1,2\kappa]} H^2_2(j \| \boldsymbol{\zeta}_j) + \sum_{j \in (4\kappa,\ell]} H^3_3(j \| \boldsymbol{\zeta}_j)$
$\boldsymbol{\tau}$
$t^1_{\mathsf{B}} := \sum_{j \in [1,2\kappa]} (\mathbf{w}_j \cdot \boldsymbol{\tau}_j) + \sum_{j \in (4\kappa,\ell]} (\mathbf{w}_j \cdot \boldsymbol{\tau}_{j+2\kappa}) - \sum_{j \in [1,2\kappa]} H^2_1(j \| \boldsymbol{\psi}_j) - \sum_{j \in (4\kappa,\ell]} H^3_1(j \| \boldsymbol{\psi}_j)$
$t^{2a}_{\mathsf{B}} := \sum_{j \in (2\kappa,4\kappa]} (\mathbf{w}_j \cdot \boldsymbol{\tau}_j) + \sum_{j \in (4\kappa,\ell]} (\mathbf{w}_j \cdot \boldsymbol{\tau}_{j+2\kappa+2s}) - \sum_{j \in (2\kappa,4\kappa]} H(j \| \boldsymbol{\psi}_j) - \sum_{j \in (4\kappa,\ell]} H^3_2(j \| \boldsymbol{\psi}_j)$
$t^{2b}_{\mathsf{B}} := \sum_{j \in (4\kappa,6\kappa]} (\mathbf{w}_{j-4\kappa} \cdot \boldsymbol{\tau}_j) + \sum_{j \in (4\kappa,\ell]} (\mathbf{w}_j \cdot \boldsymbol{\tau}_{j+2\kappa+4s}) - \sum_{j \in [1,2\kappa]} H^2_2(j \| \boldsymbol{\psi}_j) - \sum_{j \in (4\kappa,\ell]} H^3_3(j \| \boldsymbol{\psi}_j)$

Fig. 2: **Illustrated 2-of-n Signing Scheme, including Multiplication and OT protocols**. In this diagram, we collapse Protocols 4, 7, and 8 to illustrate the complete, unabstracted 2-of-n signing algorithm. Operations are color-coded according to the logical component with which they are associated: Key Share Reconstruction, Multiplication, Instance Key Exchange, Consistency Check, and Verification/Signing. Note that the multiplication protocol is configured to accept three inputs from Alice and two from Bob, and produce shares of three different products as described in Section VI-D. Alice supplies as inputs a Shamir-Share of the secret key $p(\mathsf{A})$, a secret OTe correlation vector $\boldsymbol{\nabla}$, and a correlated OTe seed vector $\mathbf{s}^{\boldsymbol{\nabla}}$. Bob supplies as inputs a Shamir-Share of the secret key $p(\mathsf{B})$ and two OTe seed vectors $\mathbf{s}^0, \mathbf{s}^1$. He receives as output a signature σ. The two have agreed upon a message m and a random encoding vector $\mathbf{c}_{\mathsf{R}}$.

	Rounds	Communication (Bits)	EC Multiplications		Hash Function Invocations	
			Alice	Bob	Alice	Bob
2-of-2 Setup	5	$\kappa \cdot (5\kappa + 11) + 6$	$3\kappa + 6$	$2\kappa + 6$	$6\kappa + 4$	$6\kappa + 4$
2-of-2 Signing	2	$\kappa \cdot (\kappa^{\mathsf{OT}} + 8\kappa + 6s + 6) + 2$	6	7	$2\kappa^{\mathsf{OT}} + 16\kappa + 12s + 4$	$3\kappa^{\mathsf{OT}} + 16\kappa + 10s + 4$
2-of-n Signing	2	$\kappa \cdot (\kappa^{\mathsf{OT}} + 10\kappa + 8s + 6) + 2$	6	7	$2\kappa^{\mathsf{OT}} + 18\kappa + 14s + 4$	$3\kappa^{\mathsf{OT}} + 18\kappa + 12s + 4$
			Max	Min	Max	Min
2-of-n Setup	5	$(n^2 - n) \cdot (\frac{5}{2}\kappa^2 + 8\kappa + 4)$	$n\kappa - \kappa + 4$	$n + 3$	$5n\kappa - 5\kappa + 1$	$4n\kappa - 4\kappa + 1$

TABLE I: **Communication and Computation Cost Equations For Our Protocol**. We assume that the hash function H is used to implement the PRG. Note that communication costs are totals for all parties over all rounds, whereas computation costs are given per party. In the 2-of-n protocol the computation cost depends upon the identity of the party; consequently we give the minimum and maximum.

	$\kappa = 256$	$\kappa = 384$	$\kappa = 521$
Lindell [2]	769 B	897 B	1043 B
This Work (2-of-2)	85.7 KiB	176.5 KiB	309.2 KiB
Gennaro *et al.* [3]	~1808 KiB	~4054 KiB	~7454 KiB
Boneh *et al.* [4]	~1680 KiB	~3768 KiB	~6924 KiB
This Work (2-of-n)	106.7 KiB	220.0 KiB	385.7 KiB

TABLE II: **Concrete Signing Communication Cost Comparison**. Assuming 2-of-n signing for Gennaro *et al.* and Boneh *et al.*, and 2-of-2 signing for the protocol of Lindell. For our protocols, we use $s = 80$ and $\kappa^{\mathsf{OT}} = 128 + s$.

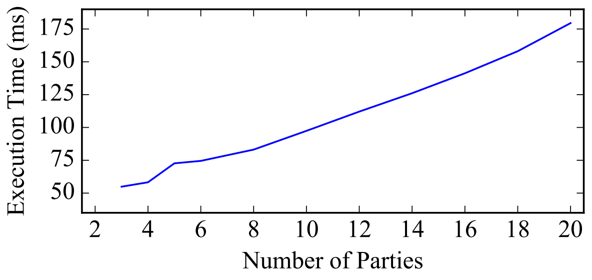


Fig. 3: **Wall Clock Times for 2-of-n Setup over LAN**. Note that all 20 parties reside on individual machines in the same datacenter, and latency is on the order of a few tenths of a millisecond.

	This Work	(3 threads)	[2]	
2-of-2 Setup	44.32	–	2435	
2-of-2 Signing	2.27	2.11	36.8	
	This Work	(3 threads)	[3]	[4]
2-of-n Signing	2.45	2.24	~650	~350

TABLE III: **Wall-clock Times in Milliseconds over LAN**, as compared to the prior approaches of Lindell [2], Gennaro *et al.* [3], and Boneh *et al.* [4]. Note that hardware and networking environments are not necessarily equivalent, but all benchmarks were performed with a single thread except where specified.

Setup			Signing	
2-of-2	2-of-4 (US)	2-of-10 (World)	2-of-2	2-of-n
342.02	376.86	1228.46	76.95	77.06

TABLE IV: **Wall-clock Times in Milliseconds over WAN**. All benchmarks were performed between one party in the eastern US and one in Ireland, except the 2-of-4 setup benchmark, which was performed among four parties in four different US states, and the 2-of-10 setup benchmark, which was performed among ten parties in America, Europe, Asia, and Australia.

configured as before with one instance per party. For initializing OT-extensions, each machine was allowed to use as many threads as there were parties, but the code was otherwise single-threaded. We collected 1000 samples for groups of parties ranging in size from 3 to 20, and we report the results in Figure 3.

Transoceanic Benchmarks: We repeated our 2-of-2 setup, 2-of-2 signing, and 2-of-n signing benchmarks with one of the machines relocated to Amazon's Ireland datacenter, collecting 1,000 samples for setup and 10,000 for signing, and in the latter case allowing three threads for hashing. In this configuration, the bandwidth between our instances was measured to be 161Mbps and the round-trip latency to be 74.6ms. In addition, we performed a 2-of-4 setup benchmark among four instances in Amazon's four US datacenters (Virginia, Ohio, California, and Oregon), and we performed a 2-of-10 setup benchmark among ten instances in ten geographically distributed datacenters (Virginia, Ohio, California, Oregon, Mumbai, Sydney, Canada, Ireland, London, and Paris). The round-trip latency between the US datacenters was between 11.2ms and 79.9ms and the bandwidth between 152Mbps and 1.10Gbps, while round-trip latency between the most distant pair of datacenters, Mumbai and Ireland, was 282ms, and the bandwidth was 39Mbps. Results are reported in Table IV. We note that in contrast to our single-datacenter benchmarks, our transoceanic benchmarks are dominated by latency costs. We expect that our protocol's low round count constitutes a greater advantage in this setting than does its computational efficiency.

A. Comparison to Prior Work

We compare our implementation to those of Lindell [2], Gennaro *et al.* [3], and Boneh *et al.* [4] (who also provide an optimized version of Gennaro *et al.*'s scheme, against which we make our comparison). Though Boneh *et al.* and Gennaro *et al.* support thresholds larger than two, we consider only their performance in the 2-of-n case. Neither Gennaro *et al.* nor Boneh *et al.* include network costs in the timings they provide, nor do they provide timings for the setup protocol that their schemes share. However, Lindell observes that Gennaro *et al.*'s scheme involves a distributed Paillier key generation protocol that requires roughly 15 minutes to run in the semi-honest setting. Unfortunately, this means we have no reliable point of comparison for our 2-of-n setup protocol.

Lindell benchmarks his scheme using a single core on each of two Microsoft Azure `Standard_DS3_v2` instances in the same datacenter, which can expect bandwidth of roughly 3GBits/Second. Lindell's performance figures do include network costs. In spite of the fact that Lindell's protocol requires vastly less communication, as reported in Section VII, we nonetheless find that, not accounting for differences in benchmarking environment, our implementation outperforms his for signing by a factor of roughly 16 (when only a single thread is allowed), and for setup by a factor of roughly 55.

Given that each 2-of-2 signature requires 85.7 KiB of data to be transferred under our scheme, but only 769 Bytes under Lindell's, there must be an environment in which his scheme outperforms ours. Specifically Lindell has an advantage when the protocol is bandwidth constrained but not computationally constrained. Such a scenario is likely when a large number of signatures must calculated in a batched fashion (mitigating the effects of latency) by powerful machines with a comparatively weak network connection.

Finally, we note that an implementation of the ordinary (local) ECDSA signing algorithm in Rust using our own elliptic curve library requires an average of 179 microseconds to calculate a signature on our benchmark machines – a factor of only 11.75 faster than our 2-of-2 signing protocol.

IX. Acknowledgments

We thank Megan Chen and Emily Wang for their contributions to this project during the summer of 2017. The authors of this work are supported by NSF grants TWC-1646671 and TWC-1664445. This work used the Extreme Science and Engineering Discovery Environment (XSEDE) Jetstream cluster [54] through allocation TG-CCR170010, which is supported by NSF grant number ACI-1548562.

X. Code Availability

Our implementation is available under the three-clause BSD license from https://gitlab.com/neucrypt/mpecdsa/.

References

[1] Y. Desmedt, "Society and group oriented cryptography: A new concept," in *CRYPTO*, 1987.

[2] Y. Lindell, *Fast Secure Two-Party ECDSA Signing*, 2017.

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

[4] D. Boneh, R. Gennaro, and S. Goldfeder, "Using level-1 homomorphic encryption to improve threshold dsa signatures for bitcoin wallet security," http://www.cs.haifa.ac.il/~orrd/LC17/paper72.pdf, 2017.

[5] W. Diffie and M. Hellman, "New directions in cryptography," *IEEE Trans. Inf. Theor.*, vol. 22, no. 6, Sep. 1976.

[6] S. Goldwasser, S. Micali, and R. L. Rivest, "A digital signature scheme secure against adaptive chosen-message attacks," *SIAM J. Comput.*, vol. 17, no. 2, Apr. 1988.

[7] 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.

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

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

[10] D. Kravitz, "Digital signature algorithm," jul 1993, uS Patent 5,231,668.

[11] S. Blake-Wilson, N. Bolyard, V. Gupta, C. Hawk, and B. Moeller, "Elliptic curve digital signature algorithm (dsa) for dnssec," https://tools.ietf.org/html/rfc4492, 2006.

[12] P. Hoffman and W. Wijngaards, "Elliptic curve digital signature algorithm (dsa) for dnssec," https://tools.ietf.org/html/rfc6605, 2012.

[13] Bitcoin Wiki, "Transaction," https://en.bitcoin.it/wiki/Transaction, 2017, accessed Oct 22, 2017.

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

[15] Y. G. Desmedt and Y. Frankel, "Threshold cryptosystems," in *CRYPTO*, 1989.

[16] T. ElGamal, "A public key cryptosystem and a signature scheme based on discrete logarithms," in *CRYPTO*, 1984.

[17] A. Shamir, "How to share a secret," *Commun. ACM*, vol. 22, no. 11, Nov. 1979.

[18] T. P. Pedersen, "A threshold cryptosystem without a trusted party," in *EUROCRYPT*, 1991.

[19] Y. Desmedt and Y. Frankel, "Shared generation of authenticators and signatures (extended abstract)," in *CRYPTO*, 1991.

[20] R. L. Rivest, A. Shamir, and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," *Commun. ACM*, vol. 21, no. 2, Feb. 1978.

[21] Y. Desmedt and Y. Frankel, "Parallel reliable threshold multisignature," 1992.

[22] R. Gennaro, S. Jarecki, H. Krawczyk, and T. Rabin, "Robust and efficient sharing of rsa functions," in *CRYPTO*, 1996.

[23] A. De Santis, Y. Desmedt, Y. Frankel, and M. Yung, "How to share a function securely," in *STOC*, 1994.

[24] V. Shoup, "Practical threshold signatures," in *EUROCRYPT*, 2000.

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

[26] D. R. Stinson and R. Strobl, "Provably secure distributed schnorr signatures and a (t, n) threshold scheme for implicit certificates," in *ACISP*, 2001.

[27] S. K. Langford, "Threshold dss signatures without a trusted party," in *CRYPTO*, 1995.

[28] R. Gennaro, S. Jarecki, H. Krawczyk, and T. Rabin, "Robust threshold dss signatures," in *EUROCRYPT*, 1996.

[29] P. MacKenzie and M. K. Reiter, *Two-Party Generation of DSA Signatures*, 2001.

[30] Bitcoin Wiki, "Multisignature," https://en.bitcoin.it/wiki/Multisignature, 2017, accessed Oct 22, 2017.

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

[32] N. Gilboa, "Two party rsa key generation," in *CRYPTO*, 1999.

[33] T. Chou and C. Orlandi, "The simplest protocol for oblivious transfer," in *LATINCRYPT*, 2015.

[34] M. Keller, E. Orsini, and P. Scholl, “Actively secure OT extension with optimal overhead,” in *CRYPTO*, 2015.
[35] E. Hauck and J. Loss, “Efficient and universally composable protocols for oblivious transfer from the cdh assumption,” Cryptology ePrint Archive, Report 2017/1011, 2017, http://eprint.iacr.org/2017/1011.
[36] J. Katz and Y. Lindell, *Introduction to Modern Cryptography, Second Edition*, 2015, ch. Digital Signature Schemes, pp. 443–486.
[37] D. R. L. Brown, “Generic groups, collision resistance, and ecdsa,” Cryptology ePrint Archive, Report 2002/026, 2002, http://eprint.iacr.org/2002/026.
[38] S. Vaudenay, “The security of dsa and ecdsa,” in *PKC*, 2003.
[39] N. Koblitz and A. Menezes, “Another look at generic groups,” Cryptology ePrint Archive, Report 2006/230, 2006, https://eprint.iacr.org/2006/230.
[40] S. Even, O. Goldreich, and A. Lempel, “A randomized protocol for signing contracts,” *Commun. ACM*, vol. 28, no. 6, Jun. 1985.
[41] M. O. Rabin, “How to exchange secrets with oblivious transfer,” Cryptology ePrint Archive, Report 2005/187, 1981, http://eprint.iacr.org/2005/187, Harvard University Technical Report 81.
[42] S. Wiesner, “Conjugate coding,” *SIGACT News*, 1983.
[43] M. Naor and B. Pinkas, “Computationally secure oblivious transfer,” *J. Cryptol.*, vol. 18, no. 1, Jan. 2005.
[44] D. Beaver, “Correlated pseudorandomness and the complexity of private computations,” in *STOC*, 1996.
[45] Y. Ishai, E. Kushilevitz, R. Ostrovsky, M. Prabhakaran, and A. Sahai, “Efficient non-interactive secure computation,” in *EUROCRYPT*, 2011.
[46] A. Beimel, A. Gabizon, Y. Ishai, E. Kushilevitz, S. Meldgaard, and A. Paskin-Cherniavsky, “Non-interactive secure multiparty computation,” in *CRYPTO*, 2014.
[47] V. Shoup, “Lower bounds for discrete logarithms and related problems,” in *EUROCRYPT*, 1997.
[48] A. Fiat and A. Shamir, “How to prove yourself: Practical solutions to identification and signature problems,” in *CRYPTO*, 1986.
[49] M. Fischlin, “Communication-efficient non-interactive proofs of knowledge with online extractors,” in *CRYPTO*, 2005.
[50] E. Waring, *Philosophy Transactions*, no. 69, pp. 59–67, 1779.
[51] M. Byali, A. Patra, D. Ravi, and P. Sarkar, “Efficient, round-optimal, universally-composable oblivious transfer and commitment scheme with adaptive security,” Cryptology ePrint Archive, Report 2017/1165, 2017, https://eprint.iacr.org/2017/1165.
[52] P. S. L. M. Barreto, B. David, R. Dowsley, K. Morozov, and A. C. A. Nascimento, “A framework for efficient adaptively secure composable oblivious transfer in the rom,” Cryptology ePrint Archive, Report 2017/993, 2017, https://eprint.iacr.org/2017/993.
[53] R. Impagliazzo and M. Naor, “Efficient cryptographic schemes provably as secure as subset sum,” *J. Cryptol.*, vol. 9, no. 4, Sep 1996.
[54] C. Stewart, T. Cockerill, I. Foster, D. Hancock, N. Merchant, E. Skidmore, D. Stanzione, J. Taylor, S. Tuecke, G. Turner, M. Vaughn, and N. Gaffney, “Jetstream: a self-provisioned, scalable science and engineering cloud environment,” in *XSEDE Conference: Scientific Advancements Enabled by Enhanced Cyberinfrastructure*, 2015. [Online]. Available: http://dx.doi.org/10.1145/2792745.2792774
[55] M. Fischlin, “A note on security proofs in the generic model,” in *ASIACRYPT*, 2000.
[56] J. Stern, D. Pointcheval, J. Malone-Lee, and N. P. Smart, “Flaws in applying proof methodologies to signature schemes,” in *Advances in Cryptology — CRYPTO 2002*, M. Yung, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 93–110.
[57] A. W. Dent, “Adapting the weaknesses of the random oracle model to the generic group model,” in *Advances in Cryptology — ASIACRYPT 2002*, Y. Zheng, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 100–109.
[58] D. Boneh and X. Boyen, “Short signatures without random oracles,” in *Advances in Cryptology - EUROCRYPT 2004*, C. Cachin and J. L. Camenisch, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 56–73.
[59] D. Boneh, X. Boyen, and H. Shacham, “Short group signatures,” in *Advances in Cryptology – CRYPTO 2004*, M. Franklin, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 41–55.

Appendix A
Additional Functionalities

In this section, we present the additional functionalities on which our protocols rely. As before, we omit notation for bookkeeping elements that we do not explicitly use such as session IDs and party specifiers, which work in the ordinary way; we also assume that if messages are received out of order for a particular session, the functionality aborts. We begin with a Selective-failure OT functionality, which differs from the traditional OT functionality in that it allows the sender to guess the receiver's choice bit. If the sender's guess is incorrect, the functionality alerts both parties, and if the sender's guess is correct, then the sender is notified while the receiver is not.

Functionality 3. $\mathcal{F}_{\mathsf{SF\text{-}OT}}$**:**

This functionality is parameterized by the group order q and runs with two parties, a sender and a receiver.

Choose: On receiving $(\texttt{choose}, \omega)$ from the receiver, store $(\texttt{choice}, \omega)$ if no such message exists in memory and send $(\texttt{chosen})$ to the sender.

Guess: On receiving $(\texttt{guess}, \hat{\omega})$ from the sender, if $\hat{\omega} \in \{0, 1, \bot\}$ and if $(\texttt{choice}, \omega)$ exists in memory, and if $(\texttt{guess}, \cdot)$ does not exist in memory, then store $(\texttt{guess}, \hat{\omega})$ in memory and do the following:

1) If $\hat{\omega} = \bot$, send $(\texttt{no-cheat})$ to the receiver.
2) If $\hat{\omega} = \omega$, send $(\texttt{cheat-undetected})$ to the sender and $(\texttt{no-cheat})$ to the receiver.
3) Otherwise, send $(\texttt{cheat-detected})$ to both the sender and receiver.

Transfer: On receiving $(\texttt{transfer}, \alpha^0, \alpha^1)$ from the sender, if $\alpha^0 \in \mathbb{Z}_q$ and $\alpha^1 \in \mathbb{Z}_q$, and if $(\texttt{complete})$ does not exist in memory, and if there exist in memory messages $(\texttt{choice}, \omega)$ and $(\texttt{guess}, \hat{\omega})$ such that $\hat{\omega} = \bot$ or $\hat{\omega} = \omega$, then send $(\texttt{message}, \alpha^\omega)$ to the receiver and store $(\texttt{complete})$ in memory.

What follows is a Correlated OT-extension functionality that allows arbitrarily many Correlated OT instances to be executed in batches of size ℓ. For each batch, the receiver inputs a vector of choice bits $\boldsymbol{\omega} \in \{0,1\}^\ell$, following which the sender inputs a vector of correlations $\boldsymbol{\alpha} \in \mathbb{Z}_q^\ell$. The functionality samples ℓ random pads from $\mathbb{Z}_q$ and sends them to the sender. To the receiver it sends only the the pads if the sender's corresponding choice bits were 0, or the sum of the pads and their corresponding correlations if the sender's corresponding choice bits were 1. Note that this functionality is nearly identical to the one presented by Keller *et al.* [34], but we add an initialization phase and the ability to perform extensions (each batch of extensions indexed by a fresh extension index $\mathsf{ext_{id}}$) only after the initialization has been performed.

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

This functionality is parameterized by the group order q and the batch size ℓ. It runs with two parties, a sender S and a receiver R, who may participate in the Init phase once, and the Choice and Transfer phases as many times as they wish.

Init: Wait for message $(\texttt{ready})$ from the sender and receiver. Store $(\texttt{ready})$ in memory and send $(\texttt{init-complete})$ to the receiver.

Choice: On receiving $(\texttt{choose}, \mathsf{ext_{id}}, \boldsymbol{\omega})$ from the receiver, if $(\texttt{choice}, \mathsf{ext_{id}}, \cdot)$ with the same $\mathsf{ext_{id}}$ does not exist in memory, and if $(\texttt{ready})$ does exist in memory, and if $\boldsymbol{\omega}$ is of the correct form, then send $(\texttt{chosen})$ to the sender and store $(\texttt{choice}, \mathsf{ext_{id}}, \boldsymbol{\omega})$ in memory.
Transfer: On receiving $(\texttt{transfer}, \mathsf{ext_{id}}, \boldsymbol{\alpha})$ from the sender, if there exists a message of the form $(\texttt{choice}, \mathsf{ext_{id}}, \boldsymbol{\omega})$ in memory with the same $\mathsf{ext_{id}}$, and if $(\texttt{complete}, \mathsf{ext_{id}})$ does not exist in memory, and if $\boldsymbol{\alpha}$ is of the correct form, then:

1) Sample a vector of random pads $\mathbf{t}_{\mathsf{S}} \leftarrow \mathbb{Z}_q^{\ell}$
2) Send $(\texttt{pads}, \mathbf{t}_{\mathsf{S}})$ to the sender.
3) Compute $\{\mathbf{t}_{\mathsf{R}i}\}_{i\in[1,\ell]} := \{\mathbf{t}_{\mathsf{S}i} + \boldsymbol{\omega}_i \cdot \boldsymbol{\alpha}_i\}_{i\in[1,\ell]}$.
4) Send $(\texttt{padded-correlation}, \mathbf{t}_{\mathsf{R}})$ to the receiver.
5) Store $(\texttt{complete}, \mathsf{ext_{id}})$ in memory.

Finally, we give functionalities for zero-knowledge proofs-of-knowledge-of-discrete-logarithm. The first corresponds to an ordinary proof, whereas the second allows the prover to commit to a proof that will later be revealed. Note that these are both standard constructions, except that they operate with groups of parties, and all parties aside from the prover receive verification.

Functionality 5. $\mathcal{F}_{\mathsf{ZK}}^{R_{\mathsf{DL}}}$**:**

The functionality is parameterized by the group $\mathbb{G}$ of order q generated by G, and runs with a group of parties $\mathbf{P}$ such that $|\mathbf{P}| = n$.
Proof: On receiving $(\texttt{prove}, x, X)$ from $\mathbf{P}_i$ where $x \in \mathbb{Z}_q$ and $X \in \mathbb{G}$, if $X = x \cdot G$, then send $(\texttt{accept}, i, X)$ to all parties in $\mathbf{P}$. Otherwise, send $(\texttt{fail}, i, X)$ to all parties in $\mathbf{P}$.

Functionality 6. $\mathcal{F}_{\mathsf{Com\text{-}ZK}}^{R_{\mathsf{DL}}}$**:**

The functionality is parameterized by the group $\mathbb{G}$ of order q generated by G, and runs with a group of parties $\mathbf{P}$ such that $|\mathbf{P}| = n$.
Commit Proof: On receiving $(\texttt{com-proof}, x, X)$ from $\mathbf{P}_i$, where $x \in \mathbb{Z}_q$ and $X \in \mathbb{G}$, store $(\texttt{com-proof}, x, X)$ and send $(\texttt{committed}, i)$ to all parties in $\mathbf{P}$.
Decommit Proof: On receiving $(\texttt{decom-proof})$ from $\mathbf{P}_i$, if $(\texttt{com-proof}, x, X)$ exists in memory, then:

1) If $X = x \cdot G$, send $(\texttt{accept}, i, X)$ to all parties in $\mathbf{P}$.
2) Otherwise send $(\texttt{fail}, i, X)$ all parties in $\mathbf{P}$.

Appendix B
Equivalence of Functionalities

We argue that our functionality $\mathcal{F}_{\mathsf{SampledECDSA}}$ (Functionality 2) does not grant any additional power to Alice by showing that an adversary who is able to forge a signature by observing the signatures produced by accessing $\mathcal{F}_{\mathsf{SampledECDSA}}$ can be used to forge an ECDSA signature in the standard Existential Unforgeability experiment that defines security for signature schemes (see Katz and Lindell [36] for a complete description of the experiment). We are only concerned with arguing that an ideal adversary interacting with $\mathcal{F}_{\mathsf{SampledECDSA}}$ as *Alice* is unable to forge a signature because Bob's view in his ideal interaction with $\mathcal{F}_{\mathsf{SampledECDSA}}$ is identical to his view when interacting with $\mathcal{F}_{\mathsf{ECDSA}}$ (Functionality 1).

Our reduction is in the Generic Group Model, which was introduced by Shoup [47]. While there are well-known criticisms of this model [55]–[57], it has also shown itself to be useful in proving the security of well-known constructions such as Short Signatures [58] and Short Group Signatures [59]. Furthermore, this is the model in which ECDSA itself is proven secure [37].

In this model an adversary can perform group operations only by querying a Group Oracle $\mathcal{G}(\cdot)$. More specifically, queries of the following types are answered by the Oracle:

1) (Group Elements) When the Oracle receives an integer $x \in \mathbb{Z}_q$, it replies with an encoding of the group element corresponding to this integer. Returned encodings are random, but the Oracle is required to be consistent when the same integer is queried repeatedly. This corresponds to the scalar multiplication operation with the generator in an ECDSA group: $Y := x \cdot G$.
2) (Group Law) When the Oracle receives a tuple of the form $(r, s, \mathcal{G}(x), \mathcal{G}(y))$, it replies with a random encoding of the group element given by $\mathcal{G}(r \cdot x + s \cdot y)$. As before, outputs must be consistent. This corresponds to a fused multiply-add operation in an ECDSA group: $Z := (r \cdot X + s \cdot Y)$, where $X = x \cdot G$ and $Y = y \cdot G$.

As usual in this model, the reduction itself will control the Group Oracle, and in particular it has the ability to program the Oracle to respond to specific queries with specific outputs. $\mathcal{F}^{\mathsf{A}}_{\mathsf{SampledECDSA}}$ is used to denote an Oracle version of the $\mathcal{F}_{\mathsf{SampledECDSA}}$ functionality accessible only as Alice. In addition to the previously defined $\mathcal{F}_{\mathsf{SampledECDSA}}$ behavior, this Oracle returns the signature $\sigma_{\mathsf{sig_{id}}}$ to Alice upon receiving $(\texttt{sign}, \mathsf{sig_{id}}, \cdot, \cdot)$. This models the realistic scenario wherein Alice obtains the output signatures, which we wish to capture in our reduction, even though the functionality does not output the signature to her on its own.

Claim B.1. *If there exists a probabilistic polynomial time algorithm* A *in the Generic Group Model with access to the* $\mathcal{F}^{\mathsf{A}}_{\mathsf{SampledECDSA}}$ *oracle, such that*

$$\Pr\left[\begin{array}{r}\mathsf{Verify_{pk}}(m,\sigma) = 1 \wedge m \notin \mathbf{Q} : \\ (m,\sigma) \leftarrow \mathsf{A}^{\mathcal{F}^{\mathsf{A}}_{\mathsf{SampledECDSA}}}(\mathsf{pk})\end{array}\right] \geq p(\kappa)$$

where $\mathbf{Q}$ *is the set of messages for which* A *sends queries of the form* $(\textit{new}, \cdot, m, \cdot)$ *to the* $\mathcal{F}^{\mathsf{A}}_{\mathsf{SampledECDSA}}$ *Oracle, and where the probability is taken over the randomness of the* $\mathcal{F}_{\mathsf{SampledECDSA}}$ *functionality, then there exists an adversary* $\mathcal{A}$ *such that*

$$\Pr_{\mathsf{pk},\mathsf{sk}}\left[\begin{array}{r}\mathsf{Verify_{pk}}(m,\sigma) = 1 \wedge m \notin \mathbf{Q} : \\ (m,\sigma) \leftarrow \mathcal{A}^{\mathsf{Sign_{sk}}(\cdot)}(\mathsf{pk})\end{array}\right] \geq p(\kappa) - \frac{\mathrm{poly}(\kappa)}{2^{-\kappa}}$$

where $\mathbf{Q}$ *is the set of messages for which* $\mathcal{A}$ *queries the signing oracle* $\mathsf{Sign_{sk}}(\cdot)$.

Proof sketch. Our reduction is structured in an intuitive way. For readability we refer to A as Alice in its interactions with $\mathcal{F}^{\mathsf{A}}_{\mathsf{SampledECDSA}}$, and we note that $\mathcal{A}$ can only interact with Alice on behalf of the $\mathcal{F}^{\mathsf{A}}_{\mathsf{SampledECDSA}}$ Oracle. First, $\mathcal{A}$ forces Alice to accept the same public key that it received externally in the forgery game, and then, for each query Alice makes to her $\mathcal{F}^{\mathsf{A}}_{\mathsf{SampledECDSA}}$ oracle, $\mathcal{A}$ can request a corresponding signature from the $\mathsf{Sign}_{\mathsf{sk}}$ oracle under the same secret key. The nonce R^{sig} in the signature received from $\mathsf{Sign}_{\mathsf{sk}}$ will not match the nonce R that Alice instructs the $\mathcal{F}^{\mathsf{A}}_{\mathsf{SampledECDSA}}$ oracle to use. However, $\mathcal{A}$ can take advantage of the fact that $\mathcal{F}^{\mathsf{A}}_{\mathsf{SampledECDSA}}$ is allowed to offset the nonce R by a random value k^{Δ} of its choosing. $\mathcal{A}$ sets k^{Δ} so that $k^{\Delta} \cdot G$ is exactly the difference between R and R^{sig}. Computing k^{Δ} directly would require $\mathcal{A}$ to know the discrete log of the R^{sig} value it was given by the $\mathsf{Sign}_{\mathsf{sk}}$ oracle; instead, $\mathcal{A}$ uses its ability to program the Group Oracle to ensure that $\mathcal{G}(k^{\Delta})$ is the difference between R and the corresponding R^{sig}. We describe $\mathcal{A}^{\mathsf{Sign}_{\mathsf{sk}}(\cdot)}$ formally below.

Algorithm 4. $\mathcal{A}^{\mathsf{Sign}_{\mathsf{sk}}(\cdot)}(\mathsf{pk})$**:**

1) Answer any query $\mathcal{G}(x)$ as $x \cdot G$, and any query $\mathcal{G}(r, s, \mathcal{G}(x), \mathcal{G}(y))$ as $r \cdot \mathcal{G}(x) + s \cdot \mathcal{G}(y)$ unless otherwise explicitly programmed at those points.
2) Send $(\texttt{public-key}, \mathsf{pk})$ to Alice.
3) When a message of the form $(\texttt{new}, \mathsf{sig}_{\mathsf{id}}, m, \mathsf{B})$ is received from Alice, sample $k_{\mathsf{B}}^{\mathsf{sig}_{\mathsf{id}}} \leftarrow \mathbb{Z}_q$, calculate $D_{\mathsf{B}} := k_{\mathsf{B}}^{\mathsf{sig}_{\mathsf{id}}} \cdot G$, store $(\texttt{sig-message}, \mathsf{sig}_{\mathsf{id}}, m, k_{\mathsf{B}}^{\mathsf{sig}_{\mathsf{id}}})$ in memory, and reply to Alice with
$$(\texttt{nonce-shard}, \mathsf{sig}_{\mathsf{id}}, D_{\mathsf{B}})$$
4) When a message of the form $\left(\texttt{nonce}, \mathsf{sig}_{\mathsf{id}}, i, R_{i,\mathsf{sig}_{\mathsf{id}}}\right)$ is received from Alice, if $(\texttt{sig-message}, \mathsf{sig}_{\mathsf{id}}, m, k_{\mathsf{B}}^{\mathsf{sig}_{\mathsf{id}}})$ exists in memory:
   a) Query the Signing Oracle with the message m to obtain a signature
   $$\left(\mathsf{sig}_{\mathsf{sig}_{\mathsf{id}},i}, R^{\mathsf{sig}}_{\mathsf{sig}_{\mathsf{id}},i}\right) = \sigma_{\mathsf{sig}_{\mathsf{id}},i} \leftarrow \mathsf{Sign}_{\mathsf{sk}}(m)$$
   Note that the oracle will only return the x-coordinate of $R^{\mathsf{sig}}_{\mathsf{sig}_{\mathsf{id}},i}$, but recovering the point itself is easy. Store $(\texttt{sig-signature}, \mathsf{sig}_{\mathsf{id}}, \sigma_{\mathsf{sig}_{\mathsf{id}},i})$ in memory.
   b) Sample $k^{\Delta}_{\mathsf{sig}_{\mathsf{id}},i} \leftarrow \mathbb{Z}_q$, then compute
   $$K^{\Delta}_{\mathsf{sig}_{\mathsf{id}},i} := R^{\mathsf{sig}}_{i,\mathsf{sig}_{\mathsf{id}}} - R_{i,\mathsf{sig}_{\mathsf{id}}}$$
   and program the Group Oracle such that
   $$\mathcal{G}\left(k^{\Delta}_{\mathsf{sig}_{\mathsf{id}}}\right) = K^{\Delta}_{\mathsf{sig}_{\mathsf{id}},i}$$
   c) Compute
   $$k^{\Delta}_{\mathsf{sig}_{\mathsf{id}},i,\mathsf{A}} = (1/k_{\mathsf{B}}^{\mathsf{sig}_{\mathsf{id}}}) \cdot k^{\Delta}_{\mathsf{sig}_{\mathsf{id}}}$$
   and program the Group Oracle such that
   $$\mathcal{G}(k^{\Delta}_{\mathsf{sig}_{\mathsf{id}},i,\mathsf{A}}) = (1/k_{\mathsf{B}}^{\mathsf{sig}_{\mathsf{id}}}) \cdot K^{\Delta}_{\mathsf{sig}_{\mathsf{id}},i}$$
   d) Send $(\texttt{offset}, \mathsf{sig}_{\mathsf{id}}, k^{\Delta}_{\mathsf{sig}_{\mathsf{id}},i,\mathsf{A}})$ to Alice.
5) When a message of the form $(\texttt{sign}, \mathsf{sig}_{\mathsf{id}}, i, k_{\mathsf{A}})$ is received from Alice, if $(\texttt{sig-signature}, \mathsf{sig}_{\mathsf{id}}, \sigma_{\mathsf{sig}_{\mathsf{id}},i})$ and $(\texttt{sig-message}, \mathsf{sig}_{\mathsf{id}}, m, k_{\mathsf{B}}^{\mathsf{sig}_{\mathsf{id}}})$ exist in memory, and $k_{\mathsf{A}} \cdot k_{\mathsf{B}}^{\mathsf{sig}_{\mathsf{id}}} \cdot G = R^{\mathsf{sig}}_{i\mathsf{sig}_{\mathsf{id}}}$, but $(\texttt{sig-complete}, \mathsf{sig}_{\mathsf{id}})$ does not exist in memory, respond with $\sigma_{\mathsf{sig}_{\mathsf{id}},i}$ and store $(\texttt{sig-complete}, \mathsf{sig}_{\mathsf{id}})$ in memory.
6) Once Alice outputs a forged signature sig^*, output this signature.

Notice that this reduction fails if Alice queries $\mathcal{G}$ on an index $k^{\Delta}_{\mathsf{sig}_{\mathsf{id}},i,\mathsf{A}}$ for any $\mathsf{sig}_{\mathsf{id}}$ and any i before $\mathcal{A}$ programs it, or if she queries it on an index $k_{\mathsf{B}}^{\mathsf{sig}_{\mathsf{id}}}$ for any $\mathsf{sig}_{\mathsf{id}}$ at any time. By a standard argument, this event occurs with probability $\mathrm{poly}(\kappa)/2^{\kappa}$. If these queries are not made, the reduction is perfect and the claim follows. $\square$