Trout++: Robust Asynchronous Two-Round ECDSA for Arbitrary Thresholds

Ariel Nof[1] and Luke Parker[2]

[1] Bar-Ilan University, Ramat-Gan, Israel
ariel.nof@biu.ac.il
[2] Serai DEX, Florida, USA
lukeparker@serai.exchange

Abstract. We present Trout++, a complete threshold signing suite for ECDSA signatures. Trout++ descends from the recent Trout protocol (Dahari-Garbian, Nof, and Parker, ACM CCS 2025) and inherits its transparent setup, two-round structure, and strong security guarantees, while introducing several significant improvements.

Unlike Trout, Trout++ offers pre-signing, where the first round is key-, signing-set-, and message- independent. This property is not only important in its own right but also enables us to apply the ROAST transformation (Ruffing, Ronge, Jin, Schneider-Bensch, and Schröder, ACM CCS 2022) to our protocol, yielding the *first* arbitrary-threshold (including with a dishonest majority), robust, asynchronous signing protocol for ECDSA signatures.

Furthermore, we introduce several optimizations to the building blocks in Trout that reduce both bandwidth and computation. Our benchmark results show that our implementation of Trout++ is approximately twice as fast *and* twice as small compared to the prior implementation of Trout. We also present a constant-time implementation with running time that is *ten times faster* than Trout's constant-time implementation.

Finally, we show how Trout++ can be composed with an account derivation scheme that does not require additional setups per derivation, enabling more practical solutions for managing key material.

In total, we achieve complexities and functionality comparable to, and closing the gap with, leading solutions for Schnorr signatures (such as FROST).

1 Introduction

ECDSA, the Elliptic Curve Digital Signature Algorithm, is a widely-deployed signature scheme. Standardized by the National Institute of Standards and Technology, it is used for Transport Layer Security (TLS) encryption, authenticating Domain Name System (DNS) records via DNSSEC, and is frequently used for signing transactions in cryptocurrency networks such as Bitcoin and Ethereum.

In the setting of *threshold* ECDSA, the private signing key is distributed across n parties with a threshold t, such that only a subset of $t + 1$ parties can reconstruct the key. To sign messages, any $t + 1$ parties must run a secure

protocol such that only the generated signature is learned by the parties, but no information about the private key is revealed. The problem of constructing efficient threshold ECDSA protocols has attracted considerable attention in the last decade or so, the challenge stemming from the fact that, unlike schemes for Schnorr, EdDSA, and BLS, the ECDSA signature is not "MPC-friendly". Producing a signature requires computing the multiplicative inverse of a nonce and multiplying it by (a derivative of) the private key. These are non-linear operations that are rather costly when computed using generic MPC techniques.

The first solution for arbitrary thresholds was introduced in [18] and [2], but with an impractical distributed key generation protocol (requiring many parties to perform a distributed Paillier key generation). In 2018, concurrent works by Gennaro and Goldfeder [17], Lindell and Nof [22], and Doerner et al. [15] presented the first arbitrary-threshold protocols for multi-party ECDSA with practical distributed key generation and signing. Since then, a long line of works proposing new protocols has emerged, aiming to improve efficiency across various metrics, offering different assumptions and cryptographic tools, and enhancing the functionality provided.

Recently, the first two-round signing protocols for threshold ECDSA were independently published: Trout [13] and the protocol by Lyu, Li, Zhou, and Deng [23]. These were the first protocols to achieve the optimal round complexity for ECDSA signatures, and in the bulletin board model, each achieved constant per-participant upload (a single message per round of size independent to the threshold, signing set). There are also some differences between the two works. Unlike Trout, [23] achieved the desired property of pre-signing, but with weaker security guarantees compared to Trout. In particular, Trout achieved the identifiable abort (IA) property, meaning that parties can identify a cheater when the protocol does not terminate successfully. Trout also offered a publicly verifiable transcript of linear complexity and relied on simulation-based proofs (whereas the security proof in [23] relies on game-based definitions).[3]

Given the above, the gap between state-of-the-art threshold ECDSA protocols and their MPC-friendly counterparts, such as Schnorr, seems almost closed. Indeed, for Schnorr, the best existing protocols [3, 21] also require two rounds.

However, some significant gaps still remain. Despite comparable complexities, the popular and widely deployed threshold Schnorr protocol, FROST [21] (as extended via following works and ecosystem adoption), still offers functionality not generally obtained by ECDSA threshold protocols. First, FROST supports account derivation as extended in Re-Randomized Frost [19], allowing the presentation of multiple identities without running additional setups or distributed key-generation protocols. Not only does this avoid executing protocols with non-trivial computational burden, it also simplifies recovery procedures by eliminating repeated setups, each which must be securely stored. Since 2012, Bitcoin

[3] Simulation based-proofs have the advantage of not requiring the forking lemma, rewinding, and so on, in order to prove security. In addition, it significantly reduces the required assumptions and provides tight security including during concurrent composition.

	Communication per party per execution		Presigning	IA	Security Type	Robust & Asynchronous
	w/o IA	IA				
Trout [13]	3.9 KB	6.3 KB	✗	✓	Simulation	✗
Lyu et al. [23]	1.9 KB	-	✓	✗	Game-based	✗
Trout++	1.8 KB	3.8 KB	✓	✓	Simulation	✓

Table 1. Comparison between Trout++ and the previous works which achieved threshold ECDSA in two rounds. Trout++ is only robust and asynchronous when composed with ROAST (requiring IA), which will perform multiple executions of Trout++.

Improvement Proposal 32 (BIP32) [4] has detailed a scheme for Bitcoin wallets to generate multiple Bitcoin addresses from a single private key, and this has become an expected functionality for users.

Furthermore, FROST can be transformed into a robust, asynchronous signing protocol via ROAST [25]. ROAST is a transform which may be applied to any protocol that meets certain security definitions and structural properties. Most notably, it requires the underlying protocol to be a two-round signing protocol unforgeable during concurrent executions, with identifiable aborts, and with support for signing-set- and message- independent presigning. Unfortunately, as explained above, both [13] and [12] achieve only some of these properties. We note that the first round of Trout relies crucially on a primitive called exponent VRF [3], which takes the signing-set and message as inputs, so achieving presigning would initially appear nontrivial. Similarly, the protocol of [12] relies on a primitive called non-interactive multiplication [5], which is not known to feasibly allow identifying cheaters.

1.1 Our Contributions

In this work, we close the gaps listed above by presenting Trout++, a multiparty ECDSA protocol for arbitrary thresholds, with the following features (see Table 1 for a summary and comparison):

- The signing protocol has two rounds, with the first round key-, signing-set-, and message- independent, thereby enabling pre-signing.
- It achieves security with identifiable aborts (IA).
- It has constant per party communication of just 3.8 KB with IA and 1.8 KB without IA in the bulletin board model.
- It is proven secure under strong simulation-based proofs.
- It achieves robustness and works in the asynchronous setting via the ROAST transformation. The resulting composition optimistically takes as few as two rounds, and in the worst case, takes at most $2 \cdot (n - (t + 1)) + 3$ rounds with $\mathcal{O}((t + 1) \cdot (n - t)^2 \cdot (n + ((t + 1) \cdot \kappa)))$ communication complexity[5].

[4] https://github.com/bitcoin/bips/blob/master/bip-0032.mediawiki

[5] n = number of parties, t = corruption threshold, κ = security parameter

In addition, we show how Trout++ can be used securely with an account-derivation scheme, allowing the creation of an arbitrary number of identities despite a constant number of key generations.

We have implemented Trout++ and benchmarked it to show its practicality and improvements over Trout. With an efficient, composable protocol that supports account derivation, and offers strong security guarantees, we close the gap between threshold ECDSA and Schnorr signing in a way not seen before.

Technical Contribution. As its name suggests, Trout++ builds upon the approach of Trout, while extending it to support pre-signing and reducing its bandwidth. This is achieved without giving up on the strong security guarantees of Trout. Recall that for a message m and a signing key x, the ECDSA signature is computed as $k^{-1} \cdot (\mathcal{H}(m) + r \cdot x)$, where r is the x-coordinate of $R = k \cdot E$ and E is the group generator. At a high level, the first round of Trout is dedicated to producing R, an encryption of k, and a commitment to a random mask. In the second round, the parties perform a secure protocol called *scaled decryption*, which multiplies encrypted and committed data *and* decrypts the result in a single round. The public output of this step enables the parties to locally compute the signature. We present improvements to Trout for both steps:

- *Computing $R = k \cdot E$ in one round*: We remove the non-trivial signing-set- and message- dependent exponent VRF [3] used in Trout, replacing it with the methodology of [1], which is more efficient and does not require such context. This subprotocol works by first producing R in a naive way and then re-randomizing it by calling a random oracle over the transcript twice to receive random values ρ and μ. The final nonce is set to $\rho \cdot R + \mu \cdot E$. In [1], it was shown that this only incurs a small constant loss in security compared to standard ECDSA signing. Unlike [23], who also used this methodology, we prove that the protocol realizes a carefully defined ideal signing functionality with presigning, rather than proving security via a game-based definition.
- *Improving scaled decryption*: In Trout, the scaled decryption protocol enables multiplication and decryption in a single round over two inputs, one of which is encrypted in a CLT ciphertext [10] and the other committed to with what is approximately a Pedersen commitment. We improve the above by replacing the public-key encryption with a one-time symmetric encryption scheme and a new, optimized commitment scheme. Then, we present a new scaled decryption protocol that works over these two building blocks. Note that this new protocol not only reduces bandwidth and computation but also avoids the use of public-key encryption in our protocol, thereby removing the need to generate a CLT public key and yielding a simpler and more efficient setup. The usage of symmetric encryption in place of public-key encryption may be of inspiration to other MPC protocols.

2 Preliminaries

Notations. We use lower-case letters to represent scalars, and upper-case letters to denote group elements. Variables used as randomness within cipher-

texts/commitments are denoted with greek letters. We use || to denote string concatenation. For a prime q, let $\mathbb{F}_q$ denote a finite field of size q. We use κ to denote a fixed security parameter. We use $[n]$ to denote the set $\{1, ..., n\}$. Composing group elements is notated additively with scaling notated multiplicatively.

Security and network model. For Trout++'s functionality, within our security proofs, we initially assume a synchronous network with authenticated point-to-point channels between the parties. In the version of the protocol that achieves IA, we instead assume a synchronous broadcast channel, which enables the parties to publish their messages (as required to agree on the transcript).

We use the standard real/ideal world paradigm to prove the security of our protocols [7]. In particular, for each of our protocols, we show that it realizes with malicious security an ideal functionality that we define later. When we prove a protocol realizes the ideal functionality with *selective (non-unanimous)* abort, it means that the ideal-world adversary can prevent each of the honest parties from obtaining their output. In contrast, when we prove security with *identifiable* abort, then either all the honest parties receive their output or all the honest parties abort while receiving an index of a corrupted party to blame.

When our protocol (with identifiable aborts) is composed with the ROAST transform [25], the resulting protocol works over an asynchronous network as defined by ROAST. Our security proofs continue to compose even with this change in network model.

2.1 Groups and Assumptions

Elliptic Curves Let $\mathsf{Gen}_{\mathsf{EC}}$ be an algorithm that, given an input 1^κ, outputs the tuple

$$\mathsf{ppEC} = (\hat{E}, q, E)$$

where $\hat{E}$ is a cyclic group with prime order q and E is a generator of $\hat{E}$. In this paper, the group operation is notated additively, and each point A in $\hat{E}$ is of the form $A = (a, \cdot)$ for some $a \in \mathbb{N}$.

Class Groups Let $\mathsf{Gen}_{\mathsf{CL}}$ be an algorithm that takes 1^κ and an odd prime order q (with length a function of κ) as an input, and outputs the tuple

$$\mathsf{ppCL} = (\bar{s}, \hat{G}, \hat{G}_q, \hat{H}, G, G_q, H).$$

The set $\hat{G}$ with the addition operation is a finite abelian group of order $q \cdot s'$, where the length of s' is a function of κ and $gcd(q,\ s') = 1$ except with probability negligible in κ. The value $\bar{s}$ is the upper bound of s'. One can decide if an element is in $\hat{G}$ in polynomial time. The set $\hat{H}$ with the addition operation is the unique cyclic subgroup of $\hat{G}$ of order q, generated by H. The group $\hat{G}_q := \{q \cdot x,\ x \in \hat{G}\}$ is the subgroup of order s' of $\hat{G}$, generated by G_q. It thus holds that $\hat{G} = \hat{G}_q \times \hat{H}$ and $G = G_q + H$ is a generator of $\hat{G}$. The discrete logarithm problem in $\hat{H}$ can be solved by a polynomial time algorithm $\mathsf{DLsolve}$, i.e., for each $x \in \mathbb{F}_q$ it holds that $x = \mathsf{DLsolve}(x \cdot H)$.

Distributions. We define two distributions $\mathcal{D}_q$ and $\mathcal{D}$. Let $\mathcal{D}_q$ (resp., $\mathcal{D}$) denote the distribution which satisfies that $\{r \cdot G_q \ : \ r \leftarrow D_q\}$ (resp., $\{r \cdot G \ : \ r \leftarrow D\}$) is close to uniformly sampling random elements in $\hat{G}_q$ (resp., $\hat{G}$). For setting concrete parameters for these distributions, see [10].

Assumptions. The unknown order assumption says that it is hard to find the order of an element in the subgroup with an unknown order.

Definition 1 (Unknown Order Assumption). *Let $\mathcal{A}$ be a PPT algorithm and consider the following experiment: The challenger runs $\mathsf{Gen}_{\mathsf{CL}}(1^\kappa)$ to obtain ppCL, $\mathcal{A}$ is given $(\mathsf{ppCL}, \mathsf{DLsolve})$ and outputs Z and e. We say that $\mathcal{A}$ wins if $Z \in \hat{G}_q$, Z is not the identity, $e \neq 0$, and $e \cdot Z$ is the identity. Denote by $\mathsf{Adv}^{ORD}_{\mathcal{A}}(\kappa)$ the success probability of $\mathcal{A}$. We say that the unknown order assumption is hard in $\hat{G}$ if for all PPT adversaries $\mathsf{Adv}^{ORD}_{\mathcal{A}}(\kappa)$ is negligible in κ.*

The hardness of the DDH assumption in the subgroup $\hat{G}_q$ is defined in the following definition.

Definition 2 (DDH assumption). *Let $\mathcal{A}$ be a PPT algorithm and consider the following experiment: The challenger runs $\mathsf{Gen}_{\mathsf{CL}}(1^\kappa)$ to obtain ppCL, samples a, b, c from $\mathcal{D}_q$ and $b \in \{0,1\}$. Then, it sets $A = a \cdot G_q$, $B = b \cdot G_q$, $C_0 = a \cdot b \cdot G_q$ and $C_1 = c \cdot G$. $\mathcal{A}$ is given $(\mathsf{ppCL}, (A, B, C_b), \mathsf{DLsolve})$ and outputs a bit b'. We say that $\mathcal{A}$ wins if $b' = b$. Denote by $\mathsf{Adv}^{DDH}_{\mathcal{A}}(\kappa)$ the success probability advantage of $\mathcal{A}$ over an adversary who simply guesses b. We say that the DDH assumption is hard in G if for all PPT adversaries $\mathsf{Adv}^{DDH}_{\mathcal{A}}(\kappa)$ is negligible in κ.*

The HSM assumption says that a PPT adversary cannot distinguish between random elements in $\hat{G}$ and $\hat{G}_q$.

Definition 3 (Hard Subgroup Membership Assumption (HSM)). *Let $\mathcal{A}$ be a PPT algorithm and consider the following experiment: The challenger runs $\mathsf{Gen}_{\mathsf{CL}}(1^\kappa)$ to obtain ppCL, samples x and x' from $\mathcal{D}_q$ and $\mathcal{D}$ respectively and $b \in \{0,1\}$. Then, it sets $Z_0 = x \cdot G_q$ and $Z_1 = x' \cdot G$. $\mathcal{A}$ is given $(\mathsf{ppCL}, Z_b, \mathsf{DLsolve})$ and outputs a bit b'. We say that $\mathcal{A}$ wins if $b' = b$. Denote by $\mathsf{Adv}^{HSM}_{\mathcal{A}}(\kappa)$ the success probability advantage of $\mathcal{A}$ over an adversary who simply guesses b. We say that the HSM assumption is hard in $\hat{G}$ if for all PPT adversaries $\mathsf{Adv}^{HSM}_{\mathcal{A}}(\kappa)$ is negligible in κ.*

The Adaptive Root Subgroup Assumption says that an adversary cannot calculate U such that $\ell \cdot U = V$ where ℓ is an odd prime and V is chosen by the adversary before it knows ℓ.

Definition 4 (Adaptive Root Subgroup Assumption (ARSA)). *Let $(\mathcal{A}_0, \mathcal{A}_1)$ be two PPT algorithms and consider the following experiment: the challenger runs $\mathsf{Gen}_{\mathsf{CL}}(1^\kappa)$ to obtain ppCL. Then, $\mathcal{A}_0$ is given ppCL to output (V, state). The challenger then chooses a random $\ell \in \mathsf{Primes}(\kappa)$. Then, $\mathcal{A}_1$ is given (ℓ, state) and outputs U. We say that $(\mathcal{A}_0, \mathcal{A}_1)$ win if $V = \ell \cdot U$ and $q \cdot U$ is not the identity. Denote by $\mathsf{Adv}^{ARSA}_{\mathcal{A}}(\kappa)$ the success probability of $(\mathcal{A}_0, \mathcal{A}_1)$. We say that the ARSA assumption is hard in $\hat{G}$ if for all PPT adversaries $\mathsf{Adv}^{ARSA}_{\mathcal{A}}(\kappa)$ is negligible in κ.*

2.2 The ECDSA signature

The ECDSA signature scheme is computed over the elliptic curve $\hat{E}$ of prime order q. Let $\mathcal{H} : \{0,1\}^* \to \mathbb{F}_q$. Then, the scheme consists of the following three algorithms:

$\underline{\mathsf{KeyGen}(1^\kappa)}$: Choose $x \in \mathbb{F}_q$. The output is a pair (x, X), where $X = x \cdot E$. The signing key is x while X is the verifying key.

$\underline{\mathsf{Sign}(x; m)}$: Given a message m, choose a random k and set $R = k \cdot E$. Let r be the x-coordinate of $R \mod q$. Output (r, s), where $s = k^{-1} \cdot (\mathcal{H}(m) + r \cdot x)$

$\underline{\mathsf{Verify}(X; m, \sigma)}$: Given a message m and a signature $\sigma = (r, s)$, check $0 < r, s < q$, set $z = ((\mathcal{H}(m) \cdot s^{-1}) \cdot E) + ((r \cdot s^{-1}) \cdot X)$ and check that z is not the point at infinity and its x-coordinate modulo q is equal to r. If it holds, output 1. Otherwise, output 0.

2.3 Shamir Secret Sharing and Feldman's VSS

The Shamir Secret sharing [26] with a threshold t consists of two algorithms: Share and $\mathsf{Reconstruct}$. In the Share algorithm, a dealer wishes to share a secret $x_0 \in \mathbb{F}_q$ to parties $P_1, ..., P_N$. In the algorithm, the dealer chooses a random polynomial $f(x) = a_0 + \sum_{j=1}^{t} a_j \cdot x^j$ of degree t, under the constraint that $f(0) = a_0 = x_0$. Then, it hands each party P_i the point $f(i)$. In the $\mathsf{Reconstruct}$ algorithm, the parties send their shares to each other and use them to compute x_0 via Lagrange interpolation. We denote by $\mathcal{L}_i^S(x)$ the Lagrange coefficient that corresponds to a subset of parties S of size $t+1$, a party $i \in S$, and an evaluation point x.

Feldman's VSS [16] enables the parties to detect cheating in the sharing process. To this end, in the sharing procedure, the dealer additionally publishes the polynomial coefficients in the exponent. That is, the dealer sends each party P_i its share $f(i)$ and also publishes $A_j = a_j \cdot E$ for $j = 0, ..., t$. Hence, each party can check that its share s_i satisfies the equation $s_i \cdot E = \sum_{j=0}^{t} i^j \cdot A_j$. Formally, we define two algorithms:

- $\underline{\left((s_1, ..., s_N), \{a_j\}_{j=0}^t, \{A_j\}_{j=0}^t\right) \leftarrow \mathsf{Vss.Share}(x_0)}$: In this algorithm, the dealer runs $\mathsf{Share}(x_0)$ to get the polynomial coefficients $a_0, ..., a_t \in \mathbb{F}_q$ (where $a_0 = x_0$), and then computes $A_j = a_j \cdot E$ for each $j \in \{0, ..., t\}$.
- $\underline{\beta \leftarrow \mathsf{Vss.Vrfy}(s_i, \{A_j\}_{j=0}^t)}$: In this algorithm, party P_i checks that $s_i \cdot E = \sum_{j=0}^{t} i^j \cdot A_j$. If the equation holds, it outputs 1. Otherwise, the output is 0.

3 Class Group Building Blocks

3.1 Class Group Encryption

We leverage a derivative of CLT encryption scheme over class groups [10]. Recall that we work over a group $\hat{G} = \hat{G}_q \times \hat{H}$, with a generator G_q for $\hat{G}_q$ and a

generator H for $\hat{H}$. The order of $\hat{G}_q$ is unknown while the order q of $\hat{H}$ is known. The discrete log problem is hard in $\hat{G}_q$ but easy in $\hat{H}$. Typically, for the secret key y^{cl} and public key $Y^{\mathsf{cl}} = y^{\mathsf{cl}} \cdot G_q$, CLT ciphertexts consist of two group elements $(C_1,\ C_2) = (\alpha \cdot G_q,\ \alpha \cdot Y^{\mathsf{cl}} + m \cdot H)$. It is possible to decrypt the ciphertext by taking $C_2 - y^{\mathsf{cl}} \cdot C_1$ and invoking the $\mathsf{DLsolve}$ algorithm over the result. In our protocol, the parties do not use y^{cl} at all and perform decryption using the randomness α. That is, decryption is carried out by taking $C_2 - \alpha \cdot Y^{\mathsf{cl}}$ and, as before, invoking the $\mathsf{DLsolve}$ over the result. Hence, we do not need to compute nor store the first element at all! This allows us to reduce both bandwidth usage and computational cost. We note this transforms CLT's public-key encryption scheme into a symmetric encryption scheme, where the randomness is needed for both encryption and decryption. As Trout++ is not a robust signing protocol itself, we already require a consistent signing set across rounds, meaning the participants performing encryption in round one are also the set performing decryption in round two (and therefore already know the randomness). This removes the need for a public-key encryption scheme, solely an additively-homomorphic symmetric encryption scheme.

Encrypt: Given a message m, choose a random $\alpha \in \mathcal{D}_q$ and output

$$\mathsf{Enc}(\alpha, m) = \alpha \cdot G_q + m \cdot H$$

Decrypt: Given a ciphertext C and a random one-time key α, output

$$\mathsf{Dec}_\alpha(C) = \mathsf{DLsolve}(C - (\alpha \cdot \hat{G}_q))$$

Note that this encryption scheme is *computationally* hiding under the HSM assumption. To see this observe that an encryption of 0 is an element in $\hat{G}_q$, whereas an encryption of any other message is an element in $\hat{G}$. Hence, encryptions of two such messages are indistinguishable if it is not possible to distinguish an element of $\hat{G}$ as an element of the subgroup $\hat{G}_q$.

As for binding, given a ciphertext C, if it is possible to open it into two pairs (α, m) and (α', m'), we have $\alpha \cdot G_q + m \cdot H = \alpha' \cdot G_q + m' \cdot H$. This implies $\alpha \cdot G_q = \alpha' \cdot G_q + (m - m') \cdot H$. However, now we have on the left side an element in $\hat{G}_q$ without any term in $\hat{H}$. Hence, the equation can hold if and only if $m - m' \equiv 0 \bmod q$. The scheme is thus perfectly binding.

Lemma 1. *The* Enc *scheme is perfectly binding. Under the HSM assumption, it is computationally hiding.*

3.2 Class Group Commitment

We now define a statistically hiding, computationally binding commitment scheme for values $\in \mathbb{F}_q$, under the assumption that G_q is a generator of $\hat{G}_q$ and the *order of* $\hat{G}_q$ *is unknown*. To commit to a value $m \in \mathbb{F}_q$, sample $\alpha \leftarrow \mathcal{D}_q$ and compute

$$\mathsf{OptCom}(\alpha, m) = \alpha \cdot (q \cdot G_q) + m \cdot G_q.$$

As the order of $\hat{G}_q$ is coprime to q except with negligible probability [11, Conjecture 5.10.1], $q \cdot G_q$ is statistically also a generator of $\hat{G}_q$. Since α is chosen from $\mathcal{D}_q$, the commitment is statistically close to uniform, hence the commitment is statistically hiding.

To show binding, assume that the committer can open the same commitment with two openings $\alpha_1,\ m_1$ and $\alpha_2,\ m_2$ such that $m_1 \neq m_2,\ \mathsf{OptCom}(\alpha_1, m_q) = \mathsf{OptCom}(\alpha_2, m_2)$. This means that $(\alpha_1 - \alpha_2) \cdot (q \cdot G_q) + (m_1 - m_2) \cdot G_q$ is the identity element and therefore $(\alpha_1 - \alpha_2) \cdot q + (m_1 - m_2)$ is the order of the group, thus breaking the unknown order assumption.

Lemma 2. OptCom *is statistically hiding. Under the unknown order assumption, it is computationally binding.*

Comparison to Trout. In Trout [13], the commitment used in the protocol was of the form $\alpha \cdot G_q + m \cdot Y$, where Y is the public key of the encryption scheme (thus its discrete logarithm with respect to the generator G_q is secret). This implies scaling two group elements, whereas with our optimized commitment, only a single scaling is required since the committer can compute $\alpha \cdot q + m$ and then perform a single scaling of G_q. Note however that the commitment in Trout was carefully designed to make the scaled decryption protocol used in the signing phase work in one round. Fortunately, we are able to construct a new scaled decryption protocol that works with the optimized commitment and symmetric encryption schemes. This will be presented in Section 6.1.

4 Zero-Knowledge Ideal Functionalities and Relations

Definition 5 (NIZKAOK protocol). *Let $\mathcal{L}$ be a NP Language and let $\mathcal{R}_\mathcal{L}$ be the associated relation. Let $\mathcal{H}$ be a random oracle. A non-interactive zero-knowledge argument of knowledge protocol consists of the following algorithms:*

- $\mathsf{NIZKAOK.prove}^R_\mathcal{H}(x, w) \to \pi$*: upon receiving a statement x and a witness w such that $(x, w) \in R$, the algorithm outputs a proof π.*
- $\mathsf{NIZKAOK.vrfy}^R_\mathcal{H}(x, \pi) \to \{0, 1\}$*: upon receiving a statement x and a proof π, the algorithm outputs* 1 *(accept) or* 0 *(reject).*

We note that the public statement x also includes any public parameters *required for defining the relation.*

The algorithms should satisfy the following requirements:

- **Completeness:** *If $(x, w) \in R$, and the algorithms are executed honestly, then the output of $\mathsf{NIZKAOK.vrfy}(x, \pi)$ is 1 with probability 1.*
- **Zero-knowledge:** *There exists a PPT simulator $\mathcal{S}$ such that for all PPT algorithms D and for all $(x, w) \in R$ there exists a negligible function* neg *such that:*

$$\mathsf{Pr}\left[D^\mathcal{H}(x, \pi) = 1 \;\middle|\; \pi \leftarrow \mathcal{S}(x)\right] - \mathsf{Pr}\left[D^\mathcal{H}(x, \pi) = 1 \;\middle|\; \pi \leftarrow \mathsf{NIZKAOK.prove}^R_\mathcal{H}(x, w)\right] \leq \mathsf{neg}(\kappa)$$

– **Knowledge extractabilty:** *There exists an extractor* Ext *such that for all PPT* $\mathcal{A}$*, there exists a negligible function* neg *such that:*

$$\Pr\left[\begin{matrix}\mathsf{NIZKAOK.vrfy}^R_{\mathcal{H}}(x,\pi^*)=1\\ \wedge(x,w^*)\notin R_{\mathcal{L}}\end{matrix}\;\middle|\;\begin{matrix}(x,\pi^*)\leftarrow\mathcal{A}^{\mathcal{H}}(\mathrm{pp})\\ w^*\leftarrow\mathsf{Ext}^{\mathcal{A}}(x,\pi^*)\end{matrix}\right]\leq\mathsf{negl}(\kappa).$$

In this work, we use the sigma protocols of [12] (which is an improvement of [27]) with the Fiat-Shamir transform to make the protocols non-interactive. These proofs have been shown to be secure under the Adaptive Root Subgroup Assumption and Hard Subgroup Membership Assumption. Zero-knowledge proofs over class groups can be complicated, since extraction traditionally requires computing an inverse in a group of unknown order. Earlier works [9] used a single-bit challenge, which required many repetitions of the proof to obtain a sufficiently small soundness error, thereby making zero-knowledge proofs for class groups slow and expensive. The works of [27, 12] were able to overcome this problem by responding over prime fields determined by the challenge, significantly improving both the proof size and the computation time while maintaining the ability to fully extract the witness. This opened the door to practical and concretely-efficient threshold signing protocols using class groups, without sacrificing notions of security, as in this work.

Relations We list several instantiations which we use in the protocol.

– Knowledge of discrete logarithms over an elliptic curve

$$R_{\mathsf{Dlog}}=\{(A_0,...,A_t,(a_0,...,a_t)):\ \forall j\in\{0,...,t\}:\ A_j=a_j\cdot E\}$$

– Knowledge of committed value

$$R_{\mathsf{ComKwlg}}=\{(B,(\beta,b)):\ B=\mathsf{OptCom}(\beta,b)\}$$

– Equality between a CL encrypted value and EC discrete log

$$R_{\mathsf{CL-EC}}=\{(A,C,(\alpha,a)):\ A=a\cdot E\ \wedge\ C=\mathsf{Enc}(\alpha,a)\}$$

– Computing affine operations over ciphertexts using committed values

$$\begin{aligned}R_{\mathsf{affCom}}=\{&(F,D,C,B,A,(\alpha,\beta,b)):\\ &F=(\beta\cdot q+b)\cdot A-\alpha\cdot B\ \wedge\ C=\mathsf{Enc}(\alpha,-)\ \ \wedge\ D=\mathsf{Com}(\beta,b)\}\end{aligned}$$

4.1 String Commitment

In our key generation protocol, we require the parties to commit to their zero-knowledge proofs before revealing them to the other parties. Hence, we assume the parties have an access to an ideal commitment functionality $\mathcal{F}_{\mathsf{com}}$, as formally defined in Functionality 1. Any UC-secure commitment can realize $\mathcal{F}_{\mathsf{com}}$. In particular, it can be trivially realized using a random oracle $\mathcal{H}$, by defining $\mathsf{Com}(x)=\mathcal{H}(\tau,x)$ where $\tau\leftarrow\{0,1\}^{\kappa}$ is random.

Functionality 1 ($\mathcal{F}_{\mathsf{com}}$ - The Commitment Functionality)

The functionality works with parties $P_1, ..., P_n$.

- Upon receiving $(\mathsf{sid}, \mathsf{commit}, x, i)$ from P_i: if sid has not been used before by P_i, store (sid, i, x) and send $(\mathsf{sid}, \mathsf{receipt}, i)$ to the other parties.
- Upon receiving $(\mathsf{sid}, \mathsf{decommit}, i)$ from P_i retrieve (sid, i, x) and send $(\mathsf{sid}, \mathsf{decommit}, i, x)$ to all parties.

5 Key Generation and Setup Protocols

In this section, we present our key generation and setup protocols. The key generation protocol is a traditional protocol to produce an ECDSA signing key in an unbiased way, resulting in each party holding a share of the key. The setup protocol produces an encryption of each party's share for use in the signing protocol. We explicitly delineate these two protocols as for key generation, any unbiased key generation protocol that outputs the public key and the shares of each party would suffice, allowing the user to choose which to compose into the resulting Trout++ Distributed Key Generation (DKG) protocol. Protocol 2 is our setup protocol that inputs the public ECDSA key and its shares, outputting the ciphertexts for each share.

Remark 1. We note the setup protocol can be executed in parallel to the first round of the signing protocol, as in our signing protocol, the first round is key-independent. This allows executing Trout++ without prior additional setup.

Protocol 2 (Trout++ Setup)

- **Public input:** $\mathsf{sid}, \mathsf{ppEC}, \mathsf{ppCL}, X, \{X_i\}_{i=1}^N$
- **Private input:** x_i
- **The protocol:** Each party P_i works as follows:
  - **Round 1:**
    1. Choose a random $\delta_i \in \mathcal{D}_q$ and compute $C_i = \mathsf{Enc}(\delta_i, x_i)$.
    2. Compute $\pi_i \leftarrow \mathsf{NIZKAOK.prove}_{\mathcal{H}}^{R_{\mathsf{CL-EC}}}(X_i, C_i, (\delta_i, x_i))$.
    3. Send $(\mathsf{sid}, C_i, \pi_i)$ to all parties.
  - **Computing the output:** Upon receiving $(\mathsf{sid}, C_k, \pi_k)$ for all $k \in [N]$:
    1. For each $k \in [N]$: compute $\beta_k \leftarrow \mathsf{NIZKAOK.vrfy}_{\mathcal{H}}^{R_{\mathsf{CL-EC}}}(X_k, C_k, \pi_k)$. If $\beta_k = 0$, send abort to all parties and halt.
    2. Output $X, \{X_k\}_{k=1}^N, \mathsf{ppCL}, \{C_k\}_{k=1}^N, \delta_i, x_i$.

Next, in Protocol 3, we describe the traditional DKG which outputs Shamir secret shares of the signing key, their corresponding commitments, and the verification key, as needed for the setup protocol.

Protocol 3 (Unbiased Key Generation)

– **Public input:** $\mathsf{sid}, \mathsf{ppEC}$
– **The protocol:** Each party P_i works as follows:
  • **Round 1:**
    1. Choose a random x_i and run $\mathsf{Vss.Share}(x_i)$ to receive $s_{i,1}, ..., s_{i,N}, \{a_{i,j}\}_{j=0}^{t}, \{A_{i,j}\}_{j=0}^{t}$ where $a_{i,0} = x_i$.
    2. Compute $\pi_i \leftarrow \mathsf{NIZKAOK.prove}_{\mathcal{H}}^{R_{\mathsf{Dlog}}}\left(\{A_{i,j}\}_{j=0}^{t}, \{a_{i,j}\}_{j=0}^{t}\right)$
    3. Send $\left(\mathsf{sid}, \mathsf{commit}, \{A_{i,j}\}_{j=0}^{t}, \pi_i, i\right)$ to $\mathcal{F}_{\mathsf{com}}$.
  • **Round 2:** Upon receiving $(\mathsf{sid}, \mathsf{receipt}, k)$ for each $k \in [N]$:
    1. Send $(\mathsf{sid}, \mathsf{decommit}, i)$ to $\mathcal{F}_{\mathsf{com}}$.
    2. Send $s_{i,k}$ to party P_k.
  • **Computing the output:** Upon receiving $(\mathsf{sid}, \mathsf{decommit}, \{A_{k,j}\}_{j=0}^{t}, \pi_k, k)$ from $\mathcal{F}_{\mathsf{com}}$ for each $k \in [N]$ and $s_{k,i}$ from each P_k:
    1. For each $k \in [N]$: compute $\beta_k \leftarrow \mathsf{NIZKAOK.vrfy}_{\mathcal{H}}^{R_{\mathsf{Dlog}}}(\{A_{k,j}\}_{j=0}^{t}, \pi_k)$. If $\beta_k = 0$, send abort to the other parties and halt.
    2. For each $k \in [N]$: run $\beta_k \leftarrow \mathsf{Vss.Vrfy}\left(s_{k,i}, \{A_{k,j}\}_{j=0}^{t}\right)$. If $\beta_k = 0$, send abort to the other parties and halt.
    3. Output $X = \sum_{k=1}^{N} A_{k,0}$, $x_i = \sum_{k=1}^{N} s_{k,i}$ and $X_i = \sum_{k=1}^{N}\left(\sum_{j=0}^{t} k^j \cdot A_{k,j}\right)$.

The above is straightforward and traditional, occurring in two rounds. However, there are other alternatives that can be used instead of the above protocol. As Trout++ already assumes an additively-homomorphic encryption scheme, one could employ the publicly-verifiable scheme detailed in [8]. Alternatively, an exponent-Verifiable Random Function (eVRF, [3]) may be used to produce an unbiased key in one round, and the same techniques can be used to construct a publicly-verifiable encryption scheme for DKG protocols ([24], [6]).

6 Signing Protocol

6.1 Optimized Scaled Decryption

We now present our improved version of Trout's Scaled Decryption protocol, achieving the same desired properties, but with our new encryption and commitment schemes (as presented in Section 3). The lines in blue are executed only when the protocol aims to acheive identifiable abort (IA); see Section 6.4 for more information.

Protocol 4 (Optimized Scaled Decryption)

The protocol is executed by a set of parties S indexed by $\{\ell_1, ..., \ell_n\}$ (s.t. $n = t{+}1 \leq N$). In the protocol, the notation P_i is used to denote the party P_{ℓ_i}.

– **Public input:** $\mathsf{ppCL}, \{B_j\}_{j=1}^{n}, \{A_j\}_{j=1}^{n}$
– **Private input of each** P_i**:** $\alpha_i, \beta_i \in \mathcal{D}_q$ and $a_i, b_i \in \mathbb{F}_q$ such that

$$A_i = \alpha_i \cdot G_q + a_i \cdot H, \qquad B_i = \mathsf{OptCom}(\beta_i,\ b_i)$$

– **The protocol:** Let

$$A = \sum_{j=1}^{n} A_j = \alpha \cdot G_q + a \cdot H, \qquad B = \sum_{j=1}^{n} B_j = \mathsf{OptCom}(\beta, b)$$

Each party P_i does the following:

1. Compute:
   - $F_i = ((\beta_i \cdot q) + b_i) \cdot A - \alpha_i \cdot B$
   - IA-mode: $\pi_i \leftarrow \mathsf{NIZKAOK.prove}_{\mathcal{H}}^{R_{\mathsf{affCom}}}(F_i, B_i, A_i, B, A, (\alpha_i, \beta_i, b_i))$.
2. Send (sid, F_i) to all parties.
   IA-mode: send $(\mathsf{sid}, F_i, \pi_i)$ to all parties.
3. Upon receiving $(\mathsf{sid}, F_j, \pi_j)$ for all $j \in [n]$.
   (a) IA-mode: For each $j \in [n]$: compute $\beta_j \leftarrow \mathsf{NIZKAOK.vrfy}_{\mathcal{H}}^{R_{\mathsf{affCom}}}(F_j, B_j, A_j, B, A, \pi_j)$.
   If $\beta_j = 0$, send abort to all parties and halt.
   (b) Compute $F = \sum_{j=1}^{n} F_j$.
   (c) $c = \mathsf{DLsolve}(F)$

– **Output:** c

Claim. If all parties act honestly, then $c = a \cdot b$.

Proof. To prove the claim, we show that $F = (a \cdot b) \cdot H$. This holds since

$$\begin{aligned} F &= \sum_{i=1}^{n} F_i = \sum_{i=1}^{n}(((\beta_i \cdot q) + b_i) \cdot A - \alpha_i \cdot B) = \\ &= \sum_{i=1}^{n}(((\beta_i \cdot q) + b_i) \cdot \alpha \cdot G_q + ((\beta_i \cdot q) + b_i) \cdot a \cdot H - \alpha_i \cdot (\beta \cdot q \cdot G_q + b \cdot G_q)) \\ &= \sum_{i=1}^{n}(((\beta_i \cdot q) + b_i) \cdot \alpha \cdot G_q + b_i \cdot a \cdot H - \alpha_i \cdot ((\beta \cdot q + b) \cdot G_q)) = a \cdot b \cdot H \end{aligned}$$

□

6.2 The Signing Protocol

In this section, we present our signing protocol. Recall that the signature is $s = k^{-1} \cdot (\mathcal{H}(m) + r \cdot x)$ where r is the x-coordinate of $k \cdot E$ reduced into the scalar field. As in previous works, we leverage Beaver's trick to compute the signature. Specifically, the parties compute $u \cdot k$ and $u \cdot (\mathcal{H}(m) + r \cdot x)$ in parallel, reveal the results, and then each party can locally compute $s = (u \cdot k)^{-1} \cdot (u \cdot (\mathcal{H}(m) + r \cdot x) = k^{-1} \cdot (\mathcal{H}(m) + r \cdot x)$.

We note that as in [13], the parties do not compute $u \cdot (\mathcal{H}(m) + r \cdot x)$, but $u \cdot (\mathcal{H}(m) \cdot r^{-1} + x)$, and only when the final signature is revealed, the parties locally multiply it with r to correct the result. Since the parties hold an encryption of x (as it must be kept secret), this optimization trades an expensive scaling of a ciphertext with a relatively cheap finite field inversion.

To perform the above in two rounds, the parties generate the nonce within the first round. In addition, each party commits to its share of u and encrypts its share of k in the first round. Thus, at the beginning of the second round, the parties can decide the nonce and compute $\mathsf{Enc}(k)$, $\mathsf{OptCom}(u)$, and $\mathsf{Enc}(\mathcal{H}(m) \cdot r^{-1} + x)$. It thus remains to call the scaled decryption protocol (Protocol 4) twice, on $\mathsf{Enc}(k)$, $\mathsf{OptCom}(u)$ and on $\mathsf{Enc}(\mathcal{H}(m) \cdot r^{-1} + x)$, $\mathsf{OptCom}(u)$. The two executions end with the parties holding $u \cdot k$ and $u \cdot (\mathcal{H}(m) \cdot r^{-1} + x)$, which suffices for computing the signature as explained above. Thus, the protocol has two overall rounds. The formal description appears in Protocol 5.

We now provide more details on the first round, where the parties produce $R = k \cdot E$. In Trout [13], this is carried out with a cryptographic primitive called an "exponent Verifiable Random Function" [3] (eVRF). Each party commits to the VRF key in the setup before computing $R_i = k_i \cdot E$ in the signing protocol, where k_i is the output of the VRF, while proving the computation was correct with respect to the commitment from the setup. Roughly speaking, since the online protocol is in fact deterministic, it is simulatable. However, the deterministic nature of the process also means that the signing set and message must be provided as input to the VRF computation, thereby preventing the desired property of a pre-signing round.

To achieve pre-signing, we replace the eVRF with a different protocol. Our starting point is the naïve protocol where each party simply chooses its share of the nonce k_i from random and sends $R_i = k_i \cdot E$ to the other parties. However, taking $R = \sum_{i=1}^{n} R_i$ means that the adversary can bias the key, since it sees the honest parties' messages before choosing its messages. To randomize the desired nonce, in a way that is not predictable by the adversary, [1] suggested to have coefficients $\rho,\ \mu$. Then, the final nonce is set to be $R = \rho \cdot \sum_{i=1}^{n} R_i + \mu \cdot E$. We sample these coefficients from random oracles where ρ is sampled in response to the entire transcript and μ is sampled in response to ρ.

This technique was also used by [23], but unlike in their work, we aim to provide a simulation-based proof. However, standard simulation typically means that the nonce is randomly chosen by the trusted party and given to the parties. This cannot work for the above technique, since the adversary can still bias the nonce. As a simple example, consider an adversary who makes queries to the random oracle with its first-round messages until the MSB of the nonce is 0 (recall that the adversary is rushing and so it receives the honest parties' messages first). Such an attack will succeed with high probability given that the adversary is allowed to make a polynomial number of queries.

To prove security of the protocol, we define an ideal functionality that enables pre-signing and randomizing of the nonce. That is, the trusted party allows the simulator to request a nonce, tweak its value and, only later, to compute the signature for a nonce that was produced earlier. Looking ahead, the ideal functionality we define in Section 6.3 has a presign and randomize command. When randomizing, the ideal-world adversary is allowed to choose the multiplicative randomization, while the trusted party chooses the additive tweak. In the proof, we show that this enables the simulator to proceed with the following strat-

egy: every time the adversary queries $\mathcal{H}_\rho$, the simulator learns a set of possible messages from the corrupted parties. The simulator then requests a new nonce from the ideal functionality. Since it now knows all messages associated with this nonce, it can program the oracle $\mathcal{H}_\mu$'s responses to return an additive randomization that is aligned with the k_i values chosen by the corrupted parties and the μ value received from the ideal functionality.

Finally, we note that [1] showed that such a randomization of a nonce which was chosen in a presigning round incurs only a constant security loss of 2^5 when compared to standard ECDSA signing (in the generic group model).

Protocol 5 (Distributed Signing)

Let $\mathcal{H}, \mathcal{H}_\mu, \mathcal{H}_\rho : \{0,1\}^* \to \mathbb{F}_q$.
The protocol is executed by a set of parties S indexed by $\{\ell_1, ..., \ell_n\}$ (s.t. $n = t+1 \leq N$). In the protocol, the notation P_i is used to denote the party P_{ℓ_i}.

- **Public input:** $\mathsf{sid}, \mathsf{ppEC}, \mathsf{ppCL}, X, \{X_j\}_{j=1}^n, \{C_j\}_{j=1}^n$.
- **Private input of each** P_i**:** x_i, δ_i such that $X_i = x_i \cdot E$ and $C_i = \mathsf{Enc}(\delta_i, x_i)$.
- **The protocol:**
  - **Round 1 (Presigning):**
    1. Choose random $k'_i, u_i \in \mathbb{F}_q$ and $\alpha_i, \beta_i \in \mathcal{D}_q$ and compute:
       - $R'_i = k'_i \cdot E$.
       - $K'_i = \mathsf{Enc}(\alpha_i, k'_i) = \alpha_i \cdot G_q + k'_i \cdot H$.
       - $U_i = \mathsf{OptCom}(\beta_i, u_i) = \beta_i \cdot q \cdot G_q + u_i \cdot G_q$.
    2. Compute:
       (a) $\pi_i^{R_{\mathsf{CL-EC}}} \leftarrow \mathsf{NIZKAOK.prove}_{\mathcal{H}}^{R_{\mathsf{CL-EC}}}(R'_i, K'_i, (\alpha_i, k'_i))$.
       (b) $\pi_i^{R_{\mathsf{ComKwlg}}} \leftarrow \mathsf{NIZKAOK.prove}_{\mathcal{H}}^{R_{\mathsf{ComKwlg}}}(U_i, (\beta_i, u_i))$.
    3. Send $\mathsf{msg}_i = (\mathsf{sid}, R'_i, K'_i, U_i, \pi_i^{R_{\mathsf{CL-EC}}}, \pi_i^{R_{\mathsf{ComKwlg}}})$ to all parties.
  - **Round 2 (Signing):**
    1. Upon receiving:
       - msg_j for all $j \in [n]$
       - a message m to sign

       compute:
       - For each $j \in [n]$: $\beta_j \leftarrow \mathsf{NIZKAOK.vrfy}_{\mathcal{H}}^{R_{\mathsf{CL-EC}}}(R'_j, K'_j, \pi_j^{R_{\mathsf{CL-EC}}})$. If $\beta_j = 0$, send abort to all parties and halt.
       - For each $j \in [n]$: $\beta_j \leftarrow \mathsf{NIZKAOK.vrfy}_{\mathcal{H}}^{R_{\mathsf{ComKwlg}}}(U_j, \pi_j^{R_{\mathsf{ComKwlg}}})$. If $\beta_j = 0$, send abort to all parties and halt.
       - $R' = \sum_{j=1}^n R'_j$.
       - $\rho = \mathcal{H}_\rho(m, X, \{\mathsf{msg}_j\}_{j=1}^n)$.
       - $\mu = \mathcal{H}_\mu(\mathsf{sid}, \rho)$.
       - $R = \rho \cdot R' + \mu \cdot E$. Let r be the x-coordinate of R reduced into $\mathbb{F}_q$.
       - $k_j = k'_j$ for all $j \neq 1$ and $k_1 = k'_1 + (\rho^{-1} \cdot \mu)$.
       - $K_j = K'_j$ for all $j \neq 1$ and $K_1 = K'_1 + (\rho^{-1} \cdot \mu) \cdot H$.
       - $x'_j = \mathcal{L}_{\ell_i}^S(0) \cdot x_j$ for all $j \neq 1$ and $x'_1 = \mathcal{L}_{\ell_i}^S(0) \cdot x'_1 + (r^{-1} \cdot \mathcal{H}(m))$.
       - $Z_j = \mathcal{L}_{\ell_i}^S(0) \cdot C_j$ for all $j \neq 1$ and $Z_1 = \mathcal{L}_{\ell_i}^S(0) \cdot C_1 + (r^{-1} \cdot \mathcal{H}(m)) \cdot H$.
    2. Run Protocol 4 on public input $(\mathsf{sid}, \mathsf{ppCL}, \{U_j\}_{j=1}^n, \{Z_j\}_{j=1}^n)$ and private input $(\mathcal{L}_i^S(0) \cdot \delta_i, \beta_i, x'_i, b_i)$ to receive $u \cdot ((r^{-1} \cdot \mathcal{H}(m)) + x)$.
    3. Run Protocol 4 on $(\mathsf{sid}, \mathsf{ppCL}, \{U_j\}_{j=1}^n, \{K_j\}_{j=1}^n)$ and private input $(\alpha_i, \beta_i, k_i, b_i)$ to receive $u \cdot (k + \rho^{-1} \cdot \mu)$.

4. Compute $s = ((u \cdot (k + \rho^{-1} \cdot \mu)) \cdot \rho)^{-1} \cdot (u \cdot ((r^{-1} \cdot \mathcal{H}(m)) + x)) \cdot r$.
5. In IA-mode, this instruction is omitted.
   Run $\mathsf{Verify}(X; m, (r, s))$ to receive β. If $\beta = 0$, then output $\perp$ and halt.

– **Output:** (r, s)

6.3 Proof of Security

Ideal signing functionality. In Functionality 6, we define an ideal functionality for distributed ECDSA signing. Note that it receives four commands: KeyGen, preSign, Randomize and Sign. In the preSign step, a new nonce $R' = k' \cdot E$ is chosen and handed to the ideal-world adversary. However, once the message is known, the nonce is randomized using the Randomize command, which sets $R = \rho \cdot R' + \mu \cdot E$, where ρ is chosen by the adversary and μ is chosen randomly by the functionality. Finally, upon receiving the Sign command, the functionality computes the signature and hands it to the adversary. We note that the functionality allows the simulator to return a different valid signature. This captures also the version of ECDSA which does not have the strongly-unforgeable property. Clearly, when the strongly-unforgeable version of ECDSA is used, the adversary cannot produce a different signature for m. As noted above, ECDSA with presigning and such a randomization have been shown in [1] to incur only a small constant loss.

Security. In the following theorem, we state the security of our protocol. The proof can be found in Appendix A.

Theorem 7. *Let $0 \leq t \leq n-1$ be a security threshold. Then, under the hardness of the HSM assumption and the DDH assumption in $\hat{G}_q$, and assuming $\mathcal{H}_\mu$ and $\mathcal{H}_\rho$ are modeled as a random oracle, Protocol 2+3 (distributed key generation) and Protocol 5 (distributed signing) securely compute $\mathcal{F}_{\mathsf{ECDSA}}$ with selective abort in the presence of malicious adversaries controlling up to t malicious parties.*

6.4 Extensions

Achieving Identifiable Abort We can easily transform Protocol 2 (setup) and Protocol 5 (signing) to achieve the stronger property of identifiable abort (IA). This is done when the instructions for IA-Mode (in blue) are executed and all messages are sent over a broadcast channel. In this case, as all messages are public and accompanied by zero-knowledge proofs, anyone observing the transcript is able to identify misbehaving participants simply by verifying the proofs.

Achieving identifiable abort for Protocol 3 (key generation) is not immediate however as it requires privately sending Shamir secret shares to the other participants. To overcome this, it is possible to use a publicly-verifiable VSS instead of the described VSS, which we leave the description of to existing works on this topic (as we do not impose any additional or special requirements and are designed to be interoperable with any Shamir-secret-shared signing key).

Functionality 6 ($\mathcal{F}_{\mathsf{ECDSA}}$ - Distributed ECDSA)

- **Parties:** $P_1, \ldots, P_N$ and the ideal-world adversary $\mathcal{S}$.
- **Parameters:** $0 \leq t < N \in \mathbb{N}$ and $\kappa \in \mathbb{N}$.
- **Operation**]
  - Upon receiving $(\mathsf{sid}, \mathsf{KeyGen})$ from all n parties:
    1. Let $(x, X) \leftarrow \mathsf{KeyGen}(1^\kappa)$.
    2. Store (sid, x).
    3. Send (sid, X) to all parties.
    4. Ignore future calls withthe same sid.
  - Upon receiving $(\mathsf{sid}, \mathsf{preSign})$ from $t+1$ parties:
    1. Retrieve (sid, x).
    2. Choose a random $k' \in \mathbb{F}_q$ and set $R' = k' \cdot E$.
    3. Send (sid, R') to $\mathcal{S}$ and store (sid, x, k').
  - Upon receiving $(\mathsf{sid}, \mathsf{Randomize}, \rho, m)$ from $t+1$ parties:
    1. Retrieve (sid, x, k').
    2. Choose $\mu \in \mathbb{F}_q$ and compute $k = \rho \cdot k' + \mu$
    3. Set $R = k \cdot E$. Let r be the x-coordinate of $R \bmod q$.
    4. Send (sid, R, μ) to $\mathcal{S}$ and store $(\mathsf{sid}, x, r, k, m)$.
    5. Ignore any future call with same sid.
  - Upon receiving $(\mathsf{sid}, \mathsf{Sign}, m, r)$ from $t+1$ parties:
    1. Retrieve $(\mathsf{sid}, x, r, k, m)$.
    2. Compute $s = k^{-1} \cdot (\mathcal{H}(m) + r \cdot x)$
    3. Send $(\mathsf{sid}, (r, s))$ to $\mathcal{S}$.
    4. Wait to receive back (r, s') from $\mathcal{S}$. If $\mathsf{Verify}(X; m, (r, s')) = 1$ output (r, s') to all parties. Otherwise, output abort.

Achieving Robustness even in the Asynchronous Setting ROAST [25] is an efficient transformation which lifts certain threshold signing protocols to work over an asynchronous network and guarantee delivery of the output (a valid signature). The underlying protocol must meet three requirements, which our protocol satisfies: (i) The protocol is semi-interactive: two rounds, where the first round does not require the message being signed nor the signing set as an input. Our first round is key-, signing-set-, and message- independent. (ii) Support for identifiable aborts: a party with the transcript can identify misbehaving parties. Our protocol offers this so long as the protocol is ran in IA-mode. (iii) The signing protocol is unforgeable even during concurrent executions, which our simulation-based proofs and method of presigning ensure. Hence, we can apply the ROAST transformation to Trout++ and obtain a robust protocol, even over an asynchronous network.

	Variable-Time	Constant-Time	Bytes
Presign	103ms	3,044ms	1,350
Sign	303ms	9,025ms	2,505
Sign without Identifiable Aborts	54ms	1,492ms	450

Table 2. The median local computation time for each participant in our protocol, excluding the verification and aggregation of the other participants' messages (which is independent of the witness), and the median size of each participant's message in bytes. These values are constant to the security parameter and independent of the set size.

	2-of-*	34-of-*	67-of-*
With Identifiable Aborts (Variable-Time)	366ms	1,608ms	2,599ms
Without Identifiable Aborts (Variable-Time)	195ms	633ms	1,085ms
With Identifiable Aborts (Constant-Time)	12,129ms	13,271ms	14,449ms
Without Identifiable Aborts (Constant-Time)	4,574ms	5,012ms	5,464ms

Table 3. The benchmarks of the signing protocol in its entirety, including the time to participate in each round and the time to verify and aggregate messages. The columns represent various set sizes with the median time taken per-participant. The constant-time variants are only constant in time with regards to the witness.

7 Implementation

Trout [13] was published with a constant-time library for class group cryptography in order to defend against side-channel attacks. We have significantly optimized this library and written a implementation of Trout++ from scratch[6].

Between the the increased efficiency of Trout++ and further work on the implementation, we have continued to close the gap between the hardened constant-time implementation and the protocol itself, making it more and more considerable for deployment.

Practical Optimizations. Like Trout, the implemented zero-knowledge proofs are derived from the methodology of [12]. We introduce a practical optimization for the elements in the response to the challenge denoted D. The CLT encryption scheme over class groups [10] requires a fundamental discriminant Δ_k and a discriminant Δ_q. Between these two class groups is a surjection π and injection ψ such that $q \cdot \psi(\pi(D)) = q \cdot D$. Instead of responding to the challenge with the element D of the class group with discriminant Δ_q, we respond with the element $\pi(D)$ of the class group with discriminant Δ_k, which has an encoding approximately 25% smaller and is allowed as the verification equation scales the element by q. This also restricts from having q valid responses to the challenge to just one (under the computational assumptions already present), ensuring the response is canonical and without malleability. We are also able to take advantage of these maps with our optimized commitments as they lack any terms from $\hat{H}$.

[6] The code is available here: https://github.com/kayabaNerve/trout

Like Trout's, our implementation compresses binary quadratic forms as detailed in [14]. The compression algorithm requires, for numbers $b,\ a',\ g,\ f$ where $g \leq f,\ b \leq g \cdot a' \leq \mathsf{lcm}(f,\ a')$, sending $b \mod f,\ b \mod a'$ before using the Chinese Remainder Theorem to reconstruct b (with the exceptional case $b = a$ handled specially). Unfortunately, the algorithm only has a statistical bound for its termination due to having to find a suitable value for f which satisfies the requirements. We introduce a slight variant of the scheme where we instead send $\lfloor b/a' \rfloor,\ b \mod a'$, effectively representing b as its Euclidean division by a'. This avoids the f variable entirely, achieving a perfect (not statistical) bound on termination and the size of the resulting encoding, not to mention a simpler scheme overall.

Experimental Results. In Table 2 and Table 3, we present the performance of our implementation. All of the benchmarks were obtained on a Mac mini with an Apple M2 Pro CPU, each participant only using a single-thread to participate in the signing protocol. All tests were ran with a 128-bit computational security parameter and a 1827-bit fundamental discriminant. When sampling from distributions $\mathcal{D}_q,\ \mathcal{D}$, and when performing batch verification, a 128-bit statistical security parameter was used. For the variable-time invocations, [4] was used for the implementation of class group cryptography.

Comparison to Trout. The resulting implementation is almost twice as fast as Trout's implementation for participants with variable-time provers and approximately *ten times as fast* for participants with constant-time provers. Additionally, the median upload per-participant in the bulletin board model was reduced from 6,345 bytes to 3,855 bytes with identifiable aborts and from 3,902 bytes to 1,800 bytes without identifiable aborts. Some of this benefit is from the Trout++ protocol while some is the result of the improvements to the implementation. For a more direct comparison of the protocols themselves, we recommend comparing the variable-time column from Table 2 to Trout's published performance numbers [13] for proving the round one and round two proofs in variable time. This eliminates any differences in the constant-time implementation by excluding it entirely, allowing focusing on the average case of the protocols themselves instead.

8 Key Derivation

Performing distributed key generation (DKG) or similar setup protocols is often significantly more expensive than executing the signing protocol itself. While this is reasonable—since setup is done once whereas signing may occur many times—there are situations where an entity wants or needs multiple signing keys. For example, distinct signing keys can be used to embed context into the signature (when the signed messages cannot effectively encode the context), or in privacy-preserving settings where multiple identities (public keys) are desirable.

Both motivations are illustrated by Bitcoin Improvement Proposal 44[7], a Bitcoin standard in which public keys encode wallet structure, distinguishing

[7] https://github.com/bitcoin/bips/blob/master/bip-0044.mediawiki

accounts and whether funds originate from internal "change" outputs or external sources. Addresses within an account are further indexed, enabling many addresses per account. In practice, this leads to the recommendation of using each address only once, generating a fresh address for every interaction to improve privacy by reducing linkability.

When the user is a group performing DKG and signing protocols, the desire for multiple public keys persists, although traditional approaches to derive keys generally do not directly apply. In centralized settings, keys are typically derived from a single root private key, requiring only a single setup. In contrast, distributed settings generally rely on secret-shared keys, practically limiting derivations to those supported homomorphically.

Groth and Shoup [20] introduced affine ("homogenous") key derivation, where signing is performed with respect to keys of the form $X + d \cdot X'$ for $X,\ X' \in \hat{E}$ and a derivation d. They show that allowing such derivations gives an adversary only negligible additional advantage compared to standard ECDSA. Next, we show how it can be composed with Trout++.

Affine Derivation. The Trout++ Setup may be performed twice, once for $\mathsf{ppEC},\ \mathsf{ppCL},\ X',\ \{X'_i\}_{i=1}^N$ with private input x'_i and once for $\mathsf{ppEC},\ \mathsf{ppCL},\ X'',\ \{X''_i\}_{i=1}^N$ with private input x''_i. As $\mathsf{ppEC},\ \mathsf{ppCL}$ are consistent across inputs, they will be consistent across the outputs, with $X', X'', \{X'_i\}_{i=1}^N,\ \{X''_i\}_{i=1}^N \in \hat{E}$ and $\{C'_i\}_{i=1}^N, \{C''_i\}_{i=1}^N \in \hat{G}$. As the derivation process is linear, it is possible to obtain an output in the same class as the output from a single invocation of the Trout++ Setup, by computing for a derivation d: $X = X' + d \cdot X'',\ \{X_i\}_{i=1}^N = X'_i + d \cdot X''_i,\ \{C_i\}_{i=1}^N = C'_i + d \cdot C''_i,\ \delta_i = \delta'_i + d \cdot \delta''_i,\ x_i = x'_i + d \cdot x''_i$.

9 Open Questions

We achieve a robust, asynchronous signing protocol (when composed with ROAST) with approximately a worst-case $\mathcal{O}(n)$ round complexity and $\mathcal{O}(n^4 + n^4 \cdot \kappa)$ communication complexity, even without a honest (super)majority. We leave such protocols with a constant round complexity even in the worst case, and/or $\mathcal{O}(n^4 + n^3 \cdot \kappa)$ communication complexity (such as when the coordinator in ROAST may aggregate preprocesses from size $n \cdot \kappa$ to just κ), as open problems to solve.

We present a proof of Protocol 6.1, Optimized Scaled Decryption, premised on the Decisional Diffie-Hellman problem over the class group while relying on the Hard Subgroup Membership assumption for our ciphertexts. It would be interesting to find a protocol either solely premised on the DDH problem or solely on the HSM assumption, to reduce the amount of assumptions present. Similarly, we would be interested in a protocol which achieves adaptive security, where the existing works on adaptively-secure variants of FROST may present paths forward (as both are two-round protocols where the first round is independent of any pre-existing witness).

Our constant-time implementation contains notable gaps between the average amount of iterations and the proven bounds on termination for various

algorithms. A frequent comment, without associated proofs, is that the output of the NUCOMP algorithm for composing two binary quadratic form is almost reduced. A proven bound on the necessary amount of reductions, which is tighter than the current bound of an amount of iterations approximately equal to the bit-length of the square root of the discriminant, would drastically reduce the time the constant-time prover takes.

References

1. Adjedj, M., Blokh, C., Couteau, G., Galansky, A., Joux, A., Makriyannis, N.: Two-round 2pc ECDSA at the cost of 1 OLE: applications to embedded cryptocurrency wallets. In: Public-Key Cryptography - PKC. pp. 355–386 (2026)
2. Boneh, D., Gennaro, R., Goldfeder, S.: Using level-1 homomorphic encryption to improve threshold DSA signatures for bitcoin wallet security. In: LATINCRYPT. pp. 352–377 (2017)
3. Boneh, D., Haitner, I., Lindell, Y., Segev, G.: Exponent-vrfs and their applications. In: Advances in Cryptology - EUROCRYPT. pp. 195–224 (2025)
4. Bouvier, C., Castagnos, G., Imbert, L., Laguillaumie, F.: I want to ride my bicycl: Bicycl implements cryptography in class groups. IACR Cryptol. ePrint Arch., paper 2022/1466 (2022)
5. Boyle, E., Devadas, L., Servan-Schreiber, S.: Non-interactive distributed point functions. In: Public-Key Cryptography - PKC. pp. 3–35 (2025)
6. Bünz, B., Choi, K., Komlo, C.: Golden: Lightweight non-interactive distributed key generation. IACR Cryptol. ePrint Arch. (2025), `https://eprint.iacr.org/2025/1924`
7. Canetti, R.: Security and composition of multiparty cryptographic protocols. J. Cryptol. **13**(1), 143–202 (2000)
8. Cascudo, I., David, B.: Publicly verifiable secret sharing over class groups and applications to DKG and YOSO. In: Advances in Cryptology - EUROCRYPT. pp. 216–248 (2024)
9. Castagnos, G., Catalano, D., Laguillaumie, F., Savasta, F., Tucker, I.: Two-party ECDSA from hash proof systems and efficient instantiations. In: Advances in Cryptology - CRYPTO. pp. 191–221 (2019)
10. Castagnos, G., Laguillaumie, F., Tucker, I.: Practical fully secure unrestricted inner product functional encryption modulo p. In: ASIACRYPT. pp. 733–764 (2018)
11. Cohen, H.: A Course in Computational Algebraic Number Theory. Springer Berlin, Heidelberg (1996)
12. Cui, H., Chan, K.Y., Yuen, T.H., Kang, X., Chu, C.: Bandwidth-efficient zero-knowledge proofs for threshold ECDSA. Comput. J. pp. 1265–1278 (2024)
13. Dahari-Garbian, H., Nof, A., Parker, L.: Trout: Two-round threshold ECDSA from class groups. In: ACM CCS. pp. 380–393 (2025)
14. Dobson, S., Galbraith, S.D., , Smith, B.: Trustless unknown-order groups. IACR Cryptol. ePrint Arch., paper 2020/196 (2020)
15. Doerner, J., Kondi, Y., Lee, E., Shelat, A.: Threshold ECDSA from ECDSA assumptions: The multiparty case. In: IEEE Symposium on Security and Privacy, SP 2019. pp. 1051–1066 (2019)
16. Feldman, P.: A practical scheme for non-interactive verifiable secret sharing. In: Annual Symposium on Foundations of Computer Science. pp. 427–437 (1987)

17. Gennaro, R., Goldfeder, S.: Fast multiparty threshold ECDSA with fast trustless setup. In: ACM CCS. pp. 1179–1194 (2018)
18. Gennaro, R., Goldfeder, S., Narayanan, A.: Threshold-optimal DSA/ECDSA signatures and an application to bitcoin wallet security. In: Applied Cryptography and Network Security, ACNS. pp. 156–174 (2016)
19. Gouvêa, C.P.L., Komlo, C.: Re-randomized FROST. IACR Cryptol. ePrint Arch. (2024), `https://eprint.iacr.org/2024/436`
20. Groth, J., Shoup, V.: On the security of ECDSA with additive key derivation and presignatures. In: Advances in Cryptology - EUROCRYPT. pp. 365–396 (2022)
21. Komlo, C., Goldberg, I.: FROST: flexible round-optimized schnorr threshold signatures. In: Selected Areas in Cryptography - SAC. pp. 34–65 (2020)
22. Lindell, Y., Nof, A.: Fast secure multiparty ECDSA with practical distributed key generation and applications to cryptocurrency custody. In: ACM CCS. pp. 1837–1854 (2018)
23. Lyu, Y., Li, Z., Zhou, H., Deng, X.: Threshold ECDSA in two rounds. In: ACM CCS. pp. 2937–2950 (2025)
24. Parker, L.: Robust Threshold eVRF DKG. Unpublished (2024), `https://github.com/serai-dex/serai/tree/next/audits/crypto/dkg/evrf`
25. Ruffing, T., Ronge, V., Jin, E., Schneider-Bensch, J., Schröder, D.: ROAST: robust asynchronous schnorr threshold signatures. In: ACM CCS. pp. 2551–2564 (2022)
26. Shamir, A.: How to share a secret. Commun. ACM **22**(11), 612–613 (1979)
27. Yuen, T.H., Cui, H., Xie, X.: Compact zero-knowledge proofs for threshold ECDSA with trustless setup. In: Public-Key Cryptography - PKC. pp. 481–511 (2021)

A Proof of Theorem 7

Proof. Denote by C the set of corrupted parties, and let $\mathcal{A}$ be the real-world adversary controlling C. We denote by J the set of honest parties. We describe a simulator to the protocol, and prove that the simulated and real execution are indistinguishable. In the simulation, the simulator plays the role of the random oracle and the honest parties.

Unbiased Key Generation. The protocol is identical to the key generation in [13]. The simulation works exactly the same as in [13], and therefore we omit the details. At the end of the simulation, the simulator holds X, $\{X_i\}_{i=1}^{N}$ and the private share x_i of each corrupted party P_i.

Setup. The simulation works as follows:

1. Simulating the honest parties' messages:
   (a) For each $j \in J$,the simulator chooses random δ_j and computes $C_i = \mathsf{Enc}(\delta_i, 0)$.
   (b) For each $j \in J$, the simulator runs the zero-knowledge simulator for the relation $R_{R_{\mathsf{CL-EC}}}$ on X_j, C_i to receive a proof π_j.
   (c) For each $j \in j$, the simulator sends $(\mathsf{sid}, C_j, \pi_j)$ to the adversary $\mathcal{A}$.
2. Upon receiving the message $(\mathsf{sid}, C_i, \pi_i)$ for each $i \in C$ from $\mathcal{A}$, the simulator:

   (a) Runs $\mathsf{NIZKAOK.vrfy}_{\mathcal{H}}^{R_{R_{\mathsf{CL-EC}}}}(X_i, C_i, \pi_i)$ for each $i \in C$, to receive β_i.
   If for some $i \in C$, $\beta_i = 0$, then the simulator sends abort to $\mathcal{F}_{\mathsf{ECDSA}}$ and simulates the honest parties abort in the real-world execution.
   (b) Runs the zero-knowledge extractor to extract δ_i for each $i \in C$.
3. The simulator outputs C_i for each i$\in [n]$ and δ_i for each $i \in C$

Signing. First, we construct a sub-simulator $\mathcal{S}_{\mathsf{sd}}$ for the scaled decryption protocol.

– $\mathcal{S}_{\mathsf{sd}}$ works as follows:
  • **Inputs:**
    ∗ $\{A_j\}_{j=1}^n$, $\{B_j\}_{j=1}^n$.
    ∗ For each $i \in C$: $\alpha_i, \beta_i, b_i, \in D_q$ such that $A_i = \mathsf{Enc}(\alpha_i, -)$ and $B_i = \mathsf{OptCom}(\beta_i, b_i)$.
    ∗ $a \cdot b$
  • **Simulation:**
    1. Let $A = \sum_{j=1}^n A_j$ and $B = \sum_{j=1}^n B_j$.
    For each $i \in C$: $\mathcal{S}_{\mathsf{sd}}$ computes $F_i = ((\beta_i \cdot q) + b_i) \cdot A - \alpha_i \cdot B$.
    2. For each $j \in J$: $\mathcal{S}_{\mathsf{sd}}$ chooses random F_j in $\hat{G}$, under the constraint that $\sum_{i \in C} F_i + \sum_{j \in J} F_j = a \cdot b \cdot H$.
    3. $\mathcal{S}_{\mathsf{sd}}$ sends F_j for each $j \in J$ to $\mathcal{A}$.
    4. Upon receiving $F_{i,j}$ for each $i \in C$ intended to P_j from $\mathcal{A}$, $\mathcal{S}_{\mathsf{sd}}$ computes for each honest party P_j: $C_j = \sum_{i \in C} F_{i,j} + \sum_{k \in [n] \setminus C} F_k$ and run $\mathsf{DLsolve}(C_j)$ to receive c_j.

Next, the simulator $\mathcal{S}$ for the signing protocol. Note that the number of parties participating in the online protocol is n. We abuse notation and denote again the set of corrupted parties by C and the set of honest parties by J (although it may include only a subset of all honest or corrupted parties). Let $j^* \in J$ be a fixed index of some honest party. The simulator $\mathcal{S}$ works as follows:

– **Input:** sid, $\{X_i\}_{i=1}^n$, $\{C_i\}_{i=1}^n$, $\{x_i\}_{i \in C}$, $\{\delta_i\}_{i \in C}$.
– **Simulating the first round:**
  1. The simulator $\mathcal{S}$ sends $\mathsf{preSign}$ to $\mathcal{F}_{\mathsf{ECDSA}}$, to receive back R'.
  2. For each $j \in J$, the simulator $\mathcal{S}$:
     (a) Chooses random k'_j and compute $R'_j = k'_j \cdot R$
     (b) For each $j \neq j^*$, it chooses random α_j and computes $K'_j = \mathsf{Enc}(\alpha_j, 0)$.
     For P_{j^*}, it chooses random $K'_{j^*} \in \hat{G}$
     (c) For each $j \neq j^*$, it chooses random β_j, u_j and computes $U_j = \mathsf{OptCom}(\beta_j, u_j)$ as in the protocol's instructions
     For P_{j^*}, it chooses random $U_{j^*} \in \hat{G}_q$
     (d) Runs the zero-knowledge simulator $\mathcal{S}$ for $R_{R_{\mathsf{CL-EC}}}$ over (K'_j, R'_j) to obtain $\pi_j^{R_{\mathsf{CL-EC}}}$.
     (e) For each $j \neq j^*$, it runs $\pi_j^{R_{\mathsf{ClKwlg}}} \leftarrow \mathsf{NIZKAOK.prove}_{\mathcal{H}}^{R_{\mathsf{ClKwlg}}}(U_j, (\beta_j, u_j))$
     For P_{j^*}, it runs the zero-knowledge simulator $\mathcal{S}$ for R_{ComKwlg} over (U_j) to obtain $\pi_j^{R_{\mathsf{CL-EC}}}$.

(f) Sends $\mathsf{msg}_j = (\mathsf{sid}, R'_j, K'_j, U_j, \pi_j^{R_{\mathsf{CL-EC}}}, \pi_j^{R_{\mathsf{ClKwlg}}})$ to $\mathcal{A}$.

3. Simulating calls to $\mathcal{H}_\mu$ and $\mathcal{H}_\rho$:
   - Every time $\mathcal{A}$ sends a new query to $\mathcal{H}_\rho$, the simulator $\mathcal{S}$ works by the following instructions:
     (a) Parse the query as $(m, X, \{\mathsf{msg}\}_{j=1}^n)$. If the same query has been issued before, ignore the call.
     (b) Choose a random $\rho' \in \mathbb{F}_q$ and hand it back to $\mathcal{A}$.
     (c) For each $i \in C$, parse msg_i as $(R'_i, K'_i, U'_i, \pi_i^{R_{\mathsf{CL-EC}}}, \pi_i^{R_{\mathsf{ClKwlg}}})$.
     (d) If (i) X is incorret, or, (ii) for some $j \in [n] \setminus C$, msg_j is different from what was sent by $\mathcal{S}$, or (iii) for some $i \in C$, $\mathsf{NIZKAOK.vrfy}_{\mathcal{H}}^{R_{\mathsf{CL-EC}}}(R'_i, K'_i, \pi_i^{R_{\mathsf{CL-EC}}}) \to 0$ or $\mathsf{NIZKAOK.vrfy}_{\mathcal{H}}^{R_{\mathsf{ClKwlg}}}(U_i, \pi_i^{R_{\mathsf{ClKwlg}}}) \to 0$, choose a random μ' and store $(\mathsf{sid}, m, X, \{\mathsf{msg}\}_{j=1}^n, \rho', \mu')$.
     (e) Otherwise:
        i. Set $\rho = \rho' \cdot \sum_{j \in J} k_j$ and send $(\mathsf{sid}, \mathsf{Randomize}, \rho, m)$ to $\mathcal{F}_{\mathsf{ECDSA}}$ to receive back μ.
        ii. For each $i \in C$, run the knowledge extractor for the relation $R_{\mathsf{CL-EC}}$ on $(R'_i, K'_i, \pi_i^{R_{\mathsf{CL-EC}}})$ to receive α_i, k'_i.
        iii. For each $i \in C$, run the knowledge extractor for the relation R_{ClKwlg} on $(U_i, \pi_i^{R_{\mathsf{ClKwlg}}})$ to receive β_i, u_i.
        iv. Set $\mu' = \mu - \rho' \sum_{i \in C} k_i$
        v. Store $(\mathsf{sid}, m, X, \{\mathsf{msg}\}_{j=1}^n, \rho', \mu')$
   - Every time $\mathcal{A}$ sends a new query (sid, ρ') to $\mathcal{H}_\mu$, the simulator $\mathcal{S}$ works by the following instructions:
     (a) If there exists a record for the same sid that contains ρ', then retrieve μ' from that record and send it to $\mathcal{A}$.
     Otherwise, choose a random μ' and send it to $\mathcal{A}$.
4. Upon receiving msg_i for each $i \in C$ from $\mathcal{A}$, the simulator $\mathcal{S}$ works as follows:
   (a) Run $\mathsf{NIZKAOK.vrfy}_{\mathcal{H}}^{R_{\mathsf{CL-EC}}}(R'_i, K'_i, \pi_i^{R_{\mathsf{CL-EC}}})$ to receive β.
   If $\beta = 0$, then send abort to $\mathcal{F}_{\mathsf{ECDSA}}$ and simulate the honest parties aborting in the real execution.
   (b) Run $\mathsf{NIZKAOK.vrfy}_{\mathcal{H}}^{R_{\mathsf{ClKwlg}}}(U_i, \pi_i^{R_{\mathsf{ClKwlg}}})$ to receive β.
   If $\beta = 0$, then send abort to $\mathcal{F}_{\mathsf{ECDSA}}$ and simulate the honest parties aborting in the real execution.

– **Simulating the second round:** Upon receiving a message m to sign:

1. The simulator $\mathcal{S}$ retrieves the record $(\mathsf{sid}, m, X, \{\mathsf{msg}\}_{j=1}^n, \rho', \mu')$.
2. The simulator $\mathcal{S}$ computes $R = \rho' \cdot \sum_{i=1}^n R'_i + \mu'$. Let r be the x-coordinate of R reduced into $\mathbb{F}_q$.
3. For each $i \in [n]$, the simulator $\mathcal{S}$ computes:
   - $K_i = K'_i$ for all $i \neq 1$ and $K_1 = K'_1 + (\rho^{-1} \cdot \mu) \cdot H$
   - $Z_i = \mathcal{L}_{\ell_i}^S(0) \cdot C_i$ for all $i \neq 1$ and $Z_1 = (\mathcal{H}(m) \cdot (r^{-1}) \cdot H) + \mathcal{L}_{\ell_i}^S(0) \cdot C_1$.
   - for each $i \neq j^*$: $\delta'_i = \mathcal{L}_{\ell_i}^S(0) \cdot \delta_i$
4. The Simulator $\mathcal{S}$ sends $(\mathsf{sid}, \mathsf{Sign}, m, r)$ to $\mathcal{F}_{\mathsf{ECDSA}}$ and then waits to receive $(\mathsf{sid}, (r, s))$ from $\mathcal{F}_{\mathsf{ECDSA}}$.

5. $\mathcal{S}$ chooses a random value in $\mathbb{F}_q$ for $v = k + (\rho^{-1} \cdot \mu) \cdot u$ and sets w such that it satisfies the equation $w \cdot r = s \cdot (v \cdot \rho)$.
6. Simulating the scaled decryption of v:
   The simulator $\mathcal{S}$ runs $\mathcal{S}_{\mathsf{sd}}$ on
   $$(\mathsf{sid}, \mathsf{ppCL}, \{K_i\}_{i=1}^n, \{U_i\}_{i=1}^n, \{\alpha_i, \beta_i, u_i\}_{i \in C}, v)$$
   to receive v_j for each $j \in J$.
7. Simulating the scaled decryption of $w = u \cdot (H(m) \cdot r^{-1} + x)$:
   The simulator $\mathcal{S}$ runs $\mathcal{S}_{\mathsf{sd}}$ on
   $$(\mathsf{sid}, \mathsf{ppCL}, \{Z_i\}_{i=1}^n, \{U_i\}_{i=1}^n, \{\delta_i', \beta_i, u_i\}_{i \in C}, w)$$
   to receive w_j for each $j \in J$.
8. For each $j \in J$, the simulator $\mathcal{S}$ computes $s_j = (v_j \cdot \rho)^{-1} \cdot (w_j \cdot r)$ and runs $\mathsf{Verify}(X; m, (r, s_j))$ to receive β_j.
   If $\beta_j = 0$, it sends abort_j to $\mathcal{F}_{\mathsf{ECDSA}}$.
   Otherwise, it sends $\mathsf{continue}_j$ to $\mathcal{F}_{\mathsf{ECDSA}}$.
9. The simulator $\mathcal{S}$ outputs whatever $\mathcal{A}$ outputs.

We prove the security by constructing a sequence of hybrids:

Hybrid$_0$: The real execution of the protocol.

Hybrid$_1$: In this hybrid, we consider an imaginary execution with a simulator who plays also the role of $\mathcal{F}_{\mathsf{ECDSA}}$, and thus knows the private key and the shares held by the honest parties. Hence, this simulator can run a real execution with $\mathcal{A}$. Clearly, **Hybrid$_1$** is identical to **Hybrid$_0$**.

Hybrid$_2$: This hybrid is identical to the previous hybrid, except that the zero-knowledge proofs sent by honest parties are replaced with a simulated proof as in the simulation. From the zero-knowledge property, the view of $\mathcal{A}$ in $Hybrid_2$ and $Hybrid_1$ is indistinguishable.

Hybrid$_3$: This hybrid is identical to the previous hybrid, but with the following change: the simulator chooses R as in the simulation. That is, it first chooses $R' = k \cdot E$ and then for each honest party P_j, it sets $k_j' = k \cdot c_j$ where c_j is chosen randomly. Then, for a random query answer ρ' programmed for $\mathcal{H}_\rho$, it sets $\rho = \rho' \cdot \sum_{j \in [n] \setminus C} c_j$. Similarly, it chooses μ and sets $\mu' = \mu - \rho'$
$cdot \sum_{i \in C} k_i$ for k_i that is extracted from $\mathcal{A}$ as in the simulation. The encryptions K_j' and commitments U_j for each honest party P_j are simulated honestly as in the real execution (but with simulated proofs).

Hybrid$_4$: The simulation is identical to the following hybrid: the simulator uses the zero-knowledge extractor to extract the secrets $\delta_i, \alpha_i, k_i', \beta_i, b_i$ of each corrupted party P_i as in the simulation. This enables it to know the output $a \cdot b$ of the two invocations of the scaled decryption protocol.

Hybrid$_5$: This hybrid is identical to the previous hybrid with one change: in the simulation of the scaled decryption, the simulator sets $F_{j^*} = a \cdot b \cdot H - \sum_{i \in [n] \setminus \{j^*\}} F_i$.

Hybrid$_6$: This hybrid is identical to **Hybrid$_5$** except for the following difference: K_{j^*}', C_{j^*} and U_{j^*} are set to random elements in their groups.

Hybrid$_7$: This hybrid is identical to **Hybrid$_6$**, except that now for all honest parties, the messages in the scaled decryption protocol are computed as follows:

for each F_j ($j \neq j^*$), the simulator chooses a random element Y in $\hat{G}_q$ and sets $F_j = b_j \cdot a \cdot H + Y$. (Note that F_{j^*} is set as in $\mathbf{Hybrid_6}$, so that the constraint $\sum_{j \in J} F_j + \sum_{i \in C} F_i = a \cdot b$ will hold).

$\mathbf{Hybrid_8}$: This hybrid is identical to $\mathbf{Hybrid_7}$, except that now K'_j and C_j for each honest party j with $j \neq j^*$ is set to be an encryption of 0.

$\mathbf{Hybrid_9}$: The hybrid identical to $\mathbf{Hybrid_8}$:, but now in the scaled decryption protocol, the message of each honest P_j with $j \neq j^*$ is a random element in $\hat{G}$ as in the code of the simulator $\mathcal{S}_{\mathsf{sd}}$.

$\mathbf{Hybrid_{10}}$: The signing protocol's simulation as described in the text above. That is, it first chooses $v = u \cdot (k + \rho^{-1} \cdot \mu)$ and $w = s \cdot v \cdot \rho \cdot r^{-1}$, and then set F_{j^*} as in the scaled description simulation.

From the point of view of $\mathcal{A}$, $\mathbf{Hybrid_3}$ is identical to $\mathbf{Hybrid_2}$, as in both cases the messages it sees follow the protocol's specification. The only difference is how R is being computed. In $\mathbf{Hybrid_2}$, $R = \rho \cdot R' + \mu \cdot E$ where ρ and μ are chosen randomly as the random oracle answers. Thus, $R = (\rho \cdot \sum_{i=1}^{n} k'_i + \mu) \cdot E$. In contrast, in $\mathbf{Hybrid_3}$, we have $\rho = \rho' \cdot \sum_{j \in J} c_j$ and $\mu = \mu' + \rho' \cdot \sum_{i \in C} k_i$ where μ' and ρ' are the random answers to the oracle queries. In this case, we have

$$R = \left(\rho' \cdot \sum_{j \in [n] \setminus C} c_j \cdot k + \mu' + \rho' \cdot \sum_{i \in C} k_i \right) \cdot E = \left(\rho' \left(\sum_{j \in [n] \setminus C} c_j \cdot k + \sum_{i \in C} k_i \right) + \mu' \right) \cdot E$$

However, since $k'_j = k \cdot c_j$ for each honest P_j, we conclude that R is the same in both cases.

$\mathbf{Hybrid_4}$ and $\mathbf{Hybrid_3}$ are identical, with the only difference is the event when the zero-knowledge extractor fails.

We proceed to show that $\mathbf{Hybrid_5}$ is equivalent to $\mathbf{Hybrid_4}$. The only difference between the two executions is the way F_{j^*} is computed. In $\mathbf{Hybrid_4}$, we have

$$\begin{aligned}
F_{j^*} &= (\beta_{j^*} \cdot q + b_{j^*}) \cdot A - \alpha_{j^*} \cdot B \\
&= (\beta_{j^*} \cdot q + b_{j^*}) \cdot (\alpha \cdot G_q + a \cdot H) - \alpha_{j^*} \cdot (\beta \cdot q + b) \cdot G_q \\
&= (\beta_{j^*} \cdot q + b_{j^*}) \cdot a \cdot H + ((\beta_{j^*} \cdot q + b_{j^*}) \cdot \alpha - \alpha_{j^*} \cdot (\beta \cdot q + b)) \cdot G_q \\
&= b_{j^*} \cdot a \cdot H + ((\beta_{j^*} \cdot q + b_{j^*}) \cdot \alpha - \alpha_{j^*} \cdot (\beta \cdot q + b)) \cdot G_q
\end{aligned}$$

In contrast, in $\mathbf{Hybrid_5}$ we have

$$\begin{aligned}
F_{j^*} &= a \cdot b \cdot H - \sum_{i \in [n] \setminus \{j^*\}} ((\beta_i \cdot q + b_i) \cdot A - \alpha_i \cdot B) \\
&= a \cdot b \cdot H - \sum_{i \in [n] \setminus \{j^*\}} ((\beta_i \cdot q + b_i) \cdot a \cdot H + ((\beta_i \cdot q + b_i) \cdot \alpha - \alpha_i \cdot (\beta \cdot q + b)) \cdot G_q) \\
&= b_{j^*} \cdot a \cdot H - \sum_{i \in [n] \setminus \{j^*\}} (((\beta_i \cdot q + b_i) \cdot \alpha - \alpha_i \cdot (\beta \cdot q + b)) \cdot G_q)
\end{aligned}$$

Note that

$$\sum_{i\in[n]\setminus\{j^*\}} (((\beta_i \cdot q + b_i) \cdot \alpha - \alpha_i \cdot (\beta \cdot q + b)) \cdot G_q)$$
$$= ((\beta \cdot q + b) \cdot \alpha - \alpha \cdot (\beta \cdot q + b)) \cdot G_q - (\beta_{j^*} \cdot q + b_{j^*}) \cdot \alpha - \alpha_{j^*} \cdot (\beta \cdot q + b)) \cdot G_q$$
$$= 0 - ((\beta_{j^*} \cdot q + b_{j^*}) \cdot \alpha - \alpha_{j^*} \cdot (\beta \cdot q + b)) \cdot G_q$$

and thus F_{j^*} is the same in both executions.

$\mathbf{Hybrid_6}$ is computationally indistinguishable from $\mathbf{Hybrid_5}$ under the HSM assumption, due to the hiding property of the encryption and commitment scheme.

We proceed to $\mathbf{Hybrid_7}$ with the following claim:

Claim. Under the hardness of the DDH assumption in $\hat{G}_q$, we have that $\mathbf{Hybrid_7}$ and $\mathbf{Hybrid_6}$ are computationally indistinguishable.

Proof. Consider an distinguisher D which participates in the following game: it receives as input

$$\{\Psi_j\}_{j=1}^{|J|}, \{\Phi_j\}_{j=1}^{|J|}, \{\Theta_{j,k}\}_{j,k=1}^{|J|}.$$

The goal is to determine whether for all j, k the tuple $(\Psi_j, \Phi_k, \Theta_{j,k})$ is a DH tuple or not. In other words, let $\Psi_j = \alpha_j \cdot G_q$ and $\Phi_j = \beta_j \cdot G_q$. Then, $\Theta_{j,k}$ is either $\alpha_j \cdot \Phi_k$ or a random element in $\hat{G}_q$.

From a standard hybrid argument, if the DDH assumption holds, then any probabilistic polynomial-time adversary wins the above game only with negligible probability.

Next, we show how the distinguisher D can generate an execution that is either $\mathbf{Hybrid_6}$ (when the tuples are DH tuples) or $\mathbf{Hybrid_7}$ (when the tuples are *not* DH tuples).

First, D needs to generate A_j and B_j for each $j \in J$. This is done by setting $A_j = \Psi_j + a_j \cdot H$ and $B_j = q \cdot \Phi_j + b_j \cdot G_q$. Next, D needs to compute F_j for each honest party P_j such that $j \neq j^*$, while setting F_{j^*} to be $a \cdot b - \sum_{i\in[n]\setminus j^*} F_j$. Recall that

$$\begin{aligned} F_j &= (\beta_j \cdot q + b_j) \cdot A - \alpha_j \cdot B \\ &= b_j \cdot a \cdot H + ((\beta_j \cdot q + b_j) \cdot \alpha - \alpha_j \cdot (\beta \cdot q + b)) \cdot G_q \\ &= b_j \cdot a \cdot H + q \cdot \beta_j \cdot \alpha \cdot G_q + b_j \cdot \alpha \cdot G_q - q \cdot \alpha_j \cdot \beta \cdot G_q + \alpha_j \cdot b \cdot G_q \end{aligned} \quad (1)$$

Letting

$$\alpha = \sum_{k\in J} \alpha_k + \sum_{i\in C} \alpha_i \qquad \beta = \sum_{k\in J} \beta_k + \sum_{i\in C} \beta_i$$

we can write each term in Eq. (1) as follows:

$$\begin{aligned} q \cdot \beta_j \cdot \alpha \cdot G_q &= q \cdot \left(\sum_{k\in J} \beta_j \cdot \alpha_k \cdot G_q\right) + q \cdot \left(\sum_{i\in C} \alpha_i\right) \cdot \beta_j \cdot G_q \\ &= q \cdot \sum_{k\in J} \Theta_{k,j} + q \cdot \left(\sum_{i\in C} \alpha_i\right) \cdot \Phi_j \end{aligned} \quad (2)$$

$$b_j \cdot \alpha \cdot G_q = b_j \cdot \left(\sum_{k \in J} \alpha_k\right) \cdot G_q + b_j \cdot \left(\sum_{i \in C} \alpha_i\right) \cdot G_q$$
$$= b_j \cdot \sum_{k \in J} \Psi_k + b_j \cdot \left(\sum_{i \in C} \alpha_i\right) \cdot G_q \tag{3}$$

$$q \cdot \alpha_j \cdot \beta \cdot G_q = q \cdot \left(\sum_{k \in J} \alpha_j \cdot \beta_k \cdot G_q\right) + q \cdot \left(\sum_{i \in C} \beta_i\right) \cdot \alpha_j \cdot G_q$$
$$= q \cdot \sum_{k \in J} \Theta_{j,k} + q \cdot \left(\sum_{i \in C} \beta_i\right) \cdot \Psi_j \tag{4}$$

and

$$b \cdot \alpha_j \cdot G_q = b \cdot \Psi_j. \tag{5}$$

Plugging Eq.(2) - (5) into Eq.(1), we have

$$F_j = b_j \cdot a \cdot H + q \cdot \sum_{k \in J} \Theta_{k,j} + q \cdot \left(\sum_{i \in C} \alpha_i\right) \cdot \Phi_j + b_j \cdot \sum_{k \in J} \Psi_k$$
$$+ b_j \cdot \left(\sum_{i \in C} \alpha_i\right) \cdot G_q - q \cdot \sum_{k \in J} \Theta_{j,k} - q \cdot \left(\sum_{i \in C} \beta_i\right) \cdot \Psi_j + b \cdot \Psi_j$$

First, observe that given the list of tuples received from its challenger, the distiguisher D has all the information needed to compute F_j. Second, observe that if the tuples are DH tuples, then F_j is computed exactly as in the real execution, i.e., as in Hybrid 6. In contrast, when the tuples are not DH tuples, then we have that each $\Theta_{j,k}$ and $\Theta_{k,j}$ are random elements. Since, we have an additive term in F_j of the form $q \cdot (\sum_{k \in J} \Theta_{k,j} - \sum_{k \in J} \Theta_{j,k})$, then F_j is computed exactly as in Hybrid 7. Hence, if the two hybrids are computationally distinguishable, then it is possible to distinguish between DH tuples and non-DH tuples. This concludes the proof. □

Next, $\mathbf{Hybrid_7}$ and $\mathbf{Hybrid_8}$ are computationally indistinguishable under the HSM assumption, since the only difference between the two hybrids is whether honest parties encrypt their real shares or encrypt the message '0'. Note that this claim is true only after the change to the honest parties' messages in the previous hybrid. If we would change the encrypted values to 0 before making the messages in the scaled decryption not depend on the randomness α_i, then the reduction to the hiding property of the encryption scheme would not work, as the distinguisher could not simulate the honest parties' messages.

Next, $\mathbf{Hybrid_8}$ and $\mathbf{Hybrid_9}$ are computationally indistinguishable under the HSM assumption. Note that in $\mathbf{Hybrid_8}$, each F_j is chosen by taking $(b_j \cdot a) \cdot H$ and masking it with a random element in $\hat{G}_q$. In contrast, in $\mathbf{Hybrid_9}$, the only change is that we take a random element in H instead of $(b_j \cdot a) \cdot H$.

Distinguishing between the two hybrids is thus equivalent to breaking the hiding property of our encryption scheme, that relies on the HSM assumption.

Finally, $\mathbf{Hybrid_9}$ and $\mathbf{Hybrid_{10}}$ are identically distributed, since the only difference is how v and w are chosen. Nevertheless, in both hybrids, they are random under the constraint that $w \cdot r \cdot (v \cdot \rho)^{-1} = s$. This concludes the proof. □