Practical Zero-Trust Threshold Signatures in Large-Scale Asynchronous Networks

Offir Friedman(✉)[0009−0002−1654−3934], Avichai Marmor[0009−0006−2175−8728], Dolev Mutzari[0000−0003−1003−0732], Yehonatan C. Scaly[0009−0003−1203−2122], and Yuval Spiizer[0009−0005−0879−2457]

dWallet Labs
{offir,avichai,dolev,yehonatan,yuval}@dwalletlabs.com

Abstract. Threshold signatures are a fundamental primitive in applied cryptography, primarily used to mitigate the custodial risk involved in managing keys. However, existing constructions rely on synchronous communication assumptions and fixed participant sets, limiting their applicability to real-world networks. In addition, as the committees managing those keys serve an ever-growing number of clients and assets, they become lucrative targets for attacks. This issue is often called the *Honeypot Problem*. A recent work proposed the *2PC-MPC* paradigm, in which a client and a decentralized network jointly generate signatures, aiming to mitigate this issue.

In this work, we present the first *asynchronous* 2PC-MPC protocol for ECDSA signatures, designed to operate over reliable broadcast channels as implemented in modern blockchains. Our protocol tolerates dynamic, post-determined quorums whose participants may change between rounds. This allows the protocol to align with asynchronous consensus layers while providing identifiable abort, public verifiability, guaranteed output delivery (assuming completeness of the broadcast channel) and censorship resistance for the client.

We introduce *global presigns*, which are generated autonomously by the network and remain client-agnostic until signing. This enables effective background preprocessing, and results in sub-second online signing network-side latency for dozens of concurrent clients.

From a technical perspective, our asynchronous design eliminates commitment rounds, yielding one-round client interaction for both key generation and signing. We also remove the need for zero-knowledge proofs toward the client. This results in a client complexity independent of network size, without proof aggregation (which is particularly challenging over asynchronous channels). Our constructions are proven UC-secure in the asynchronous setting. To achieve concrete security over 256-bit elliptic curves, we introduce the *Slightly-Enhanced ECDSA Unforgeability* assumption and provide a tight reduction in the EC-GGM model.

Beyond theoretical analysis, we implement our protocols and evaluate them in both a local benchmark and a live permissionless deployment with dozens of validators, demonstrating practical performance and robustness under heterogeneous execution speeds.

1 Introduction

In recent years, there has been growing interest in thresholdizing digital signatures [5, 16, 40], distributing the signing process among multiple parties. This is primarily because digital signatures serve as an authentication mechanism in Bitcoin [67], Ethereum [19], and similar platforms where the digital signature on a transaction constitutes a proof of ownership of the funds. Specifically, much effort has been made for thresholdizing ECDSA [1, 10, 12, 20–22, 29, 33, 34, 37, 38, 48–50, 61, 63, 81, 84, 85] due to its widespread use in cryptocurrency.

Threshold signature based solutions have been widely deployed in the cryptocurrency industry and focus mainly on two use cases: decentralized bridges (e.g. THORChain [77], Keep Network [64], Near Network [55] and Internet Computer [76]) and distributed custody (e.g. Coinbase Wallet [62], Qredo [58], Zengo [57], Fireblocks [43], Copper [30], Web3 Auth [59] and Lit Protocol [27]).

Ultimately, in both applications, the signing key may be shared between the client and a *distributed permissionless network* in a way that requires the active consent of both to produce a signature. Currently, for ECDSA-based protocols only trusted (i.e. without client involvement) custodians are deployed.

Additionally, to the best of our knowledge, due to inherent limitations of existing threshold ECDSA signing protocols, all real-world networks for both use-cases are deployed with a rather small number of nodes (e.g. Internet Computer uses up to 30 [76]). This undermines the vision of decentralization as a smaller number of nodes can more easily collude. Nowadays, blockchains typically have hundreds to thousands of validators (e.g., Ethereum currently has 4,540 active nodes [41]). Recently, [46] has taken a pivotal step towards increasing the number of nodes, and we follow their approach in this work. Their main observation is that designing the protocol over a *broadcast channel* is more scalable with the number of parties. Indeed, prior to the work of [46], all practical threshold ECDSA schemes were designed to work over *unicast channels*, assuming secure point-to-point (P2P) channels between *every* pair of parties. While secure point-to-point channels can be constructed in theory (under an appropriate PKI setup), in practice, they inherently incur a high network load and a message complexity quadratic in the number of parties, as each party computes and sends a message to every other party. Instead, [46] opts to rely on a *broadcast channel*, and reduces the message complexity to linear. Moreover, by working over a broadcast channel, their protocol achieves *identifiable abort* (IA) and *public verifiability* (PV). When the broadcast channel is implemented over a blockchain, one also gets *immutability of messages*, which enables easy *recovery from failures*. It also provides *incentive mechanisms* for faithful participation. Blockchains also provide *censorship resistance* for the client, as opposed to trusted custodians.

Nevertheless, some key aspects remain unaddressed. Primarily, their choice of assuming a synchronous broadcast channel sidesteps many of the difficulties that arise when implementing the protocol over existing broadcast channel implementations. Existing networks typically work over an *asynchronous* or a *partially synchronous reliable broadcast communication channel*. In particular, the

parties do not have synchronized clocks, cannot agree on which parties are online and which parties are unavailable. Therefore, parties who participate in one communication round may not be available for the next one. Section 2 discusses these gaps and places our approach in the context of prior work on threshold signatures. Building on [46], we address these critical challenges and provide an affirmative answer to the question below:

Can we design an asynchronous, permissionless, decentralized network, concurrently servicing numerous clients requesting digital signatures?

Zero Trust. Our protocol demands participation of a client in addition to the distributed network and thus maintains the *Zero Trust* principle [75], which has been foundational in blockchain ecosystems. Under this principle, no one is trusted, not even miners or validators, and every client calculates and verifies the state itself.

Currently, isolated networks that offer *interoperability* are compromising on Zero Trust principles in order to enable cross-network operations. Instead, they take a *Castle-and-Moat* approach, aiming to create a strong, fortified perimeter around their network. Unfortunately, *Castle-and-Moat Protocols* (CMPs) manage an ever-increasing amount of digital assets in cross-chain interactions, creating the *Honeypot Problem*: as a CMP gains higher *Total Value Locked* (TVL), the protocol becomes a more lucrative target for malicious players. For many years, we have witnessed attacks targeting CMPs. For instance, Wormhole [52] was exploited for over 320 million USD [82].

In order to resolve those issues and comply with the Zero Trust architecture, [46] proposed the *2PC-MPC* framework, which we follow in this work and briefly cover below. In this framework, both the client and a threshold of a decentralized network must collaborate in order to produce a signature. This ensures that even if the network is compromised, it cannot sign any transaction without client consent. In particular, on top of hacking the network, the adversary has to access the secret key share of each client individually, thereby addressing the honeypot problem. The only threat of an adversary controlling the network is Denial of Service (DoS), namely, preventing a client from using its wallet. Nevertheless, the requirement for the client to collaborate with the network to produce a signature raises many practical difficulties. First, the access structure is inherently hierarchical. Prior solutions that used a threshold access structure had to compromise on the size of the network, as the user must have the same weight as the whole network. Second, the client is typically lightweight, and does not have comparable computational power to parties in the decentralized network. Optimally, the computation complexity of the client should be independent of the network size. Third, the client typically does not have direct access to internal entities in the decentralized network, and might not even be aware of their identities. Exposing such communication channels may pose additional cybersecurity threat for parties in the network, and would also significantly increase the communication overhead of the client.

With the above issues in mind, [46] added to the 2PC-MPC *hierarchical access structure* an additional requirement that when met, allows for the system

to scale with the number of parties in the decentralized network. Namely, the client experiences a *two-party computation* (2PC) protocol, wherein the network emulates the second party by participating in *multi-party computation* (MPC) protocol.[1] Hence the name: 2PC-MPC.

Another important aspect of a decentralized system is the number of clients it can concurrently service. Prior works on threshold signatures are typically focused on a single key, and do not consider the system as a whole. Therefore, each node in the decentralized network must hold a private share per each wallet of each client in the system. Instead, in [46], the *private storage* of each party in the decentralized network is independent of the number of clients as well as the number of nodes. This is achieved by utilizing a *threshold additively homomorphic encryption scheme* (TAHE) [31]. Essentially, the network share of each signing key is stored in public, encrypted under the TAHE public key, and only the underlying AHE secret decryption key is secretly shared.[2] In this work, we take a different approach, enabling the network to utilize a single *global key share* which will be used across all clients. This change also significantly simplifies any resharing or reconfiguration of the secret. Specifically, one does not need to reshare the secret key of the TAHE scheme but instead may only reshare elements in $\mathbb{Z}_q$ and generate a fresh TAHE key.

1.1 Our Contribution

Our primary contribution is the design and implementation of a 2PC-MPC protocol for an *asynchronous* decentralized network, servicing numerous clients concurrently to securely generate ECDSA signatures. The motivation behind this design is to be on par with implementation requirements of existing blockchains and their underlying communication channels, as we cover in Section 2. Additionally, Section 2 also provides extensive comparison with prior works, further highlighting the challenges addressed in this work and the advantage of our approach.

In addition, our protocol supports *global presigns*, which are generated autonomously by the network and are not bound to any particular client or request until the final signing step. This design allows the network to precompute presigns in the background whenever spare capacity is available and then serve signing requests with lower latency. Practically, global presigns (i) improve utilization as each presign generated is more likely to be used, (ii) simplify load management and rate limiting by maintaining a presign buffer, and (iii) reduce client interactivity requirements, since the client is only involved at signing time rather than throughout presign generation.

[1] The client can verify that it interacts with a decentralized network if the communication channel with the network is a blockchain. However, their design potentially allows hiding the size of the network, in case a future application might demand that.

[2] The idea of using TAHE and storing only the secret-key shares to minimize private storage was proposed in the YOSO line of work [51].

Technical Contributions. On top of that, this work introduces several contributions that might have independent interest. As a side-effect of complying with asynchronous networks, we improve the round complexity, compared to the 2PC-MPC protocol in [46]. This is because we may not use commitment rounds, as parties may not be available in the next round to open them. As a result, both DKG and signing consist of a one-round trip between the blockchain and the client. Our sign phase can be broken into a *presign* phase, and an *online-signing phase*. The presign phase is executed without client involvement in an *offline phase*, potentially before the message to be signed is decided. This preprocessing can reduce computation and latency during high-demand periods. When instantiated in the standard threshold access structure, that is, by removing the client and retaining only the committee of parties, DKG completes in a single round, presigning requires two rounds; and the online signing step is a single round. In this sense, our round complexity matches the best known constructions that allow the presigning and signing phases to involve different subsets of parties [29]. If, instead, one enforces that the same set of parties participates in both presign and sign, then recent two-round protocols are also known [32, 65].

In addition, we alleviate the need for *zero-knowledge* (ZK) proofs towards the client, that were used in [46] to prove honest behavior. In order to admit the 2PC-MPC framework, [46] offered to aggregate the proofs from each party in the network. However, their aggregation approach is inherently synchronous and as such doesn't work for our purpose.

In terms of security, our protocol circumvents the *Enhanced-ECDSA unforgeability* assumption (proposed in [20]), even when allowing presigns. Enhanced ECDSA was proven secure in the EC-GGM model in [53], but the concrete parameters of the reduction are not sufficient when working over 256-bit elliptic curves commonly used nowadays (e.g., `secp256k1` [8] used in both Bitcoin and Ethereum). We propose the *Slightly-Enhanced ECDSA unforgeability* assumption, and provide a reduction (also in EC-GGM) with a satisfying complexity. We believe other threshold ECDSA protocols can be adjusted to rely on this assumption instead.

Moreover, in this work we also consider *proactive aborts*, namely, the participating subset in each session and each communication round is not only post-determined, but is also adversarially chosen. This allows our protocol to tolerate an adversary that not only corrupts part of the network participants, but also has control over the network communication channel.

In Section 1.2 we delve deeper into the concrete design of our proposed system. In Appendix A, a range of applications that can be built upon it is listed.

1.2 System Overview

In this work, we design a blockchain-based massively decentralized cryptosystem for securely maintaining digital wallets on the bulk of deployed blockchains. Below we briefly describe the system as a whole.

The blockchain. The blockchain is *permissionless*, and before the beginning of every *epoch*, *reconfiguration* allows *validators* to join or leave the network.

The blockchain is based on *Proof of Stake* (PoS), and each *validator* has a *voting power* proportional to its stake [60]. In turn, during each reconfiguration, validators may also withdraw part of their stake, or increase their stake, which could change their voting power accordingly. The weighted access structure is implemented by giving each validator a number of secret shares of the TAHE secret key proportional to their voting power. This only affects performance during the sign phase of the protocol.

Communication. Clients can communicate with the blockchain by sending transactions and parsing blocks. The MPC party emulation sub-protocols can be implemented over the *consensus protocol* [14,15,44] used for producing blocks. The *block proposer* validator is responsible for aggregating the last MPC round, computing the message to send to the client, and including it in its proposed block. Oftentimes, such a communication channel is either asynchronous (as in Narwhal-Tusk [35]) or *partially synchronous* (as in Narwhal [74]-Bullshark).

Policies. Upon creation, wallets will specify policies, either by using *smart contracts* (see [66, 79] and [54, 78] for recent overviews and surveys) implemented on-chain, or using light-clients [24] in order to validate policies posted on another blockchain. Before validators participate in signing a transaction, they validate that it matches the policy of the corresponding wallet. Policies can specify, for instance, a *whitelist* or a *blacklist* of accounts, a *transaction limit*, a limit on the number of transactions per epoch, or anything else that can be specified in a smart contract. Compromising a client will not allow an adversary to bypass the policy of its wallet. Compromising the blockchain can only be exploited to bypass policies for wallets for which the client share was also compromised. This ensures security can only be enhanced by sharing the key with the blockchain. The only added risk for the client is DOS of the network.

With the above architecture, our cryptosystem supports a range of versatile applications across multiple blockchains. In Section A, we illustrate several key use-cases enabled by the flexibility, security, and scalability of the above design.

1.3 Paper Organization

Section 2 covers limitations of prior work, pinpointing the main technical challenges we address. Section 3 contains preliminaries and definitions. Our 2PC-MPC ECDSA protocols are presented in Section 4. In Section 5 we analyze security, and Section 6 provides implementation details and evaluation. Additional material appears in the appendices: applications (Appendix A), extended preliminaries (Appendix B), formal languages (Appendix C), full security proofs (Appendix D), raw performance data (Appendix E), TAHE instantiation via Class Groups (Appendix F), and adaptation to Schnorr signatures (Appendix G).

2 Related Work

This section discusses prior work, focusing on [46] whose 2PC-MPC framework we follow.

The work in [46] is described over an *ideal broadcast functionality*. In reality, consensus protocols implement a broadcast channel over P2P channels, e.g. via gossiping [3]. This distinction is crucial, since implementing an ideal broadcast channel over asynchronous networks is theoretically impossible [44]. This abstraction complicates using off-the-shelf broadcast implementations; we address these issues in this work.

In particular, they assume a *synchronous communication channel*, with a known bound on clock drift and message transmission delays, allowing parties to agree if a party does not broadcast a message. This also occurs in [70], an adaptation of [38] that achieves identifiable abort via a synchronous broadcast channel. Synchronous communication is a stringent requirement [80]; blockchains typically use partially-synchronous [39] or asynchronous channels. In this work, protocols are defined over an asynchronous channel, supporting existing broadcast implementations (e.g., Narwhal-Tusk [35], Narwhal-Bullshark [74], Mysticeti [4]) with significantly lower latency. In asynchronous communication, parties compute and broadcast messages for the next round as soon as a threshold from the previous round broadcasts, without coordinating on which parties failed to participate. While ROAST [70] adapts FROST1 [56] for asynchronous communication, it requires running multiple FROST sessions in parallel (up to the number of honest parties), making it asymptotically more expensive and consuming significant storage for managing parallel sessions.

In addition, prior works such as [46] often require the same subset of parties to participate in all rounds of a subprotocol (DKG, presign or sign). This is not on par with existing broadcast channel implementations [4, 35, 74], which only ensure that an arbitrary threshold of parties is online during each round. Our protocol allows participants to dynamically change between rounds, which aligns with existing broadcast implementations. While [70] also addresses this indirectly by ensuring one session where a fixed threshold participates in all rounds, our approach allows different subsets in each round, yielding better performance.

Finally, beyond practical challenges, we address theoretical concerns in securing ECDSA with presigns. Prior threshold ECDSA protocols enabling presigns (e.g., [20, 22, 33, 38, 46]) assumed the Enhanced ECDSA Unforgeability assumption [20], stating that forging signatures is hard even if the adversary queries nonce parts before deciding on messages. This was established in the EC-GGM model [53], but with cubic reduction complexity, insufficient for 256-bit curves. We propose Slightly-Enhanced (SE) ECDSA Unforgeability and achieve quadratic complexity. Our presign consists of a pair of nonces; at signing, they are combined based on the message, client input, presign, and public key, making SE-ECDSA compatible with 256-bit curves. Notably, in Appendix G.3, we prove security of an analogous SE-Schnorr oracle under the *Algebraic One-More Discrete Log* (AOMDL) assumption, which may simplify simulation of threshold Schnorr protocols with presigns such as FROST3 [28].

3 Preliminaries

Notation. Group elements and sets are denoted by capital letters (G, H), while field elements are typically denoted by small letters x, r. Algorithms are denoted by calligraphic big letters $(\mathcal{A}, \mathcal{F})$. Message identifiers are denoted in a sans-serif font $(\mathsf{sid}, \mathsf{ssid})$. Subscripts are usually reserved for descriptors and superscripts are used for indexing. In cases where only a descriptor is needed, it will be subscripted. For a finite set P the notation $x \leftarrow P$ implies that x is sampled uniformly at random from P. We also write $x \leftarrow \mathcal{A}(\cdot)$ for the output of an algorithm.

Entities. We consider two sets of parties, denoted by A and B. A is a collection of "centralized" entities identified by pid_A, which play the role of the clients in the system. The set B represents a single distributed entity identified by pid_B, which plays the role of the decentralized network, wherein each party $B_i \in B$ is a validator.

Communication Model. Parties in B do not communicate directly with the parties in A. Instead, an authorized subset of B agrees on a common message to send back to A. The receiving party in A is unaware of the identity of the individual parties comprising B or its internal structure. Typically, blockchains expose a distributed ledger that is publicly accessible to everyone, yet no individual is exposed to direct communication channels with the validators that maintain it. This is formalized in $\mathcal{F}_{\mathsf{global\text{-}broadcast}}$ (Functionality 2).

The internal communication channel between parties in B is an asynchronous reliable broadcast (ARB) channel, which is captured in $\mathcal{F}_{\mathsf{broadcast}}$ (Functionality 1). This is an abstraction of the communication channel used in blockchains to achieve agreement on transactions. Some of the potential instantiations can be found in [4,35,74]. Following [73], we note that the functionality only ensures *consistency*, and not *completeness*. Completeness suggests that if a single honest party generates an output, then every honest party eventually generates an output. Instead, this is viewed as a property of the broadcast protocol itself. In our context, completeness ensures guaranteed output delivery, but for security and correctness of our protocol, consistent broadcast suffices.

Typically, a consensus protocol is implemented on top of a reliable broadcast channel, so as to agree on an *order* for the transactions. However, in our context, a weaker notion may suffice. Recall that our system produces signatures to be posted on other blockchains, who may be responsible for ordering these transactions. Therefore, we only require a mechanism to agree upon a subset of messages corresponding to an authorized subset of parties from the previous MPC round of a given session sid. This is captured by $\mathcal{F}_{\mathsf{ACS}}$ (Functionality 3). A potential instantiation can be found in [73]. Another possibility, although relying on consensus, is to take the first set of blocks that contain an authorized subset.

For brevity, when we refer to the *Reliable Broadcast Model*, we mean that we are working in the $(\mathcal{F}_{\mathsf{broadcast}}, \mathcal{F}_{\mathsf{global\text{-}broadcast}}, \mathcal{F}_{\mathsf{ACS}})$-hybrid model. The broadcast functionalities are adapted from [73], while $\mathcal{F}_{\mathsf{ACS}}$ is a rephrasing from [72].

Adversarial Model. We work within the *static* adversarial model, meaning that the adversary selects the set of parties to corrupt before any protocol begins and cannot corrupt any parties afterward. The adversary is also a *rushing* adversary, that first sees the broadcast messages of all honest parties before sending any of its own. In addition, the adversary can *proactively* and *adaptively* block or delay messages. This is modeled by letting it choose a valid subset of parties to participate in each round. Indeed, while an attacker aims to corrupt parties as early as possible and maintain control when feasible, the adaptive blocking and delaying of messages can be viewed as a form of static corruption over the network infrastructure itself. The adversary also has access to the headers of any communication between honest parties and ideal functionalities, and for ease of exposition, we assume this implicitly throughout without explicitly stating it each time.

Due to our use of a reliable broadcast channel, we assume that the number of (static + adaptive) corruptions in B is at most $n/3$ throughout, and otherwise B is considered corrupt. Essentially, an adversary that corrupts more than a third of the network may fork the network into two disjoint components and break consistency. It may also stop participating, in which case guaranteed output delivery fails, breaking liveness and censorship resistance.

Ideal Functionality. The security of the 2PC-MPC architecture is defined by the ideal functionality $\mathcal{F}_{\mathsf{tsig}}^{\mathcal{G},\Gamma_B}$ (Functionality 1). This functionality incorporates a signing oracle $\mathcal{G}$. The functionality regulates access to the oracle in a straightforward manner: it permits any authorized subset to request key generation, presigning, and signing operations from the oracle. Any other type of query is controlled by the adversary.

Zero-Knowledge Proofs (ZKPs). We recall the formal definition of *Zero-Knowledge Proofs* (ZKPs) and *Zero-Knowledge Proofs of Knowledge* (ZKPoK) in Appendix B.2. For a given ternary relation $R : \{0,1\}^* \times \{0,1\}^* \times \{0,1\}^* \mapsto \{0,1\}$, and every fixed $\mathsf{pp} \in \{0,1\}^*$, we define the following NP language $L[\mathsf{pp}] = \{x; w \mid R(\mathsf{pp}, x; w) = 1\}$. Namely, we require that for each $\mathsf{pp} \in \{0,1\}^*$, $R(\mathsf{pp}, \cdot; \cdot)$ is computable in polynomial time. In our context, pp consists of *public parameters* (e.g., a public encryption key or a specification of an elliptic curve), x is a *statement*, and w is a *witness*.

All of the ZKPs used are non-interactive, either by applying the Fiat-Shamir transform [42], or by applying Fischlin's transform [45] in case UC-extractable proofs are needed. A ZK protocol for the language L will be denoted as $\Pi_{\mathsf{zk}}^{L[\mathsf{pp}]}$ for the former case, and as $\Pi_{\mathsf{uc\text{-}zk}}^{L[\mathsf{pp}]}$ for the latter case. Formal definitions of the languages used in our protocol are provided in Appendix C.

Statement Aggregation. As all messages from B to parties in A must be aggregated, to this end we refer to a simple aggregation protocol. Given a language L and a binary operator $+ : L^2 \to L$, the aggregation protocol Π_{agg}^{L} works as follows. First, each party B_i sends to $\mathcal{F}_{\mathsf{broadcast}}$ a ZKP relative to L for $(\boldsymbol{Y}_i; \boldsymbol{w}_i)$ and validates the proofs of the other parties. Then, using $\mathcal{F}_{\mathsf{ACS}}$, the parties agree

Functionality 1: 2PC-MPC Signature Functionality $\mathcal{F}_{tsig}^{\mathcal{G},\Gamma_B}$

The functionality interacts with two disjoint sets of parties A and B along with an adversary $\mathcal{A}$ controlling a subset U. The functionality is parameterized by a signing oracle $\mathcal{G}$ and an access structure Γ_B.

1. **Key Generation**: Upon receiving $(\mathsf{keygen}, \mathsf{sid}, \mathsf{pid}_A)$ from A_{pid_A} along with $(\mathsf{keygen}, \mathsf{sid}, \mathsf{pid}_A, \mathsf{pid}_B)$ from a subset $S \in \Gamma_B$, do as follows:
   (a) Send $(\mathsf{keygen}, \mathsf{sid})$ to $\mathcal{G}$.
   (b) Act as a proxy between $\mathcal{A}$ and $\mathcal{G}$ until the oracle outputs $(\mathsf{keygen\text{-}output}, \mathsf{sid}, X)$ which it sends to all parties.
2. **Presign**: Upon receiving $(\mathsf{pres}, \mathsf{sid}, \mathsf{ssid}, \mathsf{pid}_B)$ from a subset $S \in \Gamma_B$, do as follows:
   (a) Send $(\mathsf{pres}, \mathsf{sid}, \mathsf{ssid})$ to $\mathcal{G}$.
   (b) Act as a proxy between $\mathcal{G}$ and $\mathcal{A}$ until the oracle outputs $(\mathsf{pres\text{-}output}, \mathsf{sid}, \mathsf{ssid}, \boldsymbol{K})$ which it sends to all parties.
3. **Sign**: Upon receiving $(\mathsf{sign}, X, \mathsf{msg}, \mathsf{sid}, \mathsf{ssid}, \mathsf{pid}_A)$ from $A_{\mathsf{pid}_A} \in A_X$ for which (X, A_X) is recorded, along with $(\mathsf{sign}, \mathsf{msg}, \mathsf{sid}, \mathsf{ssid}, \mathsf{pid}_A, \mathsf{pid}_B)$ from a subset $S \in \Gamma_B$, do as follows:
   (a) Send $(\mathsf{sign}, \mathsf{msg}, \mathsf{sid}, \mathsf{ssid})$ to $\mathcal{G}$.
   (b) Act as a proxy between $\mathcal{G}$ and $\mathcal{A}$ until the oracle outputs $(\mathsf{sign\text{-}output}, \mathsf{sid}, \mathsf{ssid}, \sigma)$ which it sends to all parties.

on a subset of the statements, and using $\mathcal{F}_{\mathsf{global\text{-}broadcast}}$, the statement $\sum_j \boldsymbol{Y}_j$ is sent to the client.

Homomorphic Commitments. For a commitment scheme with public parameters pp we denote the message space of the commitment as $\mathcal{M}_{\mathsf{pp}}$ and randomness space as $\mathcal{R}_{\mathsf{pp}}$. Unless specified otherwise $\mathsf{Com}_{\mathsf{pp}}$ refers to Pedersen Commitments [69] on the appropriate elliptic curve. Homomorphic Addition is denoted by $\oplus$ and scalar multiplication by $\odot$.

Threshold Additively Homomorphic Encryption (TAHE). Additively Homomorphic Encryption (AHE) is a public key encryption scheme that supports homomorphic addition of two ciphertexts. The scheme is parameterized by four abelian groups: the plaintext space $\mathcal{P}_{pk}$, the ciphertext space $\mathcal{C}_{pk}$, the randomness space $\mathcal{R}_{pk}$, and the key space $\mathcal{K}_\kappa$. We denote plaintext as pt, ciphertexts as ct, randomness as η, and public-secret key pairs as $(\mathsf{pk}; \mathsf{sk})$. We use $\oplus$ and $\odot$ for homomorphic addition and scalar multiplication respectively.

We utilize an AHE scheme with $\mathcal{P}_{pk} = \mathbb{Z}_q$, where q is the prime order of the elliptic curve used for the digital signature scheme. This can be achieved natively using class group-based AHE [23] or by interpreting the plaintext as a $\mathbb{Z}$-module using the Paillier scheme [68], as done in [46]. We require the AHE scheme to satisfy *circuit privacy* with respect to linear transformations. Specifically, the scheme admits *Secure Linear Evaluation* of two ciphertexts, ct_0 and ct_1, with coefficients a_0 and a_1, denoted as $\mathsf{AHE.Eval}(\mathsf{pk}, (\mathsf{ct}_0, \mathsf{ct}_1), (a_0, a_1); \eta)$. This evaluation should not reveal any information to an adversary holding sk and the

randomness used for encrypting ct_0 and ct_1, other than the output of the linear evaluation on the messages. We denote by Scale the operation that multiplies a single ciphertext by a scalar while preserving the same guarantee. Note that we only demand that $\mathsf{ct}_0, \mathsf{ct}_1$ are elements of the ciphertext space and are not necessarily valid encryptions, see Appendix F for discussion.

In our protocols, we utilize Threshold Additively Homomorphic Encryption (TAHE), which allows any set of $t+1$ parties to decrypt while preventing any smaller subset from doing so. This is captured by an ideal functionality $\mathcal{F}_{\mathsf{TAHE}}$ (Functionality 5). We define an Additively Homomorphic Encryption (AHE) scheme and extend it to the threshold security model. This functionality can be emulated using class group-based TAHE [18] or Paillier-based TAHE with circuit privacy ([46], following [47]). Importantly, to obtain security against malicious adversaries, TAHE schemes often use *verification keys* which should be thought of as homomorphic commitments on the secret key shares. The verification keys are produced in the *Distributed Key Generation* (DKG) phase, and are then used during the threshold decryption phase in order to prove correctness of the decryption shares.

4 Protocols

Next, we describe our protocol, consisting of four sub-protocols: global distributed key generation 1, client key generation 2, presign 3, and sign 4.

4.1 Global Key Generation

In an asynchronous network, a standard commit and reveal approach is unusable: openings may be delayed indefinitely, blocking progress. However, simply removing commitments would let parties adapt their inputs and break security. Our approach is to run two, "insecure" asynchronous sub-protocols without commitments, resulting in two public key parts $X_{0,B}, X_{1,B}$, each of which may be biased by the adversary. We observe that taking a pseudorandom combination of the two that is determined after all parties send their inputs is sufficient for security. In our protocol, this randomization happens during the client key generation phase.

Concretely, the protocol proceeds in a single round. Each party B_i samples two secret key shares $x_0^i, x_1^i \leftarrow \mathbb{Z}_q$, computes the corresponding public shares $X_0^i = x_0^i G$ and $X_1^i = x_1^i G$, and encrypts the secret shares under the TAHE public key. Each party then proves, via a ZKP in the language L_{EncDL}, that the ciphertexts and public shares are consistent. Using the aggregation protocol Π_{agg}, the parties aggregate their individual contributions into the global outputs $X_{0,B} = \sum_i X_0^i$, $X_{1,B} = \sum_i X_1^i$, along with ciphertexts $\mathsf{ct}_{\mathsf{key},0}, \mathsf{ct}_{\mathsf{key},1}$ encrypting the corresponding summed secret key shares $x_0 = \sum_i x_0^i$ and $x_1 = \sum_i x_1^i$. These global outputs are then used by all subsequent client key generation instances.

Protocol 1: Global Key-Generation: $\Pi_{\mathsf{global\text{-}keygen}}$: $\mathsf{GlobalKeyGen}(\mathbb{G}, G, q, \Gamma_B, \mathsf{sid}, \mathsf{pk})$

1. **Each** B_i **does the following:**
   - **Round 1**:
   (a) Sample $x_0^i, x_1^i \leftarrow \mathbb{Z}_q$ and $\eta_0^i, \eta_1^i \leftarrow \mathcal{R}_{pk}$.
   (b) Compute $X_0^i = x_0^i G$, $X_1^i = x_1^i G$ and $\mathsf{ct}_0^i = \mathsf{AHE.Enc}(\mathsf{pk}, x_0^i; \eta_0^i)$, $\mathsf{ct}_1^i = \mathsf{AHE.Enc}(\mathsf{pk}, x_1^i; \eta_1^i)$.
   (c) Run $\Pi_{\mathsf{agg}}^{\Gamma_B, L_{\mathsf{EncDL}}[\mathsf{pk},(\mathbb{G},G,q)]}$ on inputs $(X_0^i, \mathsf{ct}_0^i; x_0^i, \eta_0^i)$ and $(X_1^i, \mathsf{ct}_1^i; x_1^i, \eta_1^i)$ to obtain $(X_{0,B}, X_{1,B}, \mathsf{ct}_{\mathsf{key},0}, \mathsf{ct}_{\mathsf{key},1})$.
   - **Output**: $X_{0,B}, X_{1,B}, \mathsf{ct}_{\mathsf{key},0}, \mathsf{ct}_{\mathsf{key},1}$.

4.2 Client Key Generation

On the client side of the DKG, all clients operate over the common global DKG output $X_{0,B}, X_{1,B}$. Each client A samples its additive share of the secret key $x_A \leftarrow \mathbb{Z}_q$ and broadcasts $X_A = x_A \cdot G$, alongside a corresponding UC extractable zk proof π_{DL}. The coefficients $\mu_x^0, \mu_x^1, \mu_x^G \in \mathbb{Z}_q$ are then derived by querying the random oracle on the tuple $(\mathsf{sid}, \mathbb{G}, G, q, X_A, X_{0,B}, X_{1,B}, \pi_{\mathsf{DL}})$. Notably, the random oracle input includes π_{DL}, a UC-extractable ZKP of x_A. This binds the coefficients not only to the client's public share X_A but also to a proof of knowledge of the underlying secret, which is essential for security (see the proof sketch of Theorem 2). The network part is then computed as $X_B = \mu_x^0 \cdot X_{0,B} + \mu_x^1 \cdot X_{1,B} + \mu_x^G \cdot G$, and the public key is set as $X = X_B + X_A$. The corresponding secret key satisfies $x = \mu_x^0 \cdot x_0 + \mu_x^1 \cdot x_1 + \mu_x^G + x_A$, where x_0, x_1 are the network's secret shares from the global DKG.

Essentially, we view $(X_{0,B}, X_{1,B})$ as a commitment to the network part. As such the client does not need to commit to X_A as it only observes a commitment of X_B. Then, the affine transformation defined by $\mu_x^0, \mu_x^1, \mu_x^G$, can be viewed as the opening of the network's part X_B. Since the coefficients depend on the client part X_A, X_B is "opened" only after the client decides on its input. Notably, the client cannot commit before because the network DKG is global and we may not assume all clients could commit before it is performed.

4.3 Global Presign

Global presigns may be consumed by any client to generate a signature in the online signing phase. As such, by the stage they are created the client nonce contribution is yet to be committed. Therefore, similarly to the technique used in the global DKG, the network will generate two public nonces $K_{0,B}, K_{1,B}$. During online signing, the final network part will be a-posteriori derived, as a pseudorandom linear combination of the two parts $\mu_k^0 K_{0,B} + \mu_k^1 K_{1,B} + \mu_k^G G$.

The protocol proceeds in two rounds and uses a random mask γ_B to blind the secret key shares. In the first round, each party B_i samples a random mask share $\gamma_B^i \leftarrow \mathbb{Z}_q$, encrypts it under pk, and homomorphically multiplies the global key ciphertexts $\mathsf{ct}_{\mathsf{key},0}, \mathsf{ct}_{\mathsf{key},1}$ by γ_B^i to obtain encryptions of $\gamma_B^i \cdot x_0$ and $\gamma_B^i \cdot x_1$. These

Protocol 2: Centralized Party Key-Generation: $\Pi_{\mathsf{centralized\text{-}keygen}}$: $\mathsf{CentralizedKeyGen}(\mathbb{G}, G, q, \mathbf{sid}, \mathbf{pk}; \mathbf{global\text{-}keygen}, A_{\mathsf{pid}})$

1. A_{pid} **does the following:**
   - **Round 1**:
   (a) Verify that $X_{0,B}, X_{1,B} \in \mathbb{G}$ and $\mathsf{ct}_{\mathsf{key},0}, \mathsf{ct}_{\mathsf{key},1} \in \mathcal{C}_{pk}$. If any check fails, terminate and designate B as malicious; otherwise continue.
   (b) Sample $x_{A_{\mathsf{pid}}} \leftarrow \mathbb{Z}_q$ and compute $X_{A_{\mathsf{pid}}} = x_{A_{\mathsf{pid}}} G$.
   (c) Run $\Pi_{\mathsf{uc\text{-}zk}}^{L_{\mathsf{DL}}[(\mathbb{G},G,q)]}(X_{A_{\mathsf{pid}}}; x_{A_{\mathsf{pid}}})$ to generate a proof π_{DL}.
   (d) Send $(\mathsf{broadcast}, \mathsf{sid}, \mathsf{pid}_A, X_{A_{\mathsf{pid}}}, \pi_{\mathsf{DL}})$ to $\mathcal{F}_{\mathsf{broadcast}}$.
2. **Output**
   – Each B_i:
   (a) Receive $(\mathsf{sid}, \mathsf{pid}_A, X_{A_{\mathsf{pid}}}, \pi_{\mathsf{DL}})$.
   (b) Verify π_{DL}. If it fails, designate A_{pid} as malicious and abort; otherwise continue.
   (c) Query the random oracle $\mathcal{H}(\mathsf{sid}, \mathbb{G}, G, q, X_{A_{\mathsf{pid}}}, X_{0,B}, X_{1,B}, \pi_{\mathsf{DL}})$ to obtain $\mu_x^0, \mu_x^1, \mu_x^G$.
   (d) Output $X_{A_{\mathsf{pid}}}$ and record $(\mathsf{keygen}, \mathsf{pid}_A, \mathsf{global\text{-}keygen}, X_{A_{\mathsf{pid}}}, \mu_x^0, \mu_x^1, \mu_x^G)$.
   – A_{pid}:
   (a) Query the random oracle $\mathcal{H}(\mathsf{sid}, \mathbb{G}, G, q, X_{A_{\mathsf{pid}}}, X_{0,B}, X_{1,B}, \pi_{\mathsf{DL}})$ to obtain $\mu_x^0, \mu_x^1, \mu_x^G$.
   (b) Output $X_{A_{\mathsf{pid}}}$ and record $(\mathsf{keygen}, \mathsf{pid}_A, \mathsf{global\text{-}keygen}, X_{A_{\mathsf{pid}}}, \mu_x^0, \mu_x^1, \mu_x^G)$.

are aggregated via Π_{agg} to produce ct_γ (encrypting the combined mask $\gamma_B = \sum_i \gamma_B^i$) and $\mathsf{ct}_{\gamma_B \cdot \mathsf{key},0}$, $\mathsf{ct}_{\gamma_B \cdot \mathsf{key},1}$ (encrypting $\gamma_B \cdot x_0$ and $\gamma_B \cdot x_1$). In the second round, each party samples nonce shares $k_0^i, k_1^i \leftarrow \mathbb{Z}_q$ and homomorphically scales ct_γ by each nonce share, yielding encryptions of $k_0^i \cdot \gamma_B$ and $k_1^i \cdot \gamma_B$. It also computes the public nonce parts $R_{B,0}^i = k_0^i G$ and $R_{B,1}^i = k_1^i G$. The parties again aggregate via Π_{agg} to obtain the final outputs. The mask γ_B will later cancel during the sign phase when the ciphertext ct_A is divided by $\mathsf{ct}_{\alpha,\beta}$, ensuring that the final signature is correctly formed.

4.4 Online Sign

In the 2PC-MPC framework, the client runtime and communication overhead should be independent of network size. It turns out that the challenge lies in the ZKPs, as the statements sent by the network are always summed in our protocols. However, we find proof aggregation for the corresponding summed-up statements to be quite challenging over an asynchronous network.

Instead, we remove the proofs toward the client altogether. Conceptually, we want to view the client as a signing oracle. The unforgeability of the signing oracle will ensure the network will not be able to forge a signature regardless of what it sends to the client. While this concept somewhat works for EdDSA, it fails for ECDSA and allows non-trivial attacks, as explained below.

Protocol 3: Global Presign: $\Pi^{\mathsf{ECDSA}}_{\mathsf{global\text{-}presign}}$:
$\mathsf{GlobalPresign}(\mathbb{G}, G, q, \mathsf{sid}, \mathsf{pk}; \mathbf{global\text{-}keygen}, \mathbf{ct}_{\mathsf{key},0}, \mathbf{ct}_{\mathsf{key},1})$

The protocol outputs:

- ct_γ – an encryption of a randomizer γ_B.
- $\mathsf{ct}_{\gamma_B \cdot \mathsf{key},0}$, $\mathsf{ct}_{\gamma_B \cdot \mathsf{key},1}$ – encryptions of $\gamma_B \cdot x_0$ and $\gamma_B \cdot x_1$, respectively.
- $\mathsf{ct}_{\gamma \cdot k_0}$, $\mathsf{ct}_{\gamma \cdot k_1}$ – encryptions of $k_0 \cdot \gamma_B$ and $k_1 \cdot \gamma_B$, respectively.
- $R_{B,0}$, $R_{B,1}$ – curve points equal to $k_0 G$ and $k_1 G$, respectively.

1. **Each B_i does the following:**
   - **Round 1**:
   (a) Sample $\gamma_B^i \leftarrow \mathbb{Z}_q$ and $\eta_1^i, \eta_2^i, \eta_3^i \leftarrow \mathcal{R}_{pk}$, and compute:
      i. $\mathsf{ct}_\gamma^i = \mathsf{AHE.Enc}(\mathsf{pk}, \gamma_B^i; \eta_1^i)$,
      ii. $\mathsf{ct}^i_{\gamma_B \cdot \mathsf{key},0} = \mathsf{AHE.Scale}(\mathsf{pk}, \mathsf{ct}_{\mathsf{key},0}, \gamma_B^i; \eta_2^i)$,
      iii. $\mathsf{ct}^i_{\gamma_B \cdot \mathsf{key},1} = \mathsf{AHE.Scale}(\mathsf{pk}, \mathsf{ct}_{\mathsf{key},1}, \gamma_B^i; \eta_3^i)$.
   (b) Run $\Pi^{\Gamma_B,\, L_{\mathsf{EncDH}}[\mathsf{pk}, \mathsf{ct}_{\mathsf{key},0}, \mathsf{ct}_{\mathsf{key},1}]}_{\mathsf{agg}}$ on inputs $(\mathsf{ct}_\gamma^i, \mathsf{ct}^i_{\gamma_B \cdot \mathsf{key},0}, \mathsf{ct}^i_{\gamma_B \cdot \mathsf{key},1}; \gamma_B^i, \eta_1^i, \eta_2^i, \eta_3^i)$ to obtain $(\mathsf{ct}_\gamma, \mathsf{ct}_{\gamma_B \cdot \mathsf{key},0}, \mathsf{ct}_{\gamma_B \cdot \mathsf{key},1})$.
   - **Round 2**:
   (a) Sample $k_0^i, k_1^i \leftarrow \mathbb{Z}_q$ and $\eta^i_{4,0}, \eta^i_{4,1} \leftarrow \mathcal{R}_{pk}$, and compute:
      i. $\mathsf{ct}^i_{\gamma \cdot k_0} = \mathsf{AHE.Scale}(\mathsf{pk}, \mathsf{ct}_\gamma, k_0^i; \eta^i_{4,0})$,
      ii. $\mathsf{ct}^i_{\gamma \cdot k_1} = \mathsf{AHE.Scale}(\mathsf{pk}, \mathsf{ct}_\gamma, k_1^i; \eta^i_{4,1})$,
      iii. $R^i_{B,0} = k_0^i G$,
      iv. $R^i_{B,1} = k_1^i G$.
   (b) Run $\Pi^{\Gamma_B,\, L_{\mathsf{ScaleDL}}[\mathsf{pk}, (\mathbb{G}, G, q), \mathsf{ct}_\gamma]}_{\mathsf{agg}}$ on inputs $(\mathsf{ct}^i_{\gamma \cdot k_0}, R^i_{B,0}; k_0^i, \eta^i_{4,0})$ and $(\mathsf{ct}^i_{\gamma \cdot k_1}, R^i_{B,1}; k_1^i, \eta^i_{4,1})$ to obtain $(\mathsf{ct}_{\gamma \cdot k_0}, R_{B,0})$ and $(\mathsf{ct}_{\gamma \cdot k_1}, R_{B,1})$.
   (c) **Output**: Record $(\mathsf{sid}, \mathsf{global\text{-}presign}, \mathsf{ct}_\gamma, \mathsf{ct}_{\gamma_B \cdot \mathsf{key},0}, \mathsf{ct}_{\gamma_B \cdot \mathsf{key},1}, \mathsf{ct}_{\gamma \cdot k_0}, \mathsf{ct}_{\gamma \cdot k_1}, R_{B,0}, R_{B,1})$.

In our protocol, the client essentially gets the ingredients it needs to compute $R := k_A^{-1} \cdot R_B$ and an encryption of $\sigma_A := k_A \cdot (m + rx)$ under B's public key. One can think of (R, σ_A) as an ECDSA signature with respect to the group $(\mathbb{G}, R_B)$, signing key x, and nonce k_A. Party B can then translate it into an ECDSA signature over $(\mathbb{G}, G)$ by decrypting σ_A and dividing it by k_B, where $R_B = k_B \cdot G$. Therefore, intuitively, party A acts as an ECDSA signing oracle, with the key distinction that the adversary can pick the generator of the group R_B every time.

Nevertheless, the above protocol enables the following attack. Party B sends X instead of R_B, which becomes possible now that a ZKP for $R_B \leftarrow X$ is not required. It will get $k_A^{-1} \cdot X$ and (the encryption of) $\sigma_A = k_A(m + rx)$. It then multiplies $k_A^{-1} X$ by σ_A to get $(m + rx) \cdot X$, which gives away $x^2 \cdot G$. By repeating this, it can get $x^k \cdot G$, for any $k \in \mathsf{poly}(\kappa)$. This is an instance of the Strong Diffie-Hellman Problem which is known to be broken in certain settings [26].

In order to resolve this issue, we allow the client to apply an affine transformation to the nonce of B. In particular, it sends $\mathsf{Enc}(\alpha k_B + \beta)$. Notice that we could not fix α or β as a single degree of freedom would allow the adversary to extract them bringing us back to square one. This may seem to shift too much power to the client. However, as long as the derivation of the network nonce part R_B depends on α, β and k_A, this extra power is still captured by the slightly enhanced signing oracle. To this end, the client sends homomorphic commitments to α, β and k_A, and the network nonce part is computed as $R_B = \mu_k^0 \cdot R_{B,0} + \mu_k^1 \cdot R_{B,1} + \mu_k^G \cdot G$, where the coefficients $\mu_k^0, \mu_k^1, \mu_k^G$ are derived based on those commitments as well.

The sign protocol (Protocol 4) has the client first derive the combined encrypted key $\mathsf{ct}_{\gamma \cdot \mathsf{key}}$ from the presign outputs using the key-derivation coefficients $\mu_x^0, \mu_x^1, \mu_x^G$ and the public key X. The client then samples $k_A, \alpha, \beta \leftarrow \mathbb{Z}_q$ and publishes Pedersen commitments C_k, C_α, C_β along with ZKPs of well-formedness. The nonce-derivation coefficients $\mu_k^0, \mu_k^1, \mu_k^G$ are obtained from a random oracle query on the public data together with these commitments and proofs. Using $\mu_k^0, \mu_k^1, \mu_k^G$, the client combines the two presign nonces into the network nonce R'_B, applies the affine transformation to get $R_B = \alpha R'_B + \beta G$, and computes $R = k_A^{-1} \cdot R_B$. The client also homomorphically evaluates the ciphertexts to produce $\mathsf{ct}_{\alpha,\beta}$ (encrypting $\gamma_B \cdot k_B$) and ct_A (encrypting $\gamma_B \cdot k_A(m + rx)$), accompanied by ZKPs proving consistency with the commitments. The network verifies all proofs, decrypts ct_A and $\mathsf{ct}_{\alpha,\beta}$ via $\mathcal{F}_{\mathsf{TAHE}}$, and recovers $s = \mathsf{ct}_{\alpha,\beta}^{-1} \cdot \mathsf{ct}_A = k_B^{-1} \cdot k_A(m + rx) = k^{-1}(m + rx)$, yielding the final ECDSA signature (r, s).

5 Security

In this section, we introduce the *Slightly Enhanced* signing oracle 2 and prove its security in the EC-GGM model in Section 5.1. We then prove the security of our protocols in Section 5.2. Because our protocol has no commitments, a rushing adversary can introduce an *additive bias* into the public-key shares $X_{B,0}, X_{B,1}$ and into the public nonce shares $R_{B,0}, R_{B,1}$. We therefore model this explicitly by allowing an additive bias in the signing oracle. Moreover, the adversarial model permits *adaptive message delays*, which give the adversary additional control.

In the standalone setting, this can be handled by guessing a single honest party that participates in the round and rewinding if the adversary delays that party's message. In the UC setting, we instead proceed as follows. The simulator queries the signing oracle to obtain X, and then sends $X + x_i \cdot G$, where $x_i \leftarrow \mathbb{Z}_q$ is sampled uniformly at random (for each relevant party index i). Once the set of honest participants S_H is determined, the resulting final value becomes $|S_H| \cdot X + \sum_{i \in S_H} x_i + \beta$. Consequently, the signing oracle must allow not only an additive bias β, but also a *multiplicative bias* (of $|S_H|$). It remains to UC-extract β. We do so by extracting the secret key of the TAHE scheme and then decrypting the appropriate ciphertexts.

In addition to the aforementioned biases, our signing oracle supports global keys and global presigns. Namely, the adversary may derive polynomially many

public keys from the same X_0, X_1 and presigns are connected to a specific key only upon a sign request. We call this new oracle *Slightly Enhanced* and it is presented in 2.

Functionality 2: Slightly Enhanced ECDSA Signing Oracle: $\mathcal{G}^*_{\mathsf{SE\text{-}ECDSA}}$

1. Generation of global keys: Sample $\tilde{x}_0, \tilde{x}_1 \leftarrow \mathbb{Z}_q$ and send $\tilde{X}_0 = \tilde{x}_0 \cdot G$ and $\tilde{X}_1 = \tilde{x}_1 \cdot G$.
2. On input $(\mathsf{biaskey}, \mathsf{sid}, \alpha_x^0, \beta_x^0, \alpha_x^1, \beta_x^1)$ set $x_0 \leftarrow \alpha_x^0 \cdot \tilde{x}_0 + \beta_x^0$ and $x_1 \leftarrow \alpha_x^1 \cdot \tilde{x}_1 + \beta_x^1$, together with $X_0 \leftarrow x_0 \cdot G$ and $X_1 \leftarrow x_1 \cdot G$. Then set $(\mu_x^0, \mu_x^1, \mu_x^G) = \mathcal{H}(\mathsf{sid}, X_0, X_1)$. Finally, set $x = \mu_x^0 \cdot x_0 + \mu_x^1 \cdot x_1 + \mu_x^G$ and $X = x \cdot G$, and record $(\mathsf{sid}, X; x)$.
3. On input $(\mathsf{pres}, \mathsf{ssid})$, sample $\tilde{k}_0, \tilde{k}_1 \leftarrow \mathbb{Z}_q$, compute $\tilde{R}_0 = \tilde{k}_0 \cdot G$ and $\tilde{R}_1 = \tilde{k}_1 \cdot G$, record $(\mathsf{ssid}, \tilde{R}_0, \tilde{R}_1; \tilde{k}_0, \tilde{k}_1)$ and send $(\mathsf{ssid}, \tilde{R}_0, \tilde{R}_1)$.
4. On input $(\mathsf{sign}, \mathsf{sid}, \mathsf{ssid}, \mathsf{msg}; \delta_x, \alpha_{\mathsf{pres},0}, \beta_{\mathsf{pres},0}, \alpha_{\mathsf{pres},1}, \beta_{\mathsf{pres},1})$ do:
   (a) Retrieve $(\mathsf{sid}, X; x)$ and $(\mathsf{ssid}, \tilde{R}_0, \tilde{R}_1; \tilde{k}_0, \tilde{k}_1)$. If no such ssid or sid exists, ignore.
   (b) Compute $k_0 \leftarrow \alpha_{\mathsf{pres},0} \cdot \tilde{k}_0 + \beta_{\mathsf{pres},0}$ and $k_1 \leftarrow \alpha_{\mathsf{pres},1} \cdot \tilde{k}_1 + \beta_{\mathsf{pres},1}$.
   (c) Set $R_0 = k_0 \cdot G$ and $R_1 = k_1 \cdot G$.
   (d) Set $(\mu_k^0, \mu_k^1, \mu_k^G) = \mathcal{H}(\mathsf{sid}, \mathsf{ssid}, X, \delta_x, R_0, R_1, \mathsf{msg})$, and set $k = \mu_k^0 \cdot k_0 + \mu_k^1 \cdot k_1 + \mu_k^G$ and $R = k \cdot G$.
   (e) Set $r = R_{x-axis}$ and compute $s = k^{-1} \cdot (\mathcal{H}(\mathsf{msg}) + r \cdot (x + \delta_x))$.
   (f) Erase $(\mathsf{ssid}, \tilde{R}_0, \tilde{R}_1; \tilde{k}_0, \tilde{k}_1)$ from memory and return $(\mathsf{sid}, \mathsf{ssid}, \mathsf{msg}; \alpha_{\mathsf{pres},0}, \beta_{\mathsf{pres},0}, \alpha_{\mathsf{pres},1}, \beta_{\mathsf{pres},1}; R, s, r)$.

5.1 Analyzing Slightly Enhanced ECDSA

In this section, we provide a sketch of the security proof of the existential unforgeability game 1 with respect to the signing oracle 2 in the EC-GGM model [53]. In this model, a *group oracle* essentially samples a random mapping π between $\mathbb{Z}_q$ and the curve $\mathbb{G}$. The adversary interacts with the group oracle in order to perform computations over $\mathbb{G}$.

Theorem 1 (10) *Let $\mathcal{A}$ be an adversary to the existential unforgeability game 1 with respect to the ECDSA oracle $\mathcal{G}^*_{\mathsf{SE\text{-}ECDSA}}$ (Functionality 2) in the EC-GGM model (Lazy Simulation 6), that makes at most N presignature, signing, hash, or group queries, where $\mathcal{H}_{\mathsf{key}}, \mathcal{H}_{\mathcal{M}}$ and all $\mathcal{H}_k$ are modeled as independent Random Oracles. Then $\mathsf{Adv}(\mathcal{A}, \mathsf{Exp}_{\mathsf{EU}}^{\mathcal{G}^*_{\mathsf{SE\text{-}ECDSA}}}, \textit{lazy-sim}) \leq \mathcal{O}(\frac{N^2}{q})$.*

Proof (sketch). Our proof follows the steps taken in [53], and we refer to Appendix D.2 for a detailed proof.

As in [53], the proof consists of three steps. First, the signing oracle is simulated with a *lazy simulation*, wherein group mapping and the random oracle outputs are sampled "on the fly", only upon queries of new values. Second, the lazy simulation is reduced to a *symbolic simulation*, where in particular,

the private signing key and the random values of the signatures are replaced with combinations of symbolic variables. The main impact of using a slightly enhanced oracle is apparent in the symbolic simulation. In contrast to [53], we show that by using two nonces in the presign, it is possible to keep one of the two corresponding variables symbolic throughout the entire simulation. This is the main difference which leads to a better security bound. Specifically, we prove that any adversary to our protocol that can forge a signature in the EC-GGM model with non-negligible probability must use $\Omega(\sqrt{q})$ overall oracle queries.

The security proof of the symbolic simulation is based on the symbolic verification equation of the forged signature and the equality of symbolic polynomials. Following [53], we then split the proof into several cases, depending on the creation of R^* of the forged signature. We show that the probability of each case is negligible.

5.2 Protocol Simulations.

Next, we describe the main security theorems satisfied by our protocols.

Signing Protocols. The ECDSA based protocols UC realize their corresponding threshold signing functionality (Functionality 1) with Oracle 2 in the *reliable broadcast model* and $\mathcal{F}_{\mathsf{TAHE}}$ (Functionality 5)-hybrid model.

Theorem 2 *The protocols Π_{keygen}, $\Pi^{ECDSA}_{presign}$, and Π^{ECDSA}_{sign} UC realize $\mathcal{F}_{tsig}^{\mathcal{G}^*_{\mathsf{SE\text{-}ECDSA}},\binom{[n]}{t+1}}$ in the* reliable broadcast model *and $\mathcal{F}_{\mathsf{TAHE}}$-hybrid model.*

Proof (sketch). We refer the reader to Appendix D.1 for the detailed proof. For the simulator to succeed in the emulation of the ideal functionality, it must land on the signing oracle's public key and public nonce. The general approach is as follows: each sub-protocol simulation forwards its query to $\mathcal{G}^*_{\mathsf{SE\text{-}ECDSA}}$, receiving a value. The simulator embeds the value sent by the signing oracle into the honest parties' messages additively, then extracts the adversary's secrets. The actual output of the protocol is then a known bias away from the real execution, and the simulator inputs $\mathcal{G}^*_{\mathsf{SE\text{-}ECDSA}}$ with the correct biases. While the protocol allows for different sources of biases on these values, they are all affine linear, and as such even a composition of them is still affine linear and captured by the allowed biases in $\mathcal{G}^*_{\mathsf{SE\text{-}ECDSA}}$. Thus the main difficulty becomes extracting the adversary's values without re-winding and in the correct timing in the protocol.

When A is honest, the simulator embeds the oracle's public key and nonce directly into X_A and R, which works since the client speaks second. The adversary's secrets are extracted via the TAHE secret key which is itself extracted in the $\mathcal{F}_{\mathsf{TAHE}}$ simulation; and the resulting signature is embedded into the $\mathcal{F}_{\mathsf{TAHE}}$ decryption outputs. When A is malicious, during DKG it can adaptively query the random oracle for $\mu_x^0, \mu_x^1, \mu_x^G$ on various X_A values after observing X_B^0, X_B^1, before deciding which to send. The simulator must answer these queries *before* knowing the final X_A. This is resolved by including π_{DL}, a UC-extractable ZKP of x_A, in the random oracle input: the simulator extracts x_A upon each query,

interprets it as an additive bias, and forwards it to $\mathcal{G}^*_{\mathsf{SE\text{-}ECDSA}}$ which determines $\mu_x^0, \mu_x^1, \mu_x^G$. A similar challenge arises during sign: the random oracle input for $\mu_k^0, \mu_k^1, \mu_k^G$ includes UC-extractable ZKPs of k_A, α, and β, allowing the simulator to extract them and adjust the μ_k coefficients so that the resulting nonce aligns with the oracle's re-randomization.

Indistinguishability follows from circuit privacy and semantic security of the AHE scheme, the zero-knowledge property, and random oracle programming.

6 Performance

We implemented our ECDSA protocol in Rust; the code will be made publicly available upon publication. We instantiate our TAHE with Class Groups based on [17]. We also build upon [13] and implement the first constant-time Class-Group library. All times are reported for $\sigma = 64, \kappa = 128, t = \frac{2}{3}n$ and the secp256k1 curve.

Local Benchmarks. We conduct controlled local experiments on a MacBook Pro Apple M2 Max with a 3.49 GHz CPU running single-threaded. Figure 1 shows the network performance scaling of our Presign and Sign protocols with increasing party count. We omit DKG data as it is negligible except for the TAHE key generation. The Sign protocol demonstrates log-linear scaling, growing from 0.38 s (12 parties) to 0.94 s (120 parties), while the Presign protocol exhibits higher computational overhead, ranging from 2.1 s to 5.2 s.

The centralized party computation time remains effectively constant across all party counts, averaging 1.93 s with a coefficient of variation below 1%.

Real-World Deployment. We report performance data from real-world usage of our protocol over the Narwhal consensus layer with $n = 85$ validators in a permissionless setting where participants may dynamically enter and leave. Validator nodes are required to have minimum hardware specifications of 16 physical CPU cores, 128 GB RAM, 4 TB NVMe storage, and 1 Gbps network connectivity. That said, this requirement cannot be enforced or verified. Figure 2 shows the total protocol time distribution across validators (i.e. including both computation and communication). The Sign protocol achieves a mean time of 1.05 s with a standard deviation of 0.47 s, while the Presign protocol requires 17.3 s on average with 9.2 s standard deviation. The significant variance (CV $\approx$ 45-53%) demonstrates that asynchronous execution provides performance benefits beyond being merely a consensus necessity; validators complete computations at heterogeneous speeds without blocking protocol progress.

We do not have direct access to a global breakdown of communication versus computation time across validators. That said an execution report was provided to us by a single participating validator. Based on this report, communication accounts for 88.3% of the Presign protocol execution time and 54.4% of the Sign protocol execution time, where communication includes all consensus layer overhead as well as data uploads and downloads.

Disclosure of Interests. Offir Friedman, Avichai Marmor, Dolev Mutzari, Yehonatan C. Scaly, and Yuval Spiizer were employees of dWallet Labs Ltd. while working on this paper and may hold equity in the company. The authors declare that they have no other competing interests relevant to the content of this article.

References

1. Abram, D., Nof, A., Orlandi, C., Scholl, P., Shlomovits, O.: Low-bandwidth threshold ECDSA via pseudorandom correlation generators. In: SP. pp. 2554–2572. IEEE (2022)
2. Akyildirim, E., Corbet, S., Katsiampa, P., Kellard, N., Sensoy, A.: The development of bitcoin futures: Exploring the interactions between cryptocurrency derivatives. Finance Research Letters **34**, 101234 (2020). https://doi.org/https://doi.org/10.1016/j.frl.2019.07.007, https://www.sciencedirect.com/science/article/pii/S1544612319304714
3. Aysal, T.C., Yildiz, M.E., Sarwate, A.D., Scaglione, A.: Broadcast gossip algorithms for consensus. IEEE Transactions on Signal processing **57**(7), 2748–2761 (2009)
4. Babel, K., Chursin, A., Danezis, G., Kokoris-Kogias, L., Sonnino, A.: Mysticeti: Low-latency dag consensus with fast commit path. arXiv preprint arXiv:2310.14821 (2023)
5. Bacho, R., Loss, J., Tessaro, S., Wagner, B., Zhu, C.: Twinkle: Threshold signatures from ddh with full adaptive security. In: Annual International Conference on the Theory and Applications of Cryptographic Techniques. pp. 429–459. Springer (2024)
6. Backes, M., Berrang, P., Hanzlik, L., Pryvalov, I.: A framework for constructing single secret leader election from mpc. In: European Symposium on Research in Computer Security. pp. 672–691. Springer (2022)
7. BAJPAI, P.: Cryptocurrency futures: Definition and how they work on exchanges. Investopedia https://www.investopedia.com/articles/investing/012215/how-invest-bitcoin-exchange-futures.asp (2023)
8. Balasubramanian, K.: Security of the secp256k1 elliptic curve used in the bitcoin blockchain. Indian Journal of Cryptography and Network Security (IJCNS) **4**(1), 1–5 (2024)
9. Bellare, M., Neven, G.: Multi-signatures in the plain public-key model and a general forking lemma. In: Proceedings of the 13th ACM conference on Computer and communications security. pp. 390–399 (2006)
10. Blokh, C., Makriyannis, N., Peled, U.: Efficient asymmetric threshold ecdsa for mpc-based cold storage. Cryptology ePrint Archive (2022)
11. Boldyreva, A.: Threshold signatures, multisignatures and blind signatures based on the gap-diffie-hellman-group signature scheme. In: International Workshop on Public Key Cryptography. pp. 31–46. Springer (2002)
12. Boneh, D., Gennaro, R., Goldfeder, S.: Using level-1 homomorphic encryption to improve threshold DSA signatures for bitcoin wallet security. In: LATINCRYPT. Lecture Notes in Computer Science, vol. 11368, pp. 352–377. Springer (2017)
13. Bouvier, C., Castagnos, G., Imbert, L., Laguillaumie, F.: I want to ride my bicycl: Bicycl implements cryptography in class groups. Journal of Cryptology **36**(3), 17 (2023)

14. Bracha, G., Toueg, S.: Resilient consensus protocols. In: Proceedings of the second annual ACM symposium on Principles of distributed computing. pp. 12–26 (1983)
15. Bracha, G., Toueg, S.: Asynchronous consensus and broadcast protocols. Journal of the ACM (JACM) **32**(4), 824–840 (1985)
16. Brandão, L., Davidson, M.: Notes on threshold eddsa/schnorr signatures (2023)
17. Braun, L., Castagnos, G., Damgård, I., Laguillaumie, F., Melissaris, K., Orlandi, C., Tucker, I.: An improved threshold homomorphic cryptosystem based on class groups. Cryptology ePrint Archive (2024)
18. Braun, L., Damgård, I., Orlandi, C.: Secure multiparty computation from threshold encryption based on class groups. In: Annual International Cryptology Conference. pp. 613–645. Springer (2023)
19. Buterin, V., et al.: Ethereum white paper. GitHub repository **1**, 22–23 (2013)
20. Canetti, R., Gennaro, R., Goldfeder, S., Makriyannis, N., Peled, U.: UC non-interactive, proactive, threshold ECDSA with identifiable aborts. In: CCS. pp. 1769–1787 (2020)
21. Castagnos, G., Catalano, D., Laguillaumie, F., Savasta, F., Tucker, I.: Bandwidth-efficient threshold EC-DSA. In: PKC. Lecture Notes in Computer Science, vol. 12111, pp. 266–296. Springer (2020)
22. Castagnos, G., Catalano, D., Laguillaumie, F., Savasta, F., Tucker, I.: Bandwidth-efficient threshold ec-dsa revisited: Online/offline extensions, identifiable aborts proactive and adaptive security. Theoretical Computer Science **939**, 78–104 (2023)
23. Castagnos, G., Laguillaumie, F.: Linearly homomorphic encryption from. In: Cryptographers' Track at the RSA Conference. pp. 487–505. Springer (2015)
24. Chatzigiannis, P., Baldimtsi, F., Chalkias, K.: Sok: Blockchain light clients. In: International Conference on Financial Cryptography and Data Security. pp. 615–641. Springer (2022)
25. Chen, M., Hazay, C., Ishai, Y., Kashnikov, Y., Micciancio, D., Riviere, T., Shelat, A., Venkitasubramaniam, M., Wang, R.: Diogenes: lightweight scalable rsa modulus generation with a dishonest majority. In: 2021 IEEE Symposium on Security and Privacy (SP). pp. 590–607. IEEE (2021)
26. Cheon, J.H.: Security analysis of the strong diffie-hellman problem. In: Annual International Conference on the Theory and Applications of Cryptographic Techniques. pp. 1–11. Springer (2006)
27. Chris, C., David, S.: Lit protocol whitepaper. Github Repository https://github.com/LIT-Protocol/whitepaper (2024)
28. Chu, H., Gerhart, P., Ruffing, T., Schröder, D.: Practical schnorr threshold signatures without the algebraic group model. In: Annual International Cryptology Conference. pp. 743–773. Springer (2023)
29. Cohen, R., Doerner, J., Kondi, Y., Shelat, A.: Secure multiparty computation with identifiable abort via vindicating release. In: Annual International Cryptology Conference. pp. 36–73. Springer (2024)
30. Copper: Wallets as a service. Whitepaper https://copper.co/en/products/wallets-as-a-service (2021)
31. Cramer, R., Damgård, I., Nielsen, J.B.: Multiparty computation from threshold homomorphic encryption. In: Advances in Cryptology—EUROCRYPT 2001: International Conference on the Theory and Application of Cryptographic Techniques Innsbruck, Austria, May 6–10, 2001 Proceedings 20. pp. 280–300. Springer (2001)
32. Dahari-Garbian, H., Nof, A., Parker, L.: Trout: Two-round threshold ecdsa from class groups. In: Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security. pp. 380–393 (2025)

33. Dalskov, A.P.K., Orlandi, C., Keller, M., Shrishak, K., Shulman, H.: Securing DNSSEC keys via threshold ECDSA from generic MPC. In: ESORICS. vol. 12309, pp. 654–673. Springer (2020)
34. Damgård, I., Jakobsen, T.P., Nielsen, J.B., Pagter, J.I., Østergård, M.B.: Fast threshold ECDSA with honest majority. In: SCN. LNCS, vol. 12238, pp. 382–400. Springer (2020)
35. Danezis, G., Kokoris-Kogias, L., Sonnino, A., Spiegelman, A.: Narwhal and tusk: a dag-based mempool and efficient bft consensus. In: Proceedings of the Seventeenth European Conference on Computer Systems. pp. 34–50 (2022)
36. Developer, B.: Developer guides – locktime and sequence number. Bitcoin Developer https://developer.bitcoin.org/devguide/transactions.html (2023)
37. Doerner, J., Kondi, Y., Lee, E., Shelat, A.: Threshold ECDSA from ECDSA assumptions: The multiparty case. In: SP 2019. pp. 1051–1066. IEEE (2019)
38. Doerner, J., Kondi, Y., Lee, E., Shelat, A.: Threshold ECDSA in three rounds. IACR Cryptol. ePrint Arch. p. 765 (2023)
39. Dwork, C., Lynch, N., Stockmeyer, L.: Consensus in the presence of partial synchrony. J. ACM **35**(2), 288–323 (apr 1988). https://doi.org/10.1145/42282.42283, https://doi.org/10.1145/42282.42283
40. Ergezer, S., Kinkelin, H., Rezabek, F.: A survey on threshold signature schemes. Network **49** (2020)
41. Etherscan: Ethereum node tracker. Etherscan https://etherscan.io/nodetracker (2024)
42. Fiat, A., Shamir, A.: How to prove yourself: Practical solutions to identification and signature problems. In: Conference on the theory and application of cryptographic techniques. pp. 186–194. Springer (1986)
43. Fireblocks: A guide to digital asset wallets and service providers. Whitepaper https://www.fireblocks.com/a-guide-to-digital-asset-wallets-and-service-providers/ (2023)
44. Fischer, M.J., Lynch, N.A., Paterson, M.S.: Impossibility of distributed consensus with one faulty process. Journal of the ACM (JACM) **32**(2), 374–382 (1985)
45. Fischlin, M.: Communication-efficient non-interactive proofs of knowledge with online extractors. In: Annual International Cryptology Conference. pp. 152–168. Springer (2005)
46. Friedman, O., Marmor, A., Mutzari, D., Sadika, O., Scaly, Y.C., Spiizer, Y., Yanai, A.: 2pc-mpc: Emulating two party ecdsa in large-scale mpc. Cryptology ePrint Archive (2024)
47. Friedman, O., Marmor, A., Mutzari, D., Scaly, Y.C., Spiizer, Y., Yanai, A.: Tiresias: Large scale, maliciously secure threshold paillier. Cryptology ePrint Archive (2023)
48. Gagol, A., Kula, J., Straszak, D., Swietek, M.: Threshold ECDSA for decentralized asset custody. IACR Cryptol. ePrint Arch. p. 498 (2020)
49. Gennaro, R., Goldfeder, S.: Fast multiparty threshold ECDSA with fast trustless setup. IACR Cryptol. ePrint Arch. p. 114 (2019)
50. Gennaro, R., Goldfeder, S., Narayanan, A.: Threshold-optimal DSA/ECDSA signatures and an application to bitcoin wallet security. In: International Conference on Applied Cryptography and Network Security. pp. 156–174. Springer (2016)
51. Gentry, C., Halevi, S., Krawczyk, H., Magri, B., Nielsen, J.B., Rabin, T., Yakoubov, S.: Yoso: You only speak once: Secure mpc with stateless ephemeral roles. In: Annual International Cryptology Conference. pp. 64–93. Springer (2021)
52. Gray, E.: Governor. GitHub Repository https://github.com/wormhole-foundation/wormhole/blob/main/whitepapers/0007_governor.md (2022)

53. Groth, J., Shoup, V.: On the security of ecdsa with additive key derivation and presignatures. In: Annual International Conference on the Theory and Applications of Cryptographic Techniques. pp. 365–396. Springer (2022)
54. Hu, B., Zhang, Z., Liu, J., Liu, Y., Yin, J., Lu, R., Lin, X.: A comprehensive survey on smart contract construction and execution: paradigms, tools, and systems. Patterns **2**(2) (2021)
55. Illia, P., Alexander, S.: The near white paper. Whitepaper https://pages.near.org/papers/the-official-near-white-paper/ (2021)
56. Komlo, C., Goldberg, I.: Frost: flexible round-optimized schnorr threshold signatures. In: Selected Areas in Cryptography: 27th International Conference, Halifax, NS, Canada (Virtual Event), October 21-23, 2020, Revised Selected Papers 27. pp. 34–65. Springer (2021)
57. KZen, T.: Bitcoin wallet powered by two-party ecdsa - extended abstract. Whitepaper https://github.com/ZenGo-X/gotham-city/blob/master/white-paper/ (2019)
58. KZen, T.: Radical new infrastructure for digital asset ownership and blockchain interoperability. Whitepaper https://www.qredo.com/qredo-white-paper.pdf (2019)
59. Labs, T.: Web3 auth. Github Repository https://github.com/web3auth (2024)
60. Leonardos, S., Reijsbergen, D., Piliouras, G.: Weighted voting on the blockchain: Improving consensus in proof of stake protocols. International Journal of Network Management **30**(5), e2093 (2020)
61. Lindell, Y.: Fast secure two-party ECDSA signing. In: Annual International Cryptology Conference. pp. 613–644. Springer (2017)
62. Lindell, Y.: Digital asset management with mpc. Whitepaper https://www.coinbase.com/blog/digital-asset-management-with-mpc-whitepaper (2023)
63. Lindell, Y., Nof, A.: Fast secure multiparty ECDSA with practical distributed key generation and applications to cryptocurrency custody. In: CCS. pp. 1837–1854. ACM (2018)
64. Luongo, M., Pon, C.: The keep network: A privacy layer for public blockchains. Keep Netw., Rep (2018)
65. Lyu, Y., Li, Z., Zhou, H.S., Deng, X.: Threshold ecdsa in two rounds. In: Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security. pp. 2937–2950 (2025)
66. Mohanta, B.K., Panda, S.S., Jena, D.: An overview of smart contract and use cases in blockchain technology. In: 2018 9th international conference on computing, communication and networking technologies (ICCCNT). pp. 1–4. IEEE (2018)
67. Nakamoto, S.: Bitcoin whitepaper. URL: https://bitcoin.org/bitcoin.pdf-(:17.07.2019) **9**, 15 (2008)
68. Paillier, P.: Public-key cryptosystems based on composite degree residuosity classes. In: Advances in Cryptology-EUROCRYPT 99: International Conference on the Theory and Application of Cryptographic Techniques Prague, Czech Republic, May 2-6, 1999 Proceedings 18. pp. 223–238. Springer (1999)
69. Pedersen, T.P.: Non-interactive and information-theoretic secure verifiable secret sharing. In: Annual international cryptology conference. pp. 129–140. Springer (1991)
70. Ruffing, T., Ronge, V., Jin, E., Schneider-Bensch, J., Schröder, D.: Roast: robust asynchronous schnorr threshold signatures. In: Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security. pp. 2551–2564 (2022)

71. Sebastião, H., Godinho, P.: Bitcoin futures: An effective tool for hedging cryptocurrencies. Finance Research Letters **33**, 101230 (2020). https://doi.org/https://doi.org/10.1016/j.frl.2019.07.003, https://www.sciencedirect.com/science/article/pii/S1544612319301849
72. Shoup, V.: A theoretical take on a practical consensus protocol. Cryptology ePrint Archive (2024)
73. Shoup, V., Smart, N.P.: Lightweight asynchronous verifiable secret sharing with optimal resilience. Journal of Cryptology **37**(3), 27 (2024)
74. Spiegelman, A., Giridharan, N., Sonnino, A., Kokoris-Kogias, L.: Bullshark: Dag bft protocols made practical. In: Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security. pp. 2705–2718 (2022)
75. Stafford, V.: Zero trust architecture. NIST special publication **800**, 207 (2020)
76. Team, D., et al.: The internet computer for geeks. Cryptology ePrint Archive (2022)
77. Thorchain: Thorchain: A decentralised liquidity network. Whitepaper https://github.com/thorchain/Resources/blob/master/Whitepapers/THORChain-Whitepaper-May2020.pdf (2020)
78. Tolmach, P., Li, Y., Lin, S.W., Liu, Y., Li, Z.: A survey of smart contract formal specification and verification. ACM Computing Surveys (CSUR) **54**(7), 1–38 (2021)
79. Wang, S., Yuan, Y., Wang, X., Li, J., Qin, R., Wang, F.Y.: An overview of smart contract: architecture, applications, and future trends. In: 2018 IEEE Intelligent Vehicles Symposium (IV). pp. 108–113. IEEE (2018)
80. Wilhelmi, F., Giupponi, L., Dini, P.: Analysis and evaluation of synchronous and asynchronous flchain. Computer Networks **218**, 109390 (2022)
81. Wong, H.W.H., Ma, J.P.K., Yin, H.H.F., Chow, S.S.M.: Real threshold ECDSA. In: NDSS. The Internet Society (2023)
82. Wormhole: Wormhole incident report — 02/02/22. Wormholecrypto Medium https://wormholecrypto.medium.com/wormhole-incident-report-02-02-22-ad9b8f21eec6 (2022)
83. Wuille, P.: Hierarchical deterministic wallets. GitHub Repository https://github.com/bitcoin/bips/blob/master/bip-0032.mediawiki (2013)
84. Xue, H., Au, M.H., Xie, X., Yuen, T.H., Cui, H.: Efficient online-friendly two-party ECDSA signature. IACR Cryptol. ePrint Arch. p. 318 (2022)
85. Zyskind, G., Yanai, A., Pentland, A.S.: Unstoppable wallets: Chain-assisted threshold ECDSA and its applications. IACR Cryptol. ePrint Arch. p. 832 (2023)

A Applications

In this section we will refer to application of both the ECDSA protocol presented in the main paper and the Schnorr protocol presented in Appendix G.

Digital Wallets for *any* Blockchain. The address of a wallet is typically derived from its public key. Therefore, public keys generated on our blockchain can be used as accounts on other blockchains, e.g. on Bitcoin or Ethereum. A client can make a transaction by initiating a signing protocol with our blockchain. With this motivation in mind, we support both ECDSA and EdDSA signatures in our protocols, supporting nearly all blockchains to date.[3]

[3] In case the standard for digital signatures in cryptocurrency will change, protocols for the new signature schemes should be developed. Nevertheless, the system design could potentially stay the same.

Future transactions. A *future transaction* is a signed transaction submitted by a client, that is only executed *when and if* a certain predetermined client-specified condition is met. Since in our signing protocol, the client essentially sends an encryption of the signature, the blockchain has the capability of *delaying transaction execution*. Namely, validators would send their decryption shares only when and if they observe that the associated condition is met. One type of such transactions are *timelocked transactions* [36], wherein the transaction is executed only after a specified period of time/blocks.

Bridging Blockchains. We identify *bridges* as an appealing special case of future transactions. Clients can transfer, for instance, BTC in exchange for Ether, as follows. Client A would send a future transaction transferring x_A BTC to liquidity pool P, and condition it on receiving at least y_P Ether from the liquidity pool. This way, we can bridge Blockchains, without relying on implementation of smart contracts or light-clients on either of them.

Cryptocurrency Futures. Cryptocurrency futures [2,71] are futures contracts that allow investors to place bets on a cryptocurrency's future price without owning the cryptocurrency. For instance, a trader might enter a futures contract to buy Bitcoin at a fixed price at a specific future date; if the price of Bitcoin increases, they can profit from the difference, even though they never actually held any Bitcoin. This tool is often used to hedge against volatility, allowing investors to protect themselves from adverse price movements. To date, these contracts are traded on the Chicago Mercantile Exchange (CME) and cryptocurrency exchanges and Chicago Board Options Exchange (CBOE) [7]. Nevertheless, this is a Web2 solution. Instead, such a market can be implemented by an on-chain market that trades on future transactions that are dictated by such future contracts.

Recovery. Clients can produce a future transaction that transfers all of their funds or wallet to some other entity. This can be used for recovery of the wallet and its funds in case the client share of the key is lost, or compromised. The condition for initiating this recovery transaction can be signing a transaction with another wallet whose client share is more protected, or a combination of signatures from a subset of whitelist wallets with a certain threshold, counting on "friends", as an example.

Wallet Transfer. Wallets may have an *intrinsic value*.[4] Therefore, transferring ownership of a wallet can be of independent interest, and is not necessarily equivalent to transferring all of its funds. In our system, this can be achieved by transferring the client share, in a publicly verifiable manner. The blockchain can then associate the wallet with the receiver client. One can also consider lending a wallet, where the receiver has to pay a bill every couple of blocks in order to keep holding or using the wallet.

[4] For instance, for NFT creators and artists, having a verification badge on Instagram can enhance their credibility. This verification badge is a property of the account itself, regardless of its balance.

Decentralized Autonomous Organization (DAO). By supporting BIP32 [83], DAOs can manage a common wallet associated with a master public key, and delegate permissions on it. For instance, the master key may have the capability of updating the wallet policy, its child could transfer funds to any address, but up to a certain maximal amount, and each of its children can transfer funds to certain whitelists of clients. In particular, (only) the owner of the master key can add whitelists, change the limit on transaction amount, or delegate these abilities.

These applications demonstrate the versatility of our system, enabling secure cross-chain wallet management, future transaction automation, bridging between blockchains without external dependencies, wallet recovery, asset transfer, and decentralized autonomous organizations (DAOs). We believe this showcases its broad potential to address both individual and organizational needs across a variety of blockchain ecosystems.

B Extended Preliminaries

B.1 Model

Communication Functionalities Following are the detailed functionalities $\mathcal{F}_{\mathsf{broadcast}}$, $\mathcal{F}_{\mathsf{global\text{-}broadcast}}$, and $\mathcal{F}_{\mathsf{ACS}}$ from the communication model (Section 3).

B.2 Security Definitions for ZKPs

Definition B.1. *We say that a protocol between two PPT machines $\mathcal{P}$ and $\mathcal{V}$ is a Zero-Knowledge Proof for relation R if it is* Complete, Sound *and* Zero-Knowledge *as defined below. We say that the protocol is a Zero-Knowledge Proof of Knowledge if it is also* Knowledge sound.

Definition B.2. *The protocol will be called complete if for $(x, w) \in R[\mathsf{pp}]$ if $\mathcal{V}(\mathsf{pp}, r) \leftrightarrow \mathcal{P}(\mathsf{pp}, x, w)$ outputs 1 with overwhelming probability with respect to the random tape r of V.*

Definition B.3. *If $(x, w) \notin R[\mathsf{pp}]$ then for every PPT algorithm $\mathcal{P}^*$ $\mathcal{V}(\mathsf{pp}, r) \leftrightarrow \mathcal{P}^*(\mathsf{pp}, x, w)$ returns 0 with overwhelming probability with respect to the random tape r of V*

As all of our ZK protocols will be non-interactive it is enough to discuss Honest Verifier Zero-Knowledge (HVZK).

Definition B.4. *A protocol will be called HVZK if there exists a PPT simulator $\mathcal{S}$ such that $\mathsf{View}_{\mathcal{V}}(S(\mathsf{pp}, x) \leftrightarrow \mathcal{V}(\mathsf{pp}, r))$ is computationally indistinguishable from $\mathsf{View}_{\mathcal{V}}(\mathcal{V}(\mathsf{pp}, r) \leftrightarrow \mathcal{P}(\mathsf{pp}, x, w))$ for $(x, w) \in R[\mathsf{pp}]$.*

Definition B.5. *A protocol will be called Knowledge-Sound if there exists a PPT simulator $\mathcal{S}$ such that $\mathcal{S}(\mathsf{pp}, x)$ with black box access to a PPT prover $\mathcal{P}^*$ which outputs correct proofs with non-negligible probability outputs w such that $(x, w) \in R[\mathsf{pp}]$. A protocol will be called UC Knowledge-Sound or UC Extractable if the above simulator does not rewind $\mathcal{P}^*$.*

B.3 Statement Aggregation Protocol

In several sub-protocols, parties in B must send a *single* aggregated message to a client (via $\mathcal{F}_{\mathsf{global\text{-}broadcast}}$), even though each validator locally computes its own contribution. We therefore use the following generic statement-aggregation primitive.

Fix an access structure Γ_B over B and a language $L[\mathsf{pp}]$ of well-formed statements. Each party B_i holds a statement–witness pair $(\boldsymbol{Y}_i; \boldsymbol{w}_i)$ where $\boldsymbol{Y}_i$ lies in an abelian group (or module) and addition $\sum_i \boldsymbol{Y}_i$ is well-defined (componentwise, when $\boldsymbol{Y}_i$ is a vector). Parties prove membership in $L[\mathsf{pp}]$ using a NIZK proof.

At a high level, the protocol: (i) broadcasts $(\boldsymbol{Y}_i, \pi_i)$ from each party, (ii) locally validates all received proofs, (iii) uses $\mathcal{F}_{\mathsf{ACS}}$ to agree on an authorized subset of indices whose proofs validated, and (iv) outputs the aggregated statement as the sum over that subset.

B.4 Homomorphic Commitment Schemes

Definition B.6. *A non-interactive commitment scheme consists of a pair of PPT algorithms* $(\mathsf{setup}, \mathsf{Com})$*. The setup algorithm* $\mathsf{pp} \leftarrow \mathsf{setup}$ *outputs the spaces* $\mathcal{M}_{pp}$*,* $\mathcal{R}_{pp}$ *and* $\mathcal{C}_{pp}$ *along with any other public values needed for the scheme. The commitment algorithm* Com_{pp} *defines a function* $\mathcal{M}_{pp} \times \mathcal{R}_{pp} \to \mathcal{C}_{pp}$*. For a message* $m \in \mathcal{M}_{pp}$*, the algorithm draws* ρ*, and computes commitment* $C = \mathsf{Com}_{pp}(m; \rho)$*. Whenever the public parameters are clear from the context we write* Com *instead of* Com_{pp}*. Commitment schemes may also be deterministic in which case one can think of* ρ *as a trivial group and write* $\mathsf{Com}_{pp}(m)$*.*

A commitment scheme has two properties, *hiding* and *binding*, in the case of Pedersen Commitment [69] they are perfectly hiding and computationally binding which is defined as follows:

- Perfectly Hiding. For every (unbounded) adversary $\mathcal{A}$, every $\mathsf{pp} \leftarrow \mathsf{setup}(\cdot)$ and every $m_0, m_1 \in \mathcal{M}_{\mathsf{pp}}$

$$\Pr[\mathcal{A}(m_0, m_1, \mathsf{Com}(m_b, \rho) = b] = \frac{1}{2}$$

  where the probability is under a uniform choice of $b \in \{0, 1\}$ and $\mathcal{M}_{\mathsf{pp}} \leftarrow \mathcal{R}_{\mathsf{pp}}$.
- Computationally Binding. For every PPT adversary $\mathcal{A}$ and every $\mathsf{pp} \leftarrow \mathsf{Setup}(\cdot)$,

  $$\Pr[(m_0, m_1, \rho_0, \rho_1) \leftarrow \mathcal{A}(\mathsf{pp}) : m_0 \neq m_1 \wedge \mathsf{Com}(m_0, \rho_0) = \mathsf{Com}(m_1, \rho_1)] \leq \mu(\kappa)$$

  Where the probability is under the random coins of $\mathcal{A}$.

B.5 Formal Definitions For AHE and TAHE

AHE: The definition for AHE below is inspired by [25]

Definition 3 (AHE) *An* additively homomorphic encryption scheme *is associated with an ensemble* $\{\mathcal{K}_\kappa\}_\kappa$ *and consists of four polynomial time algorithms:* $\mathsf{AHE} = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec}, \mathsf{Add})$ *specified as follows:*

- $\mathsf{Gen}(1^\kappa, \mathsf{aux}) \to (\mathsf{pk}; \mathsf{sk})$. *A probabilistic algorithm that is given a security parameter* 1^κ *and possibly some auxiliary information* aux *and samples a key-pair* $(\mathsf{pk}, \mathsf{sk})$ *from* $\mathcal{K}_\kappa$. *In the following, we assume that the resulting* pk *contains the description of the security parameter* 1^κ, *the auxiliary information* aux, *as well as the plaintext, randomness and ciphertext spaces* $\mathcal{P}_{pk}$, $\mathcal{R}_{pk}$ *and* $\mathcal{C}_{pk}$, *respectively, where* $\mathcal{P}_{pk}$ *is a* $\mathbb{Z}$*-module.*
  In addition, for the purposes in this paper we require an AHE scheme to be associated with a language $L_{\mathsf{GenAHE}}[\kappa] = \{\mathsf{pk}; \mathsf{aux} \mid (\mathsf{pk}; \mathsf{sk}) = \mathsf{Gen}(1^\kappa, \mathsf{aux})\}$ *(see Eq. 13) that expresses the fact that* pk *was generated correctly.*
- $\mathsf{Enc}(\mathsf{pk}, \mathsf{pt}; \eta_{\mathsf{enc}}) \to \mathsf{ct}$. *A deterministic algorithm that on input a public key* pk, *a plaintext* $\mathsf{pt} \in \mathcal{P}_{pk}$ *and randomness* $\eta_{\mathsf{enc}} \in \mathcal{R}_{pk}$, *outputs a ciphertext* $\mathsf{ct} \in \mathcal{C}_{pk}$. *We define* $\mathsf{Enc}(\mathsf{pk}, \mathsf{pt})$ *as a probabilistic algorithm that first uniformly samples* $\eta_{\mathsf{enc}} \in \mathcal{R}_{pk}$ *and then outputs* $\mathsf{ct} = \mathsf{Enc}(\mathsf{pk}, \mathsf{pt}; \eta_{\mathsf{enc}}) \in \mathcal{C}_{pk}$.
- $\mathsf{Dec}(\mathsf{sk}, \mathsf{ct}) \to \mathsf{pt}$. *A deterministic algorithm that on input a secret key* sk *and a ciphertext* $\mathsf{ct} \in \mathcal{C}_{pk}$ *outputs a plaintext* $\mathsf{pt} \in \mathcal{P}_{pk}$.
- $\mathsf{Add}(\mathsf{pk}, \mathsf{ct}_1, \mathsf{ct}_2) \to \mathsf{ct}_3$. *A deterministic algorithm that on input a public key* pk *and two ciphertexts* $\mathsf{ct}_1, \mathsf{ct}_2 \in \mathcal{C}_{pk}$ *outputs a ciphertext* $\mathsf{ct}_3 \in \mathcal{C}_{pk}$ *such that if* $\mathsf{ct}_1 = \mathsf{Enc}(\mathsf{pk}, m_1; \eta_1)$ *and* $\mathsf{ct}_2 = \mathsf{Enc}(\mathsf{pk}, m_2; \eta_2)$ *then* $\mathsf{ct}_3 = \mathsf{Enc}(\mathsf{pk}, m_1 + m_2; \eta_1 + \eta_2)$, *where* $m_1 + m_2$ *and* $\eta_1 + \eta_2$ *are over* $\mathcal{P}_{pk}$ *and* $\mathcal{R}_{pk}$, *respectively. Imposing correctness will ensure that* Add *is a homomorphic addition. Note that efficient homomorphic scalar multiplication is implied by the* Add *operation (e.g., via double-and-add).*

In addition, we denote homomorphic addition of two ciphertexts ct_1 and ct_2 by $\mathsf{ct}_1 \oplus \mathsf{ct}_2$ and a multiplication of a ciphertext ct by a scalar α by $\alpha \odot \mathsf{ct}$.

Every affine function can be efficiently computed homomorphically by an algorithm $\mathsf{Eval}(\mathsf{pk}, f, \mathsf{ct}_1, \ldots, \mathsf{ct}_\ell; \eta_{\mathsf{eval}})$. Whenever we omit the randomness in $\mathsf{Eval}(\mathsf{pk}, f, \mathsf{ct}_1, \ldots, \mathsf{ct}_\ell)$, we refer to the process of sampling a randomness $\eta_{\mathsf{eval}} \leftarrow \{0,1\}^{\mathsf{poly}(\kappa)}$ and running $\mathsf{Eval}(\mathsf{pk}, f, \mathsf{ct}_1, \ldots, \mathsf{ct}_\ell; \eta_{\mathsf{eval}})$. Given a public key pk, an affine function $f(x_1, \ldots, x_\ell) = a_0 + \sum_{i=1}^{\ell} a_i x_i$ (with $a_i \in \mathbb{Z}$ and $\ell, \|a_i\|_2 \in \mathsf{poly}(\kappa)$ for all $0 \le i \le \ell$) and ℓ ciphertexts $\mathsf{ct}_1, \ldots, \mathsf{ct}_\ell \in \mathcal{R}_{pk}$, Eval outputs a ciphertext ct.

Definition 4 (Correctness) AHE *is* correct *if for every* κ, *every* $t \in \mathsf{poly}(\kappa)$, *every* ℓ*-ary affine function* f *as above, and every plaintexts* $\mathsf{pt}_1, \ldots, \mathsf{pt}_t \in \mathcal{P}_{pk}$,

$$\Pr[\mathsf{Dec}(sk, \mathsf{Eval}(pk, f, (\mathsf{ct}_1, \ldots, \mathsf{ct}_\ell))) = f(\mathsf{pt}_1, \ldots, \mathsf{pt}_\ell)] \ge 1 - \mathsf{neg}(\kappa)$$

where $(pk, sk) \leftarrow \mathsf{Gen}(1^\kappa)$ *and* $\mathsf{ct}_i \leftarrow \mathsf{Enc}(pk, \mathsf{pt}_i)$, *and the probability is over the coins used by algorithms* Gen, Enc *and* Eval.

Note that for correctness, algorithm Eval may simply output $\mathsf{ct} = \mathsf{Enc}(pk, a_0) \bigoplus_{i=1}^{\ell} a_i \odot \mathsf{ct}_i$, however, such an algorithm may not satisfy *Linear Circuit Privacy*(see below).

Definition 5 (Semantic security) *AHE is* semantically secure *if for every PPT adversary $\mathcal{A}$ there exists a PPT algorithm $\mathcal{S}$ such that for every pair of PPTs $f, h : \{0,1\}^* \to \{0,1\}^*$, and every $\mathsf{pt} \in \mathcal{P}_{pk}$:*

$$\begin{aligned} \mid \Pr[\mathcal{A}(1^\kappa, \mathsf{Enc}(\mathsf{pk}, \mathsf{pt}), 1^{|\mathsf{pt}|}, h(1^\kappa, \mathsf{pt})) = f(1^\kappa, \mathsf{pt})] \\ - \Pr[\mathcal{S}(1^\kappa, 1^{|\mathsf{pt}|}, h(1^\kappa, \mathsf{pt})) = f(1^\kappa, \mathsf{pt})] \mid < \mathsf{neg}(\kappa) \end{aligned} \tag{1}$$

where $(pk, sk) \leftarrow \mathsf{Gen}(1^\kappa)$ and h provides the auxiliary information function in possession of the adversary. The probability is over the random coins of Gen and Enc.

Definition 6 (Linear Circuit Privacy) *AHE has* Linear Circuit Privacy *if there exists a PPT algorithm $\mathcal{S}_{\mathsf{Eval}}$ such that for every PPT adversary $\mathcal{A}$, every ℓ-ary affine function f as above, every $(\mathsf{pk}, \mathsf{sk})$ and every plaintexts $\mathsf{pt}_1, \ldots, \mathsf{pt}_\ell \in \mathcal{P}_{pk} \cup \{\perp\}$ and randomness $\eta_1, \ldots, \eta_\ell$,*

$$\begin{aligned} &\mid \Pr[\mathcal{A}(1^\kappa, \mathsf{pk}, \mathsf{sk}, \{\mathsf{pt}_i, \eta_i\}_{i=1}^{\ell}, \mathsf{Eval}(\mathsf{pk}, f, (\mathsf{ct}_1, \ldots, \mathsf{ct}_\ell))) = 1] \\ &- \Pr[\mathcal{A}(1^\kappa, \mathsf{pk}, \mathsf{sk}, \{\mathsf{pt}_i, \eta_i\}_{i=1}^{\ell}, \mathcal{S}_{\mathsf{Eval}}(\mathsf{pk}, f(\mathsf{pt}_1, \ldots, \mathsf{pt}_\ell))) = 1]| \leq \mathsf{neg}(\kappa) \end{aligned} \tag{2}$$

where $\mathsf{ct}_i \in \mathcal{C}_{pk}$ and $\mathsf{pt}_i = \mathsf{Dec}(\mathsf{sk}, \mathsf{ct}_i)$, and the probability is over the coins used by Eval and the random coins of $\mathcal{A}$ and $\mathcal{S}$.

Remark B.1. We build upon the definition from [46] and extend it to cases in which not every element of the ciphertext space is a valid encryption. Indeed in their work which use the classical Paillier encryption [68] this distinction is void. But it is not the case for Elgamal style encryption such as Class-Group [23] nor for lattice based AHE. See Appendix F for further discussion.

TAHE: A TAHE scheme is defined by a tuple of algorithms $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Add}, \mathsf{TDec}, \mathsf{Rec})$. $\mathsf{Gen}, \mathsf{Enc}, \mathsf{Add}$ works similarly to the definition of AHE while TDec is an operation that each party runs locally on the ciphertext and generate a decryption share. Collecting $t+1$ such decryption shares as inputs to Rec outputs a correct decryption.

Let us first formalize the TAHE algorithms and its correctness. The security properties of the TAHE scheme will be captured via an ideal functionality 5 and the AHE security definition.

Definition 7 (TAHE) *Threshold additively homomorphic encryption scheme is associated with an ensemble $\{\mathcal{K}_\kappa\}_\kappa$ and consists of five polynomial time algorithms $\mathsf{TAHE} = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Add}, \mathsf{TDec}, \mathsf{Rec})$ specified as follows:*

- $(\mathsf{pk}, \mathsf{vk}_i; [\mathsf{sk}]_i) \leftarrow \mathsf{Gen}(1^\kappa, i, t, n, \mathsf{aux})$. *A probabilistic (possibly interactive) protocol that is given a security parameter 1^κ, the number of parties n, a threshold $1 < t < n$ and possibly some auxiliary information* aux *and samples a key-pair $(\mathsf{pk}, \mathsf{sk}) \in \mathcal{K}_\kappa$ such that $[\mathsf{sk}]_i$ is a sharing on sk and vk_i is a*

verification key. In the following, we assume that the resulting pk *contains the description of the security parameter* 1^κ*, the auxiliary information* aux*, as well as the plaintext, randomness and ciphertext spaces* $\mathcal{P}_{pk}, \mathcal{R}_{pk}, \mathcal{C}_{pk}$*, respectively, where* $\mathcal{P}_{pk}$ *is a* $\mathbb{Z}$*-module.*

- $\mathsf{ct} \leftarrow \mathsf{Enc}(\mathsf{pk}, \mathsf{pt}; \eta)$*. A deterministic algorithm that on input a public key* pk*, a plaintext* $\mathsf{pt} \in \mathcal{P}_{pk}$ *and randomness* $\eta \in \mathcal{R}_{pk}$*, outputs a ciphertext* $\mathsf{ct} \in \mathcal{C}_{pk}$*. We define* $\mathsf{Enc}(\mathsf{pk}, \mathsf{pt})$ *as a probabilistic algorithm that first uniformly samples* $\eta \leftarrow \mathcal{R}_{pk}$ *and then outputs* $\mathsf{ct} = \mathsf{Enc}(\mathsf{pk}, \mathsf{pt}; \eta) \in \mathcal{C}_{pk}$
- $(\mathsf{ds}_i, \pi^i_{\mathsf{ds}}) \leftarrow \mathsf{TDec}(\mathsf{ct}, \mathsf{pk}, \mathsf{vk}_i; [\mathsf{sk}]_i)$*. A deterministic algorithm that on input of a public key* pk *a verification key* vk_i*, a secret key share* $[\mathsf{sk}]_i$ *and a ciphertext* $\mathsf{ct} \in \mathcal{C}_{pk}$ *outputs a decryption share* ds_i *along with a proof of correct generation* π^i_{ds}*.*
- $\mathsf{pt} \leftarrow \mathsf{Rec}(S, \{\mathsf{ds}_i\}_{i \in S})$*. A deterministic algorithm that on input of a set* $S \in \binom{[n]}{t+1}$ *and a tuple of elements* $\{\mathsf{ds}_i\}_{i \in S} \in \mathcal{C}_{pk}^{|S|}$ *outputs a elements* $\mathsf{pt} \in \mathcal{P}_{pk}$*.*
- $\mathsf{ct}_3 \leftarrow \mathsf{Add}(pk, \mathsf{ct}_1, \mathsf{ct}_2) \rightarrow$*. A deterministic algorithm that on input a public key* pk *and two ciphertexts* $\mathsf{ct}_1, \mathsf{ct}_2 \in \mathcal{C}_{pk}$ *outputs a ciphertext* $\mathsf{ct}_3 \in \mathcal{C}_{pk}$*.*

Correctness. TAHE is *correct* if for every κ, every $\ell \in \mathsf{poly}(\kappa)$, every ℓ-ary affine function f, any subset $S \in \binom{[n]}{t+1}$ and every plaintexts $\mathsf{pt}_1, \ldots, \mathsf{pt}_\ell \in \mathcal{P}_{pk}$,

$$\Pr[\mathsf{Rec}(S, \{\mathsf{TDec}([\mathsf{sk}]_i), \mathsf{Eval}(\mathsf{pk}, f, (\mathsf{ct}_j)_{j \in [\ell]})\}_{i \in S}) = f(\mathsf{pt}_1, \ldots, \mathsf{pt}_\ell)] \geq 1 - \mathsf{neg}(\kappa)$$

where $(\mathsf{pk}, \{[\mathsf{sk}]_i\}_{i \in [n]}) \leftarrow \mathsf{Gen}(1^\kappa)$ and $\mathsf{ct}_j \leftarrow \mathsf{Enc}(\mathsf{pk}, \mathsf{pt}_j)$, and the probability is over the coins used by algorithms Gen, Enc and Eval. We will say that a TAHE scheme is secure if it realizes the below ideal functionality 5 $\mathcal{F}_{TAHE}$

C Languages used in our protocols.

The languages in our protocols are listed below:

- *Knowledge of the discrete log of an elliptic-curve point.*

$$L_{\mathsf{DL}}[(\mathbb{G}, G, q)] = \{(P; x) \mid P = x \cdot G\} \tag{3}$$

- *Knowledge of decommitment.*

$$L_{\mathsf{DCom}}[\mathsf{pp}] = \{C; m, \rho \mid C = \mathsf{Com}_{\mathsf{pp}}(m; \rho) \wedge m \in \mathcal{M}_{\mathsf{pp}} \wedge \rho \in \mathcal{R}_{\mathsf{pp}} \wedge C \in \mathcal{C}_{\mathsf{pp}}\} \tag{4}$$

- *Knowledge of plaintext.*

$$L_{\mathsf{pt}}[\mathsf{AHE}, \mathsf{pk}] = \{\mathsf{ct}; m, \eta \mid \mathsf{ct} = \mathsf{AHE.Enc}(\mathsf{pk}; m, \eta) \wedge m \in \mathcal{P}_{pk} \wedge \eta \in \mathcal{R}_{pk} \wedge C \in \mathcal{C}_{pk}\} \tag{5}$$

  If ct is a vector then interpret the language as proving knowledge of plaintext for each element in the vector.
- *Commitment of discrete log.*

$$L_{\mathsf{DComDL}}[\mathsf{pp}, (\mathbb{G}, \boldsymbol{G}, q)] = \{C, Y; \boldsymbol{m}, \rho \mid C = \mathsf{Com}_{\mathsf{pp}}(\boldsymbol{m}; \rho) \wedge Y = \sum_\ell m_\ell \cdot G_\ell\} \tag{6}$$

– *Vector Commitment of discrete log.*

$$L_{\mathsf{VecDComDL}}[\mathsf{pp},(\mathbb{G},\boldsymbol{G},q)]=\{\boldsymbol{C},Y;\boldsymbol{m},\boldsymbol{\rho}\mid\forall\ell\ C_\ell=\mathsf{Com}_{\mathsf{pp}}(m_\ell;\rho_\ell)\wedge Y=\sum_\ell m_\ell\cdot G_\ell\} \tag{7}$$

– *Equality between two commitments with different public parameters.* (Recall that $\mathcal{M}_{\mathsf{pp}}=\mathbb{Z}_q$.)

$$\begin{aligned}L_{\mathsf{DComEQ}}[\mathsf{pp}_1,\mathsf{pp}_2]=\{C_1,C_2;m,\rho_1,\rho_2\mid C_1=&\mathsf{Com}_{\mathsf{pp}_1}(m,\rho_1)\\ \wedge C_2=&\mathsf{Com}_{\mathsf{pp}_2}(m,\rho_2)\}\end{aligned} \tag{8}$$

– *Committed affine evaluation.*

$$\begin{aligned}L_{\mathsf{DComEval}}[\mathsf{pp},pk,\mathsf{ct},q,\ell,s,\kappa]=\{&\mathsf{ct}^{\mathsf{Eval}},\boldsymbol{C};\boldsymbol{a},\rho,\omega,\eta\\ &\mid\mathsf{ct}^{\mathsf{Eval}}=\mathsf{AHE.Eval}(pk,f_{\boldsymbol{a}}(\boldsymbol{x}),\mathsf{ct};\eta)\\ &\wedge\ \boldsymbol{C}=\mathsf{Com}_{\mathsf{pp}}(\boldsymbol{a};\rho)\\ &\wedge\ a_\ell\in[0,q)\ \wedge\ \rho\in\mathcal{R}_{\mathsf{pp}}\ \wedge\ \eta\in\mathcal{R}_{pk}\}\end{aligned} \tag{9}$$

– *AHE encryption of a discrete log.*

$$L_{\mathsf{EncDL}}[pk,(\mathbb{G},G,q)]=\{(\mathsf{ct},X;x,\eta)\mid\mathsf{ct}=\mathsf{AHE.Enc}(pk,x;\eta)\wedge X=x\cdot G\wedge x\in[0,q)\} \tag{10}$$

– *AHE scaling of a discrete log.*

$$\begin{aligned}L_{\mathsf{ScaleDL}}[pk,(\mathbb{G},G,q),\mathsf{ct}_0]=\{(\mathsf{ct}_1,X;x,\eta)\mid\mathsf{ct}_1=\mathsf{AHE.Scale}(pk,\mathsf{ct}_0,x;\eta)\wedge\\ X=x\cdot G\wedge x\in[0,q)\}\end{aligned} \tag{11}$$

– *AHE analogue of a DH tuple; namely, ciphertexts of values x,y and xy.*

$$\begin{aligned}&L_{\mathsf{EncDH}}[pk,\mathsf{ct}_x]=\{(\mathsf{ct}_y,\mathsf{ct}_z;y,\eta_y,\eta_z)\mid\mathsf{ct}_y=\mathsf{AHE.Enc}(pk,y;\eta_y)\\ &\wedge\mathsf{ct}_z=\mathsf{AHE.Eval}(pk,f,\mathsf{ct}_x;0,\eta_z)\text{ s.t. }f(x)=y\cdot x\mod q\wedge y\in[0,q)\}\end{aligned} \tag{12}$$

– *Correct AHE key generation.* The specification of the language depends on the choice of the AHE scheme (Definition B.5) and so it is not mentioned here explicitly.

$$L_{\mathsf{GenAHE}}[\kappa]=\{pk;\mathsf{aux}\mid(pk;sk)=\mathsf{AHE.Gen}(1^\kappa,\mathsf{aux})\} \tag{13}$$

– All languages can be extended to include range claims, this will be done by adding range to the name of the language and including a vector representing the bounds on the witnesses.

D Omitted Security Proofs and Definitions

This section provides the omitted proofs and definitions of the security of our protocols.

D.1 Proof of UC realization of the ECDSA protocol (Theorem 2)

Next, we provide the full proof of Theorem 2.

Proof. Denote by U the set of parties the adversary $\mathcal{A}$ controls. We split the proof for between the cases of malicious A_{pid_A} and honest A_{pid_A}.

Honest A_{pid} Simulation:

We assume WLOG, that all parties in B are corrupted. This is possible since there are no secrets held in our protocol (except the secret share of the threshold encryption whose simulation is encapsulated in the ideal functionality), which means honest parties in B are simulated according to the honest protocol and the simulation remains unchanged.

Key Generation Simulation:

1. Receive $\tilde{X}_0, \tilde{X}_1$ from $\mathcal{F}_{\mathsf{tsig}}^{\mathcal{G}^*_{\mathsf{SE\text{-}ECDSA}}}$.
2. Upon receiving $(\mathsf{global\text{-}broadcast}, \mathsf{sid}, X_{0,B}, X_{1,B})$ from $\mathcal{A}$, verify that $X_{0,B}, X_{1,B} \in \mathbb{G} \setminus \{0\}$. If not, abort.
3. Send $(\mathsf{biaskey}, \mathsf{sid}, \mathsf{pid}, 1, 0, 1, 0)$ to $\mathcal{F}_{\mathsf{tsig}}^{\mathcal{G}^*_{\mathsf{SE\text{-}ECDSA}}}$ and receive (sid, X).
4. Sample $\rho_{\mathsf{pid}_A} \in \mathbb{Z}_q$, set $X_A = X + \rho_{\mathsf{pid}_A} \cdot G$.
5. Sample $\mu_x^0, \mu_x^1, \mu_x^G$ and calculate $X_B = \mu_x^0 \cdot X_{0,B} + \mu_x^1 \cdot X_{1,B} + \mu_x^G \cdot G$.
6. Use the zero-knowledge property of $\Pi_{\mathsf{uc\text{-}zk}}^{L_{\mathsf{DL}}}$ to generate a proof π_{DL} for the statement X_A that passes verification.
7. Program the random oracle on $(\mathsf{sid}, \mathbb{G}, G, q, X_A, X_{0,B}, X_{1,B}, \pi_{\mathsf{DL}})$ to return $(\mu_x^0, \mu_x^1, \mu_x^G)$.
8. Record $(\mathsf{sid}, \mathsf{pid}_A, \rho_{\mathsf{pid}_A}, X_B)$.
9. Send $(\mathsf{broadcast}, \mathsf{sid}, \mathsf{pid}_A, X_A, \pi_{\mathsf{DL}})$ to the adversary.

Presign and Sign Simulation:

1. The simulator first performs the following tasks:
   (a) Extracts the private key sk correspondent to the published public key pk from a zk proof of the public-key validity of the TAHE.
   (b) Upon receiving a message, sends $(\mathsf{pres}, \mathsf{ssid})$ to $\mathcal{F}_{\mathsf{tsig}}^{\mathcal{G}^*_{\mathsf{SE\text{-}ECDSA}}}$ for $i \in U$, and receives $(\mathsf{pres}, \mathsf{sid}, \mathsf{ssid}, K_0, K_1)$ in response.
   (c) Upon receiving $(\mathsf{global\text{-}broadcast}, \mathsf{ssid}, (\mathsf{pid}_A, \mathsf{ct}_\gamma, \mathsf{ct}_{\gamma\cdot\mathsf{key}}, \mathsf{ct}_{\gamma\cdot k_0}, \mathsf{ct}_{\gamma\cdot k_1}, R_{B,0}, R_{B,1}))$ from the adversary, the simulator verifies them as an honest A_{pid_A} would, aborting if the verification fails. This contains only public checks such that the ciphertexts are elements of the ciphertext space and that the points are non-zero points on the elliptic curve.
   (d) Samples $k_A, \alpha, \beta \leftarrow \mathbb{Z}_q$ and computes C_α, C_β, π_α, π_β, C_k, C_{kx}, π_k, $\mathsf{ct}_{\gamma\cdot k}$, R'_B, $\mathsf{ct}_{\alpha,\beta}$, R_B, π_{R_B}, and $\pi_{\mathsf{ct}_{\alpha,\beta}}$ step by step exactly as in the protocol.
   (e) Obtains $(\mu_k^0, \mu_k^1, \mu_k^G)$ from the random oracle on the entire transcript.
   (f) Use the private key sk to decrypt ct_γ, $\mathsf{ct}_{\gamma\cdot k_0}$, $\mathsf{ct}_{\gamma\cdot k_1}$, and $\mathsf{ct}_{\gamma\cdot\mathsf{key}}$ and go to the following step according to the specific case.
2. If one of the following applies:
   – At least one of the descriptions of ct_γ, $\mathsf{ct}_{\gamma\cdot k_0}$, $\mathsf{ct}_{\gamma\cdot k_1}$, and $\mathsf{ct}_{\gamma\cdot\mathsf{key}}$ failed.
   – The value of the plaintext of ct_γ is an encryption of zero.
   – The plaintext of $\mathsf{ct}_{\gamma\cdot k_0}$ divided by the plaintext of ct_γ is not the discrete logs of $R_{B,0}$, or the plaintext of $\mathsf{ct}_{\gamma\cdot k_1}$ divided by the plaintext of ct_γ is not the discrete logs of $R_{B,1}$.

   The simulator does the following:

   (a) Samples $a_1 \leftarrow \mathbb{Z}_q$ and computes R, ct_A, a_2, C_1, and C_2 step by step exactly as in the protocol (except for ct_A that corresponds to the sampled value of a_1).
   (b) Generates π_{kx}, π_R as in the protocol.
   (c) Invokes the zero-knowledge (zk) property of $\Pi_{\mathsf{uc\text{-}zk}}^{L_{\mathsf{DComEval}}}$ to generate a proof π_{DComEval} for the statement $(\mathsf{ct}_A, (C_1, C_2))$, ensuring it passes verification.
   (d) Sends everything to the adversary.
   (e) Aborts if it receives a non-valid signature from the adversary.
3. Otherwise, the simulator does as follows:
   (a) Decrypts ct_γ and $\mathsf{ct}_{\gamma\cdot\mathsf{key}}$ and obtains γ (which is a non-zero) and γ_x, respectively. The simulator computes $x'_B = \frac{\gamma_x}{\gamma}$ and sets $\beta_{\mathsf{key}} = x'_B + \rho_{\mathsf{pid}_A}$.
   (b) Sends $(\mathsf{sign}, \mathsf{msg}, \mathsf{sid}, \mathsf{ssid}, \beta_{\mathsf{key}}, 1, 0, 1, 0)$ to $\mathcal{F}_{\mathsf{tsig}}^{\mathcal{G}^*_{\mathsf{SE\text{-}ECDSA}}}$ and receives a signature $\sigma = (r, s)$ with the public nonce K.
   (c) Sets $R = K$ and computes C_1, C_2, and a_2 according to r.
   (d) Sets $\mathsf{ct}_A = \mathsf{AHE.Scale}(\mathsf{pk}, \mathsf{ct}_{\alpha,\beta}, s; *)$.
   (e) Invokes the zero-knowledge (zk) property of $\Pi_{\mathsf{uc\text{-}zk}}^{L_{\mathsf{DComEval}}}$ and $\Pi_{\mathsf{uc\text{-}zk}}^{L_{\mathsf{DL}}}$ to generate the proofs π_{ct_A}, π_R for the statements $(\mathsf{ct}_A, (C_1, C_2))$ and (C_k, R_B), respectively. Thus ensuring it passes the verifications.
4. The simulator aborts if the forged signature it receives from the adversary is incorrect.

Honest A_{pid} Indistinguishability:

Key Generation: Both in the real and ideal worlds the adversary sees (X_A, π_{DL}) and random oracle returns. The value of X_A is uniformly distributed independently from the values of B in both the real and the simulated executions. The value of π_{DL} is indistinguishable from the zk property of $\Pi_{\mathsf{uc\text{-}zk}}^{L_{\mathsf{DL}}}$.

Presign and Sign: In the presign protocol, only the adversary acts. So there is no difference between the real execution and the simulation.

In either one of the cases described in Step 2 the adversary does not expect to obtain a signature. The only differences between the real execution and the simulation are the zero-knowledge (zk) proofs and the value of a_1. The zk proofs are indistinguishable due to the zk properties. In all these cases and in both executions, if a_1 can be extracted by the adversary, then the discrete log of R_B is unknown to the adversary:

- If one of the decryptions of ct_γ, $\mathsf{ct}_{\gamma\cdot k_0}$, and $\mathsf{ct}_{\gamma\cdot k_1}$ failed then due to the secure random evaluation of AHE, the plaintext of $\mathsf{ct}_{\alpha,\beta}$ is indistinguishable from random. Thus, to the adversary it is independent from the discrete log of R_B.
- If one of the decryptions of ct_γ or $\mathsf{ct}_{\gamma\cdot\mathsf{key}}$ fails then due to the secure random evaluation of AHE the plaintext of $\mathsf{ct}_{\gamma\cdot\mathsf{key}}$ is indistinguishable from random, and thus a_1 can not be extracted.
- If ct_γ is an encryption of zero, then a_1 can not be extracted by the adversary.
- If the plaintexts of ct_γ, $\mathsf{ct}_{\gamma\cdot k_0}$, and $\mathsf{ct}_{\gamma\cdot k_1}$ do not correspond to the discrete logs of $R_{B,0}$ and $R_{B,1}$ then the plaintext of $\mathsf{ct}_{\alpha,\beta}$ is independent of the discrete log of R_B.

In the real execution, the value of a_1 depends on $k_A \cdot x_A$. The only other information of k_A sent to the adversary is $R = k_A^{-1} R_B$. So, without the knowledge of the discrete log of R_B, $k_A \cdot x_A$ is indistinguishable from random. Thus, if a_1 can be extracted by the adversary, it is indistinguishable from random to the adversary. So if the adversary can extract the value of a_1, its values in the two executions are indistinguishable.

In the other case (Step 3), the only differences between the real execution and the simulation are the values of R, ct_A, the signature, and their zk proofs. The value of R

is indistinguishable as it is randomly sampled by the value of k_A in the real execution, and randomly sampled by the oracle in the simulation. The same is true for the zk proofs. Since the plaintexts of $\mathsf{ct}_{\gamma \cdot k_0}$ and $\mathsf{ct}_{\gamma \cdot k_1}$ are the discrete logs of $R_{B,0}$ and $R_{B,1}$ multiplied by the plaintext of ct_γ, the plaintext of $\mathsf{ct}_{\alpha,\beta}$ is the discrete log of R_B. Then, the adversary expects to get a signature as if the public key was $X + (x'_B + \rho_{\mathsf{pid}_A}) \cdot G$. Due to the secure function evaluation, the adversary has no knowledge of how ct_A is computed, and thus, as long as it provides a signature it is indistinguishable from the real execution. Note that in the simulation, the plaintext of ct_A depends on $\mathsf{ct}_{\alpha,\beta}$ only if $\mathsf{ct}_{\alpha,\beta}$ is a valid encryption of a value corresponding to the discrete log of R_B. Therefore, the adversary has no knowledge of this dependency.

Malicious A_{pid_A} Simulation: In this case, the adversary corrupted A_{pid_A} and several parties in B that are not enough to form a quorum.

Key Generation Simulation:

1. Hold sk as it simulates $\mathcal{F}_{\mathsf{TAHE}}$.
2. Receive $(\tilde{X}_0, \tilde{X}_1)$ from $\mathcal{F}_{\mathsf{tsig}}^{\mathcal{G}^*_{\mathsf{SE\text{-}ECDSA}}}$.
3. For every $i \in B \setminus U$:
   (a) Sample $x_0^i, x_1^i \leftarrow \mathbb{Z}_q$ and $\eta_0^i, \eta_1^i \leftarrow \mathcal{R}_{pk}$.
   (b) Set $X_0^i = \tilde{X}_0 + x_0^i \cdot G$, $X_1^i = \tilde{X}_1 + x_1^i \cdot G$, $\mathsf{ct}_0^i = \mathsf{Enc}_{\mathsf{pk}}(x_0^i, \eta_0^i)$, and $\mathsf{ct}_1^i = \mathsf{Enc}_{\mathsf{pk}}(x_1^i, \eta_1^i)$.
   (c) Generate fake proofs using the zk property for the aggregation protocol and use these values as inputs to the aggregation protocol with the adversary.
4. Upon receiving a random oracle call on $(X'_A, X'_{0,B}, X'_{1,B}, \pi'_{\mathsf{DL}})$:
   (a) Check if π'_{DL} is a valid zk proof of X'_A and extract the witness x'_A. If it is not then return a random value.
   (b) If there is a record of $(X'_A, X'_{0,B}, X'_{1,B}, \pi'_{\mathsf{DL}}, \mu'^0_x, \mu'^1_x, \mu'^G_x, \rho)$. Then return $(\overline{\mu^0_x}, \overline{\mu^1_x}, \overline{\mu^G_x}) = (\mu'^0_x, \mu'^1_x, \mu'^G_x - (1 - \rho \cdot \mu'^0_x) \cdot x_A)$.
   (c) Otherwise, sample $\rho \leftarrow \mathbb{Z}_q$.
   (d) Call the random oracle on $(X'_{0,B} + \rho \cdot X'_A, X'_{1,B})$ and obtain μ'^0_x, μ'^1_x, and μ'^G_x.
   (e) Return $(\mu'^0_x, \mu'^1_x, \mu'^G_x - (1-\rho \cdot \mu'^0_x) \cdot x_A)$ and record $(X'_A, X'_{0,B}, X'_{1,B}, \pi'_{\mathsf{DL}}, \mu'^0_x, \mu'^1_x, \mu'^G_x, \rho)$.
5. During the protocol, $\mathcal{S}$ receives $\mathsf{ct}_0^{i'}$ and $\mathsf{ct}_1^{i'}$ for $i' \in U$, along with zk proofs, which it verifies.
   (a) If a proof fails, flag i' as malicious. Otherwise, decrypt $\mathsf{ct}_0^{i'}$ and $\mathsf{ct}_1^{i'}$ to obtain $x_0^{i'}$ and $x_1^{i'}$.
   (b) Emulate $\mathcal{F}_{\mathsf{ACS}}$ and receive an authorized subset S_B from the adversary. This subset must be one where the zk proofs have been validated. Then compute:
      – $X_{0,B} = \sum_{i \in S_B} X_0^i$
      – $X_{1,B} = \sum_{i \in S_B} X_1^i$
      – $\mathsf{ct}_0 = \sum_{i \in S_B} \mathsf{ct}_0^i$
      – $\mathsf{ct}_1 = \sum_{i \in S_B} \mathsf{ct}_1^i$
   (c) Receive from the adversary $(\mathsf{global\text{-}broadcast}, i, X_{0,B}, X_{1,B})$ for every $i \in S_B \setminus U$ and send $(\mathsf{global\text{-}broadcast}, X_{0,B}, X_{1,B})$ to the adversary.
   (d) Upon receiving $(\mathsf{broadcast}, \mathsf{sid}, \mathsf{pid}_A, X_A, \pi_{\mathsf{DL}})$, Verify π_{DL}. If the verification fails, abort; otherwise, extract the witness x_A (this can be done without rewinding since its a UC proof).
   (e) Check if there is a record of $(X_A, X_{0,B}, X_{1,B}, \pi_{\mathsf{DL}}, \mu_x^0, \mu_x^1, \mu_x^G, \rho)$. If there is none, the simulator does as if it received a random oracle call on $(X_A, X_{0,B}, X_{1,B}, \pi_{\mathsf{DL}})$, creates such record and obtains $(\overline{\mu_x^0}, \overline{\mu_x^1}, \overline{\mu_x^G})$.
   (f) Set:

i. $\alpha_x^{(0)} = \alpha_x^{(1)} = |S_B \setminus U|$
ii. $\beta_x^{(0)} = \rho \cdot x_A + \sum_{i \in S_B} x_0^i$
iii. $\beta_x^{(1)} = \sum_{i \in S_B} x_1^i$

(g) Send $(\mathsf{bias}, \mathsf{sid}, \alpha_x^{(0)}, \beta_x^{(0)}, \alpha_x^{(1)}, \beta_x^{(1)})$ to $\mathcal{F}_{\mathsf{tsig}}^{\mathcal{G}^*_{\mathsf{SE\text{-}ECDSA}}}$.
(h) Compute $X_B = \overline{\mu_x^0} \cdot X_{0,B} + \overline{\mu_x^1} \cdot X_{1,B} + \overline{\mu_x^G} \cdot G$ and $\mathsf{ct}_{\mathsf{key}}$ respectively.

Presign and Sign Simulation:

$\mathcal{S}$ sends $(\mathsf{pres}, \mathsf{sid}, \mathsf{ssid}, i)$ to $\mathcal{F}_{\mathsf{tsig}}^{\mathcal{G}^*_{\mathsf{SE\text{-}ECDSA}}}$ for $i \in U$, and receives $(\mathsf{pres}, \mathsf{sid}, \mathsf{ssid}, K_0, K_1)$ in response.

1. Upon receiving $(\mathsf{pres}, \mathsf{sid}, \mathsf{ssid})$ the simulator sends it to the oracle and obtains $(\mathsf{sid}, \mathsf{ssid}, \tilde{R}_0, \tilde{R}_1)$. Then:
   (a) On the first round, the simulator samples γ_i for each honest party and computes ct_γ^i and $\mathsf{ct}_{\gamma \cdot \mathsf{key}}^i$ as encryptions of zeros. It cheats on the aggregated zk proofs.
   (b) Between the rounds, the simulator emulates $\mathcal{F}_{\mathsf{ACS}}$ and receives an authorized subset $S_{0,B}$ from the adversary. This subset must be one where the zk proofs have been validated. The simulator extracts γ_i for each $i \in U \cap S_{0,B}$.
   (c) On the second round, the simulator sets $\mathsf{ct}_{\gamma \cdot k_0}^i$ and $\mathsf{ct}_{\gamma \cdot k_1}^i$ to be encryptions of zeros. The simulator samples k_0^i and k_1^i, and sets $R_{B,0}^i = k_0^i \cdot G + \tilde{R}_0$ and $R_{B,1}^i = k_1^i \cdot G + \tilde{R}_1$.
   (d) After the second round, the simulator emulates $\mathcal{F}_{\mathsf{ACS}}$ again and receives an authorized subset $S_{1,B}$ from the adversary. This subset must be one where the zk proofs have been validated. The simulator extracts k_0^i and k_1^i for each $i \in U \cap S_{1,B}$. The simulator records $(\mathsf{sid}, \mathsf{ssid}, S_{0,B}, S_{1,B}, \{\gamma_i\}_{i \in S_{0,B}}, \{k_0^i\}_{i \in S_{1,B}}, \{k_1^i\}_{i \in S_{1,B}})$. It passes the adversary the correct values $(\mathsf{ct}_\gamma, \mathsf{ct}_{\gamma \cdot \mathsf{key}}, \mathsf{ct}_{\gamma \cdot k_0}, \mathsf{ct}_{\gamma \cdot k_1}, R_{B,0}, R_{B,1})$, outputted by the aggregation protocol.
2. When the simulator receives a random oracle call of the form: $(\mathsf{sid}, \mathsf{msg}, \mathbb{G}, G, q, H, X, (\mathsf{ct}_\gamma, \mathsf{ct}_{\gamma \cdot \mathsf{key}}, \mathsf{ct}_{\gamma \cdot k_0}, \mathsf{ct}_{\gamma \cdot k_1}, R_{B,0}, R$ $\pi_k, \pi_\alpha, \pi_\beta)$, the simulator does as follows:
   (a) The simulator validates the proof. If a proof fails it calls the random oracle on the same input and returns its output.
   (b) The simulator samples ρ.
   (c) The simulator extracts α, β and k_A from the proofs.
   (d) The simulator computes $R'_{B,0} = \rho \cdot R_{B,0}$.
   (e) The simulator calls the random oracle on $(\mathsf{sid}, \mathsf{ssid}, X, 0, R'_{B,0}, R_{B,1}, \mathsf{msg})$ and receives μ_x^0, μ_x^1, and μ_x^G.
   (f) The simulator computes:
      – $\overline{\mu_x^0} = \mu_x^0 \cdot \rho \cdot k_A \cdot \alpha^{-1}$.
      – $\overline{\mu_x^1} = \mu_x^1 \cdot k_A \cdot \alpha^{-1}$.
      – $\overline{\mu_x^G} = \alpha^{-1}(k_A \cdot \mu_x^G - \beta)$
   (g) The simulator sends $(\overline{\mu_x^0}, \overline{\mu_x^1}, \overline{\mu_x^G})$ to the adversary.
   (h) The simulator records the random oracle call with the sampled ρ and $(\overline{\mu_x^0}, \overline{\mu_x^1}, \overline{\mu_x^G})$.
3. When the simulator receives $(\mathsf{broadcast}, \mathsf{sid}, (R, R_B, C_k, C_\alpha, C_\beta, C_{kx}, \mathsf{ct}_A, \mathsf{ct}_{\alpha,\beta}, \pi_k, \pi_\alpha, \pi_\beta, \pi_{kx}, \pi_R, \pi_{R_B}, \pi_{\mathsf{ct}_{\alpha,\beta}}, \pi_{\mathsf{ct}_A}))$, it loads the corresponding records and does as follows:
   (a) If there is no random call record of $(\mathsf{sid}, \mathsf{msg}, \mathbb{G}, G, q, H, X, \mathsf{pres}_{X,\mathsf{sid}}, C_k, C_{kx}, X_A, C_\alpha, C_\beta, \pi_k, \pi_\alpha, \pi_\beta)$ then the simulator sends a random oracle request to itself and create the corresponding record. The simulator obtains $(\overline{\mu_x^0}, \overline{\mu_x^1}, \overline{\mu_x^G})$ and ρ.
   (b) If a zk proof fails, the simulator aborts.

(c) The simulator extracts and computes:
   i. $k_{B,0} = \sum_{i \in S_{1,B}} k^i_{B,0}$
   ii. $k_{B,1} = \sum_{i \in S_{1,B}} k^i_{B,1}$
(d) The simulator sends to the oracle $(\mathsf{sign}, \mathsf{msg}, \mathsf{sid}, \mathsf{ssid}, 0, |S_{1,B} \backslash U| \cdot \rho, \rho \cdot k_{B,0}, |S_{1,B} \backslash U|, k_{B,1})$ and receives $\sigma = (r, s)$.
(e) The simulator samples ρ'. On $(\mathsf{decrypt}, \mathsf{pk}, \mathsf{ct}_A)$ the simulator returns $s \cdot \rho'$, and on $(\mathsf{decrypt}, \mathsf{pk}, \mathsf{ct}_{\alpha,\beta})$ it returns ρ'.
(f) The simulator verifies that it receives σ from the adversary.

Malicious A_{pid_A} Indistinguishability:

Key Generation: The view of the adversary in the real and ideal worlds is $\{X_{0,i}, X_{1,i}\}_{i \notin U}, \{\mathsf{ct}_{0,i}, \mathsf{ct}_{1,i}\}_{i \notin U}$ along with the zk proofs and random oracle calls. The $X_{0,i}$ and $X_{1,i}$ are uniformly distributed in both the real and simulated views. The encryptions are indistinguishable from the semantic security of the TAHE scheme and the zk proofs are indistinguishable from the zk property. The value of the modified random oracle are indistinguishable from a true random oracle since the values $(\mu'^0_x, \mu'^1_x, \mu'^G_x - (1 - \rho \cdot \mu'^0_x) \cdot x_A)$ are uniformly distributed when ρ is uniformly distributed.

From the soundness property of the zk proofs for malicious parties $\mathsf{ct}_{0,i} = \mathsf{Enc}_{\mathsf{pk}}(log_G(X_{0,i}))$ and $\mathsf{ct}_{1,i} = \mathsf{Enc}_{\mathsf{pk}}(log_G(X_{1,i}))$ with overwhelming probability, thus the simulation extracts the correct values via decrypting ct^i_0 and ct^i_1. From the knowledge soundness of π_{DL} the simulator extracts the correct witnesses with overwhelming probability. Finally, notice that the final X is indeed the final output of $\mathcal{F}^{\mathcal{G}^*_{\mathsf{SE\text{-}ECDSA}}}_{\mathsf{tsig}}$ as:

$$
\begin{aligned}
X &= \mu^0_x \cdot X_0 + \mu^1_x \cdot X_1 + \mu^G_x \cdot G \\
&= \mu^0_x \cdot (\alpha^0_x \cdot \tilde{X}_0 + \beta^0_x \cdot G) + \mu^1_x \cdot (\alpha^1_x \cdot \tilde{X}_1 + \beta^1_x \cdot G) + \mu^G_x \cdot G \\
&= \mu^0_x \cdot (|S_B \setminus U| \cdot \tilde{X}_0 + \rho \cdot x_A \cdot G + \sum_{i \in S_B} x^i_0 \cdot G) + \\
&\qquad + \mu^1_x \cdot (|S_B \setminus U| \cdot \tilde{X}_1 + \sum_{i \in S_B} x^i_1 \cdot G) + \mu^G_x \cdot G \\
&= \mu^0_x \cdot (\rho \cdot x_A \cdot G + \sum_{i \in U} (x^i_0 \cdot G) + \sum_{i \in S_B \setminus U} (\tilde{X}_0 + x^i_0 \cdot G)) + \\
&\qquad + \mu^1_x \cdot (\sum_{i \in U} (x^i_1 \cdot G) + \sum_{i \in S_B \setminus U} (\tilde{X}_1 + x^i_1 \cdot G)) + \mu^G_x \cdot G \\
&= \mu^0_x \cdot (\rho \cdot X_A + \sum_{i \in U} X^i_0 + \sum_{i \in S_B \setminus U} X^i_0) + \\
&\qquad + \mu^1_x \cdot (\sum_{i \in U} X^i_1 + \sum_{i \in S_B \setminus U} X^i_1) + \mu^G_x \cdot G \\
&= \mu^0_x \cdot (\rho \cdot X_A + \sum_{i \in S_B} X^i_0) + \mu^1_x \cdot \sum_{i \in S_B} X^i_1 + \mu^G_x \cdot G \\
&= \mu^0_x \cdot (\rho \cdot X_A + X_{0,B}) + \mu^1_x \cdot X_{1,B} + \mu^G_x \cdot G \\
&= \overline{\mu^0_x} \cdot (\rho \cdot X_A + X_{0,B}) + \overline{\mu^1_x} \cdot X_{1,B} + (\overline{\mu^G_x} + (1 - \rho \cdot \overline{\mu^0_x}) x_A) \cdot G \\
&= X_A + \overline{\mu^0_x} \cdot X_{0,B} + \overline{\mu^1_x} \cdot X_{1,B} + \overline{\mu^G_x} \cdot G
\end{aligned}
$$

Presign and sign: In the presign protocol, the differences between the real execution and the simulation are the encryptions. The adversary does not hold the private key

and thus the encryptions are indistinguishable for the adversary. The random oracle is indistinguishable between the two executions due to the sampling of the value ρ that causes the output to be uniformly sampled. In the sign protocol, the differences between the real execution and the simulation are the values of pt_A and the plaintext of $\mathsf{ct}_{\alpha,\beta}$. From the hiding property of encryption scheme and since the adversary does not hold sk, the two values are indistinguishable.

Denote by R' the value of R that the oracle send to the simulator. Note that R' equals the value of R that the adversary computes:

$$
\begin{aligned}
R &= k_A^{-1} \cdot R_B \\
&= k_A^{-1} \cdot (\alpha \cdot R'_B + \beta \cdot G) \\
&= k_A^{-1} \cdot \left(\alpha \cdot \left(\overline{\mu_x^0} \cdot R_{B,0} + \overline{\mu_x^1} \cdot R_{B,1} + \overline{\mu_x^G} \cdot G\right) + \beta \cdot G\right) \\
&= k_A^{-1} \cdot \left(\alpha \cdot \left(\frac{\mu_x^0 \cdot \rho \cdot k_A}{\alpha} \cdot R_{B,0} + \frac{\mu_x^1 \cdot k_A}{\alpha} \cdot R_{B,1} + \frac{k_A \cdot \mu_x^G - \beta}{\alpha} \cdot G\right) + \beta \cdot G\right) \\
&= k_A^{-1} \cdot \left(\mu_x^0 \cdot \rho \cdot k_A \cdot R_{B,0} + \mu_x^1 \cdot k_A \cdot R_{B,1} + k_A \cdot \mu_x^G \cdot G - \beta \cdot G + \beta \cdot G\right) \\
&= \mu_x^0 \cdot \rho \cdot R_{B,0} + \mu_x^1 \cdot R_{B,1} + \mu_x^G \cdot G \\
&= \mu_x^0 \cdot \rho \cdot \left(\sum_{i \in S_{1,B}} R_{B,0}^i\right) + \mu_x^1 \cdot \left(\sum_{i \in S_{1,B}} R_{B,1}^i\right) + \mu_x^G \cdot G \\
&= \mu_x^0 \cdot \rho \cdot \left(\sum_{i \in S_{1,B} \cap U} R_{B,0}^i + \sum_{i \in S_{1,B} \setminus U} R_{B,0}^i\right) + \\
&\qquad + \mu_x^1 \cdot \left(\sum_{i \in S_{1,B} \cap U} R_{B,1}^i + \sum_{i \in S_{1,B} \setminus U} R_{B,1}^i\right) + \mu_x^G \cdot G \\
&= \mu_x^0 \cdot \rho \cdot \left(\sum_{i \in S_{1,B} \cap U} (k_{B,0}^i \cdot G) + \sum_{i \in S_{1,B} \setminus U} (k_{B,0}^i \cdot G + \tilde{R}_0)\right) + \\
&\qquad + \mu_x^1 \cdot \left(\sum_{i \in S_{1,B} \cap U} (k_{B,1}^i \cdot G) + \sum_{i \in S_{1,B} \setminus U} (k_{B,1}^i \cdot G + \tilde{R}_1)\right) + \mu_x^G \cdot G \\
&= \mu_x^0 \cdot \rho \cdot \left(|S_{1,B} \setminus U| \cdot \tilde{R}_0 + k_{B,0} \cdot G\right) + \mu_x^1 \cdot \left(|S_{1,B} \setminus U| \cdot \tilde{R}_1 + k_{B,1} \cdot G\right) + \mu_x^G \cdot G \\
&= \mu_x^0 \cdot R_0 + \mu_x^1 \cdot R_1 + \mu_x^G \cdot G = R'
\end{aligned}
$$

D.2 Proof of the Security of Slightly Enhanced ECDSA

Our security analysis for Slightly Enhanced ECDSA is based on the approach introduced in [53]. For convenience of readers familiar with their work, in this section we follow their notation. In Table 1 we compare the notation used in this section (and in [53]) with the notation in the rest of this paper.

A simulation of Functionality 2 in the EC-GGM is presented in Figure 6. Utilizing the fact that the group is modeled as an EC-GGM, we design it as a lazy simulation, implying that elements of the group are sampled only when they are needed.

Object	Notation: [53]	Notation: this work
EC group	E	$\mathbb{G}$
Generator	G	G
Secret key	d	x
Public key	D	X
Nonce	r	k
Ephemeral public key (EPK)	R	K
X-coordinate of EPK	t	r
Additive key derivation	e_k	β'_{key}

Table 1: Notation summary, [53] vs this work.

The design of the lazy simulation 6 and the reduction to the symbolic simulation 7 are essentially the same as the security proof of ECDSA with presigns in [53, Theorem 3]. Therefore, our main innovation lies in the security proof of the symbolic simulation. This proof consists of two steps:

1. Reduction to the *modified* symbolic simulation 8: The key distinction is that, unlike the original simulation, the modified version does not let the adversary select affine transformations in key-bias and sign requests; instead, these transformations are sampled pseudo-randomly. However, the adversary's influence over these values through transformations is ultimately negligible. We demonstrate this by programming the random oracles $\mathcal{H}_{\mathsf{key}}$ and $\mathcal{H}_k$ to cancel the adversary's transformations.
2. Security proof of the modified symbolic simulation: In a nutshell, sampling two points per presign allows us to introduce *two symbolic variables* per presign in the symbolic simulation, and fix only one of them in the corresponding sign request. This in turn results in many independent symbolic variables. We show that this essentially forces the adversary to fix the messages to be signed *before* choosing the point R^* that will be used in the forgery, which corresponds to the nonce part of the signature. Then, by modeling the hash functions as random oracles, we can extract its future sign queries and minimize the effect of presign.

As a consequence of our approach, and in contrast to [53], we do not reduce the security of the scheme to concrete problems on the underlying hash functions, but prove the security when the hash functions are modeled as random oracles. On the positive side, our scheme and our proof approach enables better bounds on the success probability of an efficient adversary, and hold even for an adversary which is allowed to apply affine transformations on the key and the presign values.

Additionally, our security proof allows the adversary to ask for signatures with respect to multiple keys derived linearly from the same global key, without any effect on the security of the system.

Similarly to [53], we begin with presenting a corresponding symbolic simulation:

Lemma 8 *A PPT adversary forging signatures for the lazy simulation (Simulation 6) with success rate ϵ, can forge signatures for the symbolic simulation (Simulation 7) with success rate at least $\epsilon - \mathcal{O}(\frac{N^2}{q})$.*

Proof. We describe below a sequence of hybrids where Hybrid_0 is the lazy simulation and Hybrid_2 is the symbolic simulation. We show that in each step the success rate gain is upper bounded by $\mathcal{O}(\frac{N^2}{q})$.

Hybrid$_1$. We replace $\tilde{d}$ and $\tilde{r}_k^0, \tilde{r}_k^1$ for every k with symbolic variables $\tilde{\mathbf{d}}$ and $\tilde{\mathbf{r}}_k^0$, $\tilde{\mathbf{r}}_k^1$. At the end of the simulation, sample $\tilde{d}, \tilde{r}_k^1 \leftarrow \mathbb{Z}_q^*$. In addition, we sample s_k as in the symbolic simulation. When run with the same random tape, $\mathsf{Hybrid}_0, \mathsf{Hybrid}_1$ may return a different output only if a collision has occurred. Namely, if there are two elements $i \neq j \in \mathsf{Domain}(\pi)$ with $i(d, \tilde{r}_1^1, \tilde{r}_2^1, \dots) = j(d, \tilde{r}_1^1, \tilde{r}_2^1, \dots)$. Since there are $\mathcal{O}(N)$ such polynomials, and since the coefficient of each monomial is independent of the evaluation point, the probability of such a collision is bounded by $O(\frac{N^2}{q})$, by the Schwartz-Zippel Lemma. See [53, Appendix B] for the full details.

Hybrid$_2$. This final hybrid, which matches the symbolic simulation, differ from Hybrid_1 in only one aspect: If a newly sampled element for $\mathsf{Domain}(\pi)$ or $\mathsf{Range}(\pi)$ collides with an existing one, the simulation aborts instead of resampling. Since the number of samples is bounded by $\mathcal{O}(N)$, the total number of elements in $\mathsf{Range}(\pi)$ and $\mathsf{Domain}(\pi)$ is also at most $\mathcal{O}(N)$. Consequently, the probability of a collision is at most $\mathcal{O}(\frac{N^2}{q})$.

Next, we prove the equivalence between the symbolic simulation and its modified version:

Lemma 9 *A PPT adversary forging signatures for the symbolic simulation (Simulation 7) with success rate ϵ, can forge signatures for the modified symbolic simulation (Simulation 8) with success rate at least $\epsilon - \mathcal{O}(\frac{N^2}{q})$.*

Proof. There are two differences between the simulations, in the bias key and the sign queries. We begin with handling the bias key query and bounding the effect of its modification, the modification of the sign queries is handled similarly. In the symbolic simulation, the adversary sends coefficients $\alpha_x^0, \beta_x^0, \alpha_x^1, \beta_x^1$, and the simulation sets

$$(\mu_x^0, \mu_x^1, \mu_x^G) \leftarrow \mathcal{H}_{\mathsf{key}}\left(\pi(\alpha_x^0 \cdot \tilde{\mathbf{d}}_0 + \beta_x^0),\ \pi(\alpha_x^1 \cdot \tilde{\mathbf{d}}_1 + \beta_x^1)\right),$$

and sets

$$d(\tilde{\mathbf{d}}_0, \tilde{\mathbf{d}}_1) \leftarrow \mu_x^0(\alpha_x^0 \cdot \tilde{\mathbf{d}}_0 + \beta_x^0) + \mu_x^1(\alpha_x^1 \cdot \tilde{\mathbf{d}}_1 + \beta_x^1) + \mu_x^G. \tag{14}$$

In the modified simulation, on the other hand, the adversary only gets to choose the seed id_x from which the tuple $(\mu_x^0, \mu_x^1, \mu_x^G)$ is derived, and the symbolic secret key is defined as $\mu_x^0\tilde{\mathbf{d}}_0 + \mu_x^1\tilde{\mathbf{d}}_1 + \mu_x^G$.

First, we note that in Simulation 8 the identifier id_x is assumed to encapsulate all context and inputs controlled by the adversary. In this case, id_x represents the tuple $(\alpha_x^0, \beta_x^0, \alpha_x^1, \beta_x^1)$, capturing the fact that the adversary's only influence on the randomized state is through this input to the random oracle. More generally, id_x may include other session-specific identifiers or auxiliary context as needed.

By programming $\mathcal{H}_{\mathsf{key}}$ and utilizing the properties of a symbolic representation, we can show these processes are equivalent: We replace the random oracle of the symbolic simulation with a programmed random oracle $\mathcal{H}'_{\mathsf{key}}$ defined as follows:

1. Receive a tuple (D_0, D_1) of group elements as inputs. If these values were received earlier, return the corresponding output; otherwise, continue.
2. For $j = 0, 1$: if $D_j \notin \mathsf{Range}(\pi)$:
   (a) Sample $i \in \mathbb{Z}_q$. If $i \in \mathsf{Domain}(\pi)$, abort.
   (b) Set $D_j = \pi(i)$ and $-D_j = \pi(-i)$.
3. Denote $d_0 = \pi^{-1}(D_0)$ and $d_1 = \pi^{-1}(D_1)$.

4. If d_0 is of the form $\alpha_x^0 \cdot \tilde{\mathbf{d}}_0 + \beta_x^0$ then extract α_x^0 and β_x^0; otherwise, stop and return an arbitrary output. Similarly, extract α_x^1 and β_x^1 from d_1 if possible and stop otherwise.
5. Set $(\mu_x^0, \mu_x^1, \mu_x^G) = \mathcal{H}_{\mathsf{key}}(\pi(\tilde{\mathbf{d}}_0), \pi(\tilde{\mathbf{d}}_1), (D_0, D_1))$ to be the output of $\mathcal{H}_{\mathsf{key}}$ in the modified simulation (with the tuple (D_0, D_1) modeled as id_x).
6. Return
$$(\bar{\mu}_x^0, \bar{\mu}_x^1, \bar{\mu}_x^G) = \left(\frac{\mu_x^0}{\alpha_x^0}, \frac{\mu_x^1}{\alpha_x^1}, \mu_x^G - \frac{\mu_x^0 \beta_x^0}{\alpha_x^0} - \frac{\mu_x^1 \beta_x^1}{\alpha_x^1} \right).$$

One can see that substituting $\mathcal{H}'_{\mathsf{key}}\left(\pi(\alpha_x^0 \cdot \tilde{\mathbf{d}}_0 + \beta_x^0),\ \pi(\alpha_x^1 \cdot \tilde{\mathbf{d}}_1 + \beta_x^1)\right)$ in (14) yields $\mu_x^0 \tilde{\mathbf{d}}_0 + \mu_x^1 \tilde{\mathbf{d}}_1 + \mu_x^G$, which is equivalent to the secret key $d(\tilde{\mathbf{d}}_0, \tilde{\mathbf{d}}_1)$ of the modified simulation. Therefore, Simulation 7 and Simulation 8 are equivalent as long as $\mathcal{H}'_{\mathsf{key}}$ never aborts. It can be easily verified that the probability that $\mathcal{H}'_{\mathsf{key}}$ ever aborts is bounded by $\mathcal{O}(\frac{N^2}{q})$.

A very similar programming of $\mathcal{H}_k$ allows canceling the adversary's bias in the presign values as well. We omit this analysis as it follows analogously.

Next, we may proceed to proving security of slightly enhanced ECDSA:

Theorem 10 *Let $\mathcal{A}$ be an adversary to the existential unforgeability game 1 with respect to the ECDSA oracle $\mathcal{G}^*_{\mathsf{SE\text{-}ECDSA}}$ (Functionality 2) in the EC-GGM model (LazySimulation 6), that makes at most N presignature, signing, hash, or group queries, when $\mathcal{H}_{key}, \mathcal{H}_{\mathcal{M}}$ and all $\mathcal{H}_k$ are modeled as independent Random Oracles. Then*

$$\mathit{Adv}(\mathcal{A}, \mathsf{Exp}^{\mathcal{G}^*_{\mathsf{SE\text{-}ECDSA}}}_{\mathsf{EU}}, \textit{lazy-sim}) \leq \mathcal{O}(\frac{N^2}{q}).$$

Proof. By Lemma 8 and Lemma 9, it is sufficient to bound the advantage in the modified symbolic simulation (Simulation 8). Following the proof of [53, Theorem 1], we denote the forgery by $(D^*, h^*, R^*, s^*, t^*)$, where $h^* = \mathcal{H}_{\mathcal{M}}(m^*)$. We focus on the initial map query that produced R^*, denoted as $R^* \leftarrow (\mathsf{map}, r^*_{\mathsf{init}})$. Notably, r^*_{init} may be either a scalar or a function of symbolic variables. Since the expression for $\pi^{-1}(R^*)$ may change over time due to variable substitutions, we let r^*_{final} denote its final form. The forgery is successful if the equation

$$r^*_{\mathsf{final}} = s^{*-1}(h^* + t^*(\mu_x^0 \tilde{\mathbf{d}}_0 + \mu_x^1 \tilde{\mathbf{d}}_1 + \mu_x^G)) \tag{15}$$

holds for the triplet $(\mu_x^0, \mu_x^1, \mu_x^G)$ corresponding to D^*. Without loss of generality, we assume that the adversary evaluates each public key D before making the corresponding key bias request, evaluates each R_k before requesting the corresponding signature, and verifies that the forgery equation holds before outputting the forgery. Consequently, the query $R^* \leftarrow (\mathsf{map}, r^*_{\mathsf{init}})$ must originate from either $\mathcal{A}$'s group query, the key generation process, or a presign request.

Notably, the probability that some call to $\mathcal{H}_{\mathsf{key}}$ or $\mathcal{H}_k$ outputs a triplet containing a zero coordinate is at most $\mathcal{O}\left(\frac{N}{q}\right)$. Therefore, we may henceforth assume that all such outputs consist of three non-zero elements.

First, observe that a successful forgery cannot involve an R^* produced during the key generation. Indeed, if $r^*_{\mathsf{init}} = \pm\tilde{\mathbf{d}}_i$ for $i \in 0, 1$, then $r^*_{\mathsf{final}} = \pm\tilde{\mathbf{d}}_i$, since neither $\tilde{\mathbf{d}}_0$ nor $\tilde{\mathbf{d}}_1$ is affected by the substitutions. Both μ_x^0 and μ_x^1 are assumed to be non-zero, which implies that the coefficients of $\tilde{\mathbf{d}}_0$ and $\tilde{\mathbf{d}}_1$ in (15) are either both zero or both non-zero — contradicting the structure of a valid forgery.

Similarly, a successful forgery cannot have $r^*_{\mathsf{init}} = \pm\tilde{\mathbf{r}}^1_k$, since (15) does not involve the variable $\tilde{\mathbf{r}}^1_k$, and it is also unaffected by substitutions.

Moreover, a forgery cannot involve $r^*_{\mathsf{init}} = \pm\tilde{\mathbf{r}}^0_k$, even though this variable is affected by substitutions. Indeed, substituting for $\tilde{\mathbf{r}}^0_k$ transforms (15) into:

$$s^{*-1}(h^* + t^* d) = \pm\left(\mu^0_k\right)^{-1}\left(-\mu^1_k\tilde{\mathbf{r}}^1_k + s_k^{-1}t_k d_k + s_k^{-1}h_k - \mu^G_k\right),$$

where d and d_k are functions of $\tilde{\mathbf{d}}_0$ and $\tilde{\mathbf{d}}_1$. This symbolic equation is impossible, since $\tilde{\mathbf{r}}^1_k$ appears only once and is not canceled (as both μ^0_k and μ^1_k are non-zero).

We therefore conclude that the query $R^* \leftarrow (\mathsf{map}, r^*_{\mathsf{init}})$ must have been issued directly by the adversary, and not as part of the key generation or presign processes. Next, we split forgeries into two types and analyze the probability of each separately.

Type I forger. Suppose $R^* = \eta R_k$ for some index k and $\eta \in \{\pm 1\}$, where R_k is the first term of the k-th queried signature. Following the description of the signing oracle, we obtain that

$$r^*_{\mathsf{final}} = \eta\pi^{-1}(R_k) = \eta s_k^{-1}(h_k + t_k((\mu^0_x)_k\tilde{\mathbf{d}}_0 + (\mu^1_x)_k\tilde{\mathbf{d}}_1 + (\mu^G_x)_k + e_k)),$$

for the triplet $((\mu^0_x)_k, (\mu^1_x)_k, (\mu^G_x)_k)$ corresponding to the public key D used in the k-th signature query. From the identity $R^* = \eta R_k$, we obtain $t^* = \bar{C}(\eta R_k) = t_k$, since the X-coordinate of an EC element is invariant under negation.

Combining this with (15), we find that

$$\eta s_k^{-1}(h_k + t_k((\mu^0_x)_k\tilde{\mathbf{d}}_0 + (\mu^1_x)_k\tilde{\mathbf{d}}_1 + (\mu^G_x)_k + e_k)) = s^{*-1}(h^* + t_k(\mu^0_x\tilde{\mathbf{d}}_0 + \mu^1_x\tilde{\mathbf{d}}_1 + \mu^G_x))$$

holds as a symbolic equation, implying:

$$\begin{cases} \eta s_k^{-1} t_k (\mu^0_x)_k = s^{*-1} t_k \mu^0_x, \\ \eta s_k^{-1} t_k (\mu^1_x)_k = s^{*-1} t_k \mu^1_x, \\ \eta s_k^{-1}\left(h_k + t_k(\mu^G_x)_k + t_k e_k\right) = s^{*-1}\left(h^* + t_k\mu^G_x\right). \end{cases} \tag{16}$$

Recalling that both t_k and μ^0_x are non-zero, it follows from (16) that:

$$\begin{cases} \dfrac{\mu^0_x}{\mu^1_x} = \dfrac{(\mu^0_x)_k}{(\mu^1_x)_k}, \\ \mu^0_x\left(h_k + t_k(\mu^G_x)_k + t_k e_k\right) = (\mu^0_x)_k\left(h^* + t_k\mu^G_x\right). \end{cases} \tag{17}$$

Bounding the probability of a Type I forgery. Let Z_{I} be the event that, at any point during $\mathcal{A}$'s execution, one of the following two events occurs:

1. For some index k, $\mathcal{H}_k$ returns the same coefficient triplet $(\mu^0_k, \mu^1_k, \mu^G_k)$ up to negation for two distinct inputs.
2. $\mathcal{H}_{\mathsf{key}}$ produces a ratio collision, i.e., it returns two different coefficient triplets with the same μ^0_x/μ^1_x.
3. $\mathcal{H}_{\mathcal{M}}$ produces the same output on two different messages.

Since both hash functions are assumed to behave as random oracles, we can bound $\Pr[Z_{\mathrm{I}}] = \mathcal{O}\left(\frac{N^2}{q}\right)$.

Next, denote the i-th map query by $\mathcal{P}_i \leftarrow (\mathsf{map}, \alpha_i)$. For each $i \in [N]$ and each $h \in \mathsf{Range}(\mathcal{H}_{\mathcal{M}})$, define $B^{\mathrm{I}}_{i,h}$ to be the event that Z_{I} does not occur and $\mathcal{A}$ outputs a

valid Type I forgery with $h^* = h$, and $R^* = \mathcal{P}_i$. Since the adversary can make at most N queries to the map oracle and to $\mathcal{H}_{\mathcal{M}}$, there are at most N^2 possible combinations of (i, h). Hence, if we can show that $\Pr[B^{\mathrm{I}}_{i,h}] = \mathcal{O}\left(\frac{1}{q}\right)$ for all i and $h \in \mathsf{Range}(\mathcal{H}_{\mathcal{M}})$, then we obtain

$$\Pr[\text{Type I forgery}] \leq \Pr[Z_{\mathrm{I}}] + \sum_{i,h} \Pr\left[B^{\mathrm{I}}_{i,h}\right] \leq \mathcal{O}\left(\frac{N^2}{q}\right),$$

which completes the argument for Type I forgeries.

we now proceed to prove that $\Pr[B^{\mathrm{I}}_{i,h}] = \mathcal{O}\left(\frac{1}{q}\right)$ for all i, h.

From (17) we have the relation

$$\frac{\mu^0_x}{\mu^1_x} = \frac{(\mu^0_x)_k}{(\mu^1_x)_k}.$$

Since event (2) of Z_{I} does not occur, the two triplets must match: $(\mu^0_x, \mu^1_x, \mu^G_x) = ((\mu^0_x)_k, (\mu^1_x)_k, (\mu^G_x)_k)$. Substituting this equation into (17) yields $h_k + t_k e_k = h$. Notably, the offset e_k must be non-zero. Indeed, $e_k = 0$ implies $h_k = h$. Since event (3) does not occur, this further implies $m_k = m^*$. Hence, the forgery corresponds exactly to a previously queried signature, contradicting the definition of a successful forgery.

Next, we assume $e_k \neq 0$. Since $h_k + t_k e_k = h$, we obtain $t_k = \frac{h - h_k}{e_k}$. Notably, t_k is derived from R_k, which is sampled uniformly from the curve after fixing $r_k = \eta\alpha_i$. From the description of the signing oracle one can see that $r_k = \eta\alpha_i$ identifies the triplet $(\mu^0_k, \mu^1_k, \mu^G_k)$. Since event (1) of Z_{I} does not occur, this triplet identifies the input to $\mathcal{H}_k$, including h_k and e_k. Therefore, t_k is sampled independently of h_k and e_k, implying that the probability of $t_k = \frac{h - h_k}{e_k}$ is at most $\mathcal{O}\left(\frac{1}{q}\right)$.

We conclude that $\Pr\left[B^{\mathrm{I}}_{i,h}\right] = \mathcal{O}\left(\frac{1}{q}\right)$, completing the proof that

$$\Pr[\text{Type I forgery}] = \mathcal{O}\left(\frac{N^2}{q}\right).$$

Type II forger. Suppose $R^* \neq \pm R_k$ for all k. Denote by K_{II} the set of indices of presign variables appearing in r^*_{init}. Note that K_{II} may be empty. Then the initial expression for $\pi^{-1}(R^*)$ can be written as:

$$r^*_{\mathsf{init}} = a + b_0\tilde{\mathbf{d}}_0 + b_1\tilde{\mathbf{d}}_1 + \sum_{k \in K_{\mathrm{II}}} \left(c_k\tilde{\mathbf{r}}^0_k + \bar{c}_k\tilde{\mathbf{r}}^1_k\right), \tag{18}$$

where for each $k \in K_{\mathrm{II}}$, at least one of c_k or $\bar{c}_k$ is non-zero.

In order for (15) to hold, all variables in (18) other than $\tilde{\mathbf{d}}_0$ and $\tilde{\mathbf{d}}_1$ must be eliminated by substitutions. We first observe that all presigns $k \in K_{\mathrm{II}}$ must have been used prior to the forgery. Otherwise, the variables $\tilde{\mathbf{r}}^0_k$ or $\tilde{\mathbf{r}}^1_k$ would remain in r^*_{final}, contradicting (15). Moreover, since the k-th presign was used, it induces a substitution for $\tilde{\mathbf{r}}^0_k$ of the form

$$\tilde{\mathbf{r}}^0_k \mapsto -\frac{\mu^1_k}{\mu^0_k}\tilde{\mathbf{r}}^1_k + \cdots,$$

so that the coefficient of $\tilde{\mathbf{r}}^1_k$ in r^*_{final} becomes $\bar{c}_k - \frac{c_k\mu^1_k}{\mu^0_k}$. Since this variable must not appear in r^*_{final}, it must vanish, i.e., $\bar{c}_k = \frac{c_k\mu^1_k}{\mu^0_k}$. Hence, if $\bar{c}_k = 0$, it follows that $c_k = 0$,

contradicting the assumption that at least one of $c_k, \bar{c}_k$ is non-zero. Therefore, $\bar{c}_k \neq 0$, implying:

$$\forall k \in K_{\text{II}} : \frac{c_k}{\bar{c}_k} = \frac{\mu_k^0}{\mu_k^1}. \tag{19}$$

Bounding the probability of a Type II forgery. Let Z_{II} be the event that, at any point during $\mathcal{A}$'s execution, one of the following events occurs:

1. For some index k, $\mathcal{H}_k$ produces a ratio collision; that is, it returns two coefficient triplets with the same ratio μ_k^0/μ_k^1 for distinct inputs.
2. $\mathcal{H}_{\mathsf{key}}$ produces a ratio collision; that is, it returns two coefficient triplets with the same ratio μ_x^0/μ_x^1 for distinct inputs.
3. $\mathcal{H}_{\mathcal{M}}$ outputs 0.

Since both hash functions are assumed to behave as random oracles, we can bound $\Pr[Z_{\text{II}}] = \mathcal{O}\left(\frac{N^2}{q}\right)$.

Next, denote the i-th map query by $\mathcal{P}_i \leftarrow (\mathsf{map}, \alpha_i)$, where

$$\alpha_i = a^i + b_0^i \tilde{\mathbf{d}}_0 + b_1^i \tilde{\mathbf{d}}_1 + \sum_{k \in K_{\text{II}}} \left(c_k^i \tilde{\mathbf{r}}_k^0 + \bar{c}_k^i \tilde{\mathbf{r}}_k^1 \right).$$

For each $i \in [N]$ and each $h \in \mathsf{Range}(\mathcal{H}_{\mathcal{M}})$, define $B_{i,h}^{\text{II}}$ to be the event that Z_{II} does not occur and $\mathcal{A}$ outputs a valid forgery with $h^* = h$, and $R^* = \mathcal{P}_i$. Since the adversary can make at most N queries to the map oracle and to $\mathcal{H}_{\mathcal{M}}$, there are at most N^2 possible combinations of (i, h). Hence, if we can show that $\Pr[B_{i,h}^{\text{II}}] = \mathcal{O}\left(\frac{1}{q}\right)$ for all $i \in [N]$ and $h \in \mathsf{Range}(\mathcal{H}_{\mathcal{M}})$, then we obtain

$$\Pr[\text{Type II forgery}] \leq \Pr\left[Z_{\text{II}}\right] + \sum_{i,h} \Pr\left[B_{i,h}^{\text{II}}\right] \leq \mathcal{O}\left(\frac{N^2}{q}\right),$$

which completes the argument.

we now proceed to prove that $\Pr[B_{i,h}^{\text{II}}] = \mathcal{O}\left(\frac{1}{q}\right)$ for all i, h.

Suppose the forgery uses $R^* = \mathcal{P}_i$, and let us fix the randomness of all signing queries indexed by K_{II}. From (19), we obtain that for every $k \in K_{\text{II}}$, the coefficients in the symbolic expression must satisfy

$$\frac{c_k^i}{\bar{c}_k^i} = \frac{\mu_k^0}{\mu_k^1},$$

where $(\mu_k^0, \mu_k^1, \mu_k^G) \leftarrow \mathcal{H}_k(D, e_k, \tilde{R}_k^0, \tilde{R}_k^1, m_k, \mathsf{id}_k)$ are the coefficients which will be used in the k-th signing process.

Since the values μ_k^0/μ_k^1 are assumed to be distinct across inputs (i.e., Z_{II} does not occur), the ratio $c_k^i/\bar{c}_k^i$ uniquely determines the input to $\mathcal{H}_k$. Therefore, for each $k \in K_{\text{II}}$, the coefficients c_k^i and $\bar{c}_k^i$ (fixed when the i-th map query is made) determine a unique choice of:

- the message m_k,
- the offset e_k,
- the resulting triplet $(\mu_k^0, \mu_k^1, \mu_k^G)$,
- the public key D_k, and thus the corresponding secret key d_k used in the substitution.

Consequently, the values substituted for the symbolic variables $\tilde{\mathbf{r}}_k^0$ are fully determined by the choice of α_i and the fixed randomness in the relevant signing queries (those indexed by K_{II}). Therefore, there is at most one possible instantiation of the symbolic expression that leads to a forgery, and it must take the form:

$$r^*_{\mathsf{final}} = A^i + B_0^i \tilde{\mathbf{d}}_0 + B_1^i \tilde{\mathbf{d}}_1, \tag{20}$$

for some constants $A^i, B_0^i, B_1^i \in \mathbb{F}_q$ determined by the above substitutions.

This outcome is not surprising, given that the symbolic variables $\tilde{\mathbf{r}}_k^0$ and $\tilde{\mathbf{r}}_k^1$ must be eliminated for the forgery equation to match the required form. This can only happen if the coefficients satisfy the constraints in (19). These constraints, in turn, rigidly fix the inputs to all relevant signing queries, thereby determining the final symbolic form r^*_{final}.

Let us now complete the argument. Recall that we are analyzing a forgery with message m^* such that $\mathcal{H}_{\mathcal{M}}(m^*) = h$, and we assume $h \neq 0$. Combining (20) with the forgery condition (15), we obtain:

$$A^i + B_0^i \tilde{\mathbf{d}}_0 + B_1^i \tilde{\mathbf{d}}_1 = \eta s^{*-1} \left(h + t^* \left(\mu_x^0 \tilde{\mathbf{d}}_0 + \mu_x^1 \tilde{\mathbf{d}}_1 + \mu_x^G \right) \right)$$

for some $\eta \in \{\pm 1\}$, where $(\mu_x^0, \mu_x^1, \mu_x^G)$ are the coefficients corresponding to D^*. Equating coefficients on both sides, we find:

$$B_0^i = \eta s^{*-1} t^* \mu_x^0, \qquad B_1^i = \eta s^{*-1} t^* \mu_x^1, \qquad A^i = \eta s^{*-1} \left(h + t^* \mu_x^G \right). \tag{21}$$

Since η, s^*, t^*, and μ_x^1 are all non-zero, we conclude that $B_1^i \neq 0$. Dividing the expressions in 21, we obtain:

$$\frac{B_0^i}{B_1^i} = \frac{\mu_x^0}{\mu_x^1}, \qquad \frac{A^i}{B_1^i} = \frac{h + t^* \mu_x^G}{t^* \mu_x^1} \implies t^* h^{-1} \left(\frac{A^i \mu_x^1}{B_1^i} - \mu_x^G \right) = 1. \tag{22}$$

To complete the bound on $\Pr\left[B_{i,h}^{\mathrm{II}}\right]$, observe the following:

- Since the ratio μ_x^0/μ_x^1 uniquely identifies the triplet $(\mu_x^0, \mu_x^1, \mu_x^G)$ (by the assumption that Z_{II} does not occur), equation (22) determines a unique choice of the triplet.
- Given this unique triplet, the second equation in (22) determines a unique value for t^*.

But since t^* is sampled uniformly from $\mathbb{F}_q$ (and independently of the coefficients A^i, B_0^i, B_1^i and of the triplet $(\mu_x^0, \mu_x^1, \mu_x^G)$), the probability that it satisfies the identity in (22) is at most $\mathcal{O}\left(\frac{1}{q}\right)$.

We conclude that $\Pr\left[B_{i,h}^{\mathrm{II}}\right] = \mathcal{O}\left(\frac{1}{q}\right)$, completing the proof that

$$\Pr[\text{Type II forgery}] = \mathcal{O}\left(\frac{N^2}{q}\right).$$

E Performance Full Data

We thereby give the full experimental data. For the local network experiment data is given for presign and sign in Table 2 and Table 3 respectively. Client running times are given in Table 4. For the live experiment the data is given in Table 6 and Table 5 and summarized in Table 7.

Number of Parties	Threshold	Mean Time (s)	Std Dev (s)
12	9	2.102	0.071
30	21	3.020	0.157
45	31	3.333	0.154
60	41	3.732	0.238
85	57	4.274	0.292
100	67	4.584	0.286
120	81	5.221	0.581

Table 2: Presign Protocol Running Time for ECDSA secp256k1 using Class Groups. Results averaged over 10 repetitions.

Parties	Threshold	Client Mean (s)	Client Std (s)	Node Mean (s)	Node Std (s)
12	9	1.926	0.059	0.382	0.011
30	21	1.921	0.046	0.428	0.009
45	31	1.926	0.027	0.482	0.007
60	41	1.921	0.032	0.554	0.012
85	57	1.908	0.019	0.685	0.020
100	67	1.959	0.085	0.778	0.019
120	81	1.932	0.061	0.938	0.039

Table 3: Sign Protocol Running Time for ECDSA secp256k1 using Class Groups. Results averaged over 10 repetitions. Client denotes centralized party, Node denotes decentralized party.

Parties	Client Mean (s)	Client Std (s)	Notes
12	1.926	0.059	Overall Mean: 1.928 s
30	1.921	0.046	Overall Std Dev: 0.016 s
45	1.926	0.027	CV: 0.8%
60	1.921	0.032	
85	1.908	0.019	
100	1.959	0.085	
120	1.932	0.061	

Table 4: Client computation time independence from party count. Sign protocol demonstrates constant client overhead regardless of network size, with coefficient of variation $< 1\%$, confirming scalability to large validator sets.

Validator	**Presign (ms)**	**Validator**	**Presign (ms)**
Validator 1	28,867	Validator 44	10,552
Validator 2	24,626	Validator 45	13,903
Validator 3	1,692	Validator 46	20,144
Validator 4	19,041	Validator 47	10,943
Validator 5	11,451	Validator 48	10,204
Validator 6	18,895	Validator 49	12,272
Validator 7	9,732	Validator 50	30,496
Validator 8	1,679	Validator 51	25,470
Validator 9	17,215	Validator 52	23,567
Validator 10	10,866	Validator 53	10,134
Validator 11	21,933	Validator 54	17,461
Validator 12	7,175	Validator 55	11,392
Validator 13	19,305	Validator 56	12,120
Validator 14	25,174	Validator 57	36,546
Validator 15	16,436	Validator 58	15,694
Validator 16	35,783	Validator 59	21,392
Validator 17	1,646	Validator 60	20,510
Validator 18	23,343	Validator 61	14,504
Validator 19	29,277	Validator 62	10,104
Validator 20	21,454	Validator 63	14,333
Validator 21	12,606	Validator 64	27,246
Validator 22	18,449	Validator 65	18,475
Validator 23	1,696	Validator 66	18,651
Validator 24	25,182	Validator 67	2,882
Validator 25	20,579	Validator 68	12,059
Validator 26	25,248	Validator 69	6,960
Validator 27	26,737	Validator 70	11,194
Validator 28	29,214	Validator 71	12,120
Validator 29	27,156	Validator 72	11,738
Validator 30	27,189	Validator 73	11,753
Validator 31	16,875	Validator 74	55,079
Validator 32	19,091	Validator 75	11,611
Validator 33	13,837	Validator 76	10,904
Validator 34	18,852	Validator 77	9,658
Validator 35	1,648	Validator 78	11,996
Validator 36	16,226	Validator 79	19,001
Validator 37	20,076	Validator 80	23,095

Table 5: Presign protocol computation times across validators on Narwhal consensus.

Validator	Sign (ms)	Validator	Sign (ms)
Validator 1	865	Validator 44	750
Validator 2	855	Validator 45	1,069
Validator 3	574	Validator 46	989
Validator 4	780	Validator 47	1,331
Validator 5	1,311	Validator 48	1,131
Validator 6	773	Validator 49	1,285
Validator 7	1,205	Validator 50	991
Validator 8	658	Validator 51	867
Validator 9	797	Validator 52	1,018
Validator 10	829	Validator 53	1,161
Validator 11	858	Validator 54	815
Validator 12	818	Validator 55	969
Validator 13	887	Validator 56	1,561
Validator 14	885	Validator 57	2,774
Validator 15	1,456	Validator 58	1,195
Validator 16	1,116	Validator 59	1,674
Validator 17	577	Validator 60	757
Validator 18	1,021	Validator 61	1,480
Validator 19	822	Validator 62	1,289
Validator 20	847	Validator 63	1,130
Validator 21	760	Validator 64	799
Validator 22	802	Validator 65	823
Validator 23	556	Validator 66	838
Validator 24	857	Validator 67	856
Validator 25	875	Validator 68	937
Validator 26	850	Validator 69	984
Validator 27	859	Validator 70	985
Validator 28	923	Validator 71	1,022
Validator 29	848	Validator 72	973
Validator 30	1,857	Validator 73	3,868
Validator 31	915	Validator 74	954
Validator 32	808	Validator 75	1,288
Validator 33	1,026	Validator 76	1,280
Validator 34	808	Validator 77	1,552
Validator 35	563	Validator 78	858
Validator 36	842	Validator 79	867
Validator 37	1,427		

Table 6: Sign protocol computation times across validators on Narwhal consensus.

Statistic	Sign (s)	Presign (s)
Mean	1.05	17.3
Std Dev	0.43	8.7
Median	0.86	17.0
Min	0.56	1.6
Max	3.9	55
CV (%)	42.8	48.7

Table 7: Summary statistics for Sign and Presign protocol computation times across 85 validators. CV = Coefficient of Variation (Std Dev / Mean × 100).

Protocol	Computation (s)	Total Time (s)	Communication (s)	Comm. %
Sign	0.939	2.059	1.121	54.4%
Presign	7.769	66.330	58.561	88.3%

Table 8: Communication overhead analysis for ECDSA Secp256k1 protocols. Measurements from c4d-standard-32 hardware running two validators. Communication time computed as difference between total protocol time and computation time.

F Instantiating TAHE

This appendix discusses concrete instantiations of the ideal functionality $\mathcal{F}_{\mathsf{TAHE}}$ (Functionality 5) used by our protocols. At a high level, our use of TAHE requires: (i) threshold decryption with public verifiability of decryption shares (for robustness against malicious validators), and (ii) secure linear evaluation / linear circuit privacy (Definition 6), since the protocol homomorphically combines ciphertexts that may originate from adversarial parties.

We briefly compare three families of candidates: Paillier-based schemes, class-group schemes, and lattice-based schemes. We also discuss a practical optimization for threshold decryption in the signature-generation setting.

F.1 Candidate TAHE Families

Paillier-based TAHE. Threshold Paillier decryption is naturally compatible with asynchronous participation (e.g., once $t+1$ decryption shares are available, reconstruction can proceed). A key practical challenge is that *distributed RSA modulus generation* is typically described as a multi-round MPC protocol and is commonly instantiated under synchronous assumptions. One pragmatic mitigation is to treat the RSA modulus as a long-lived setup artifact (generated once), and then derive fresh encryption keys for epochs on top of that modulus when possible. For the classical Paillier ciphertext space, every element is syntactically a ciphertext, which simplifies handling of malformed inputs relative to Definition 6. On the other hand mismatch between the plaintext of the encryption scheme and the elliptic curve modulus causes some other issues on this front. This issue is solvable via adding a random multiple of the elliptice curve modulus of the correct size (See also the circuit-privacy discussion in [46].)

Class-group TAHE. Class-group based constructions (e.g., [18]) provide an ElGamal-like structure in an unknown-order group and support efficient threshold decryption. While some presentations describe key generation synchronously, the underlying ingredients can often be adapted to an asynchronous setting by (a) avoiding commit-and-open patterns and (b) using an asynchronous agreement primitive (e.g., $\mathcal{F}_{\mathsf{ACS}}$) to select

a qualified set of contributions. We outline such an "ElGamal-style" asynchronous key-generation template in Section F.2.

A subtlety is that some robustness arguments in the literature invoke honest-majority assumptions for complaint mechanisms and guaranteed output delivery of the DKG/VSS layer. In our setting, protocol security already assumes $t < n/3$ due to reliable broadcast consistency constraints, but one may still want the TAHE threshold to be larger in practice. If one is willing to trade off some liveness guarantees of the key generation subroutine (while retaining correctness and privacy), this gap can sometimes be addressed by using PVSS-style techniques in place of interactive complaints.

Lattice-based threshold encryption. Lattice-based threshold encryption is considerably more complex to integrate in our setting. Existing threshold-decryption protocols are often synchronous (or require heavy interaction), and achieving efficient public verifiability of shares together with good communication complexity remains challenging. Moreover, secure linear evaluation typically requires explicit circuit-privacy measures (e.g., noise flooding), and proofs of correct evaluation are often used in practice, precisely the type of client-facing proof machinery our 2PC-MPC design tries to avoid.

That said, lattice-based approaches may become attractive if one wishes to support limited multiplicative depth homomorphically (beyond affine maps). This could, in principle, enable additional preprocessing patterns, but this is outside the scope of our current implementation.

F.2 El-Gamal Style Asynchronous Key Generation and Secure Function Evaluation

We use the following notations:

1. We use multiplicative notation.
2. $\mathbb{H}$ and $\mathbb{F} \leq \mathbb{H}$: finite abelian groups. $\mathbb{H}$ is of unknown order and $\mathbb{F}$ is of known order q and admits an efficient algorithm for computing discrete logs in the subgroup.
3. $\mathcal{D}_\kappa$: a probability distribution such that for $g \leftarrow \mathbb{H}/\mathbb{F}, r \leftarrow \mathcal{D}_\kappa$ then g^r is indistinguishable from random.

Let us start with Lemma for secure function evaluation:

Lemma 11 *Defining* Eval *by* $\mathsf{ct}_0^{\alpha+r_\alpha q} \cdot \mathsf{ct}_1^{\beta+r_\beta q} + \mathsf{Enc}_{\mathsf{pk}}(0; r)$ *where* $r_\alpha, r_\beta, r \leftarrow \mathcal{D}_\kappa$ *satisfy secure function evaluation for ElGamal style encryption.*

Proof. Let $\mathsf{ct}_0 = (g_0, \bar{g}_0), \mathsf{ct}_1 = (g_1, \bar{g}_1)$. Note that each of the elements can be composed to give $g_0 = g_{0,q} f^{a_0}, \bar{g}_0 = \bar{g}_{0,q} f^{\bar{a}_0}, g_1 = g_{1,q} f^{a_1}, \bar{g}_1 = \bar{g}_{1,q} f^{\bar{a}_1}$ then evaluation with α, β and randomness r will give $(g^r \cdot g_{0,q}^{\alpha+r_\alpha q} g_{1,q}^{\beta+r_\beta q} f^{\alpha a_0 + \beta a_1}, g^{rs} \cdot \bar{g}_{0,q}^{\alpha+r_\alpha q} \bar{g}_{1,q}^{\beta+r_\beta q} f^{\alpha \bar{a}_0 + \beta \bar{a}_1})$. First we note that $g_{0,q}, g_{1,q}, \bar{g}_{0,q}, \bar{g}_{1,q} \neq 1$ as this can be check publicly by exponentiation by q. since $gcd(\mathbb{H}/\mathbb{F}, q) = 1$ thus whatever subgroup of $\mathbb{H}/\mathbb{F}$ the elements $g_{0,q}, \bar{g}_{0,q}, g_{1,q}, \bar{g}_{1,q}$ are in the result of the exponentiation by $\alpha + r_\alpha q$ or $\beta + r_\beta q$ is indistinguishable from random in the subgroup. Thus getting pt_f the simulator may send $g^{r'}, g^{r's} f^{\mathsf{pt}_f}$. Both elements of the ciphertext will be indistinguishable by the subgroup indistingusihability property (the first will be in $\mathbb{H}/\mathbb{F}$ instead of $\mathbb{H}$ and the second the other way around). In the case that $\mathsf{pt}_f = \perp$ the simulator will randomize γ and will send $g^{r'}, g^{r's} f^{\gamma}$ the first coordinate will be indistingusihable as first from the low order assumption and then from the subgroup indistinguishability property. This is since the elements $g_{0,q}, g_{1,q}, \bar{g}_{0,q}, \bar{g}_{1,q}$ will have large enough order such that extracting g^r will be impossible.

In the following we give a general overview for an El-Gamal style asynchronous key generation 9 .

Correctness: We have $\mathsf{pk} = g^{\sum_{j \in S_B} \alpha_j}$ and $\sum_{j \in S_B} \lambda_{j,0}^{S_B} [\mathsf{sk}]_j = \Delta \mathsf{sk}$ and $\mathsf{vk}_j = g^{\sum_{j' \in S} f_{j'}(j)} = g^{[\mathsf{sk}]_j}$.

Simulation: The simulator requests a pubic key from the functionality receiving pk. It then randomizes a value α_i for every honest party similarly to the honest protocol. It then sends $C_{i,j} = g^{[\alpha_i]_j} + \mathsf{pk}$ and cheats in the zk-proof. Upon receiving the shares on behalf of the malicious parties it follows the honest protocol by validating the proofs and emulating $\mathcal{F}_{\mathsf{ACS}}$. It then uses the secret keys to the public encryptions sk_i to extract $t+1$ valid shares on α_i for every malicious party and thus can reconstruct α_i. It then request from the functionality to change the public key to be $\mathsf{pk}^{|S_{B'} \setminus U|} \cdot g^{-\sum_{j \in S_B} \alpha_j}$.
Indistinguishability: The messages sent by the honest party are indistinguishable based on the hiding property of the commitment scheme, the zk property of the zk proof and the security of the Shamir Secret Sharing over the integers.

F.3 Reconstruction Optimization

As shown in [46] in the case of signature generation it forgo the zk proofs needed during threshold decryption and for a single participant to combine the decryption shares and broadcast the result. Then proofs will be sent only if the resulting decryption is not a valid signature. Unfortunately in the asynchronous setting there is a clear problem, the participant which performs the recombination may not be available for the broadcast round and parties may not be available for sending their proofs in case of a failure. The first issue can be solved using a leader election algorithm [6] and the second can be solved by sending the zk proofs during threshold decryption but verifying them only in the case of failure. In some setting such as proof-of-stake blockchains leader election is often needed anyways and thus this would not affect performance significantly. On the contrast in system with small number of participants but high network latency it may be better for each party to locally recombine the decryption shares. Thus the distributed decryption looks as follows:

1. Generate the decryptions shares ds_i and proofs π_{ds}^i and broadcast them.
2. Elect a leader.
3. The elected leader recombines the shares and checks if the signature verifies. If it verifies it broadcasts it, else it verifies the proofs until collecting t+1 valid shares. At this points it calculates a correct signature along with a message cheaters and the failed proofs.
4. The rest of the parties upon receiving a signature check its validity, if it is not valid the consider the leader malicious. Upon receiving a message cheaters they verify the proofs for the decryption shares and consider any party for which the proofs failed to be malicious.

G Schnorr-Based Threshold Signature Protocol

In this appendix we provide the full description and security analysis of our Schnorr-based threshold signing protocol. We first describe the honest two-party protocol and its extension to the distributed setting, then prove oracle-level security via a reduction to the Algebraic One More Discrete Log (AOMDL) problem, and finally show that

the full protocol UC-realizes the threshold signature functionality in the $\mathcal{F}_{\mathsf{TAHE}}$-hybrid model.

Throughout, we work in the same group setting, notation, and hybrid model as in the ECDSA protocol from Section 4, and we rely on the same distributed additive homomorphic encryption functionality $\mathcal{F}_{\mathsf{TAHE}}$ and reliable broadcast.

G.1 Protocol Description

Honest Two-Party Protocol We begin by describing the honest two-party protocol between a distributed party B and a centralized party A.

1. **Distributed Key Generation**: Run the distributed key generation as in Section 4, instantiated with the special choice
$$x_{1,B} = 0, \qquad \mu_x^1 = 1, \qquad \mu_x^G = 0,$$
so that the resulting key shares match the Schnorr structure.
2. **Presign**: The distributed party B samples $k_0, k_1 \leftarrow \mathbb{Z}_q$ and computes
$$K_{B,0} = k_0 \cdot G, \qquad K_{B,1} = k_1 \cdot G,$$
which it sends to A.
3. **Sign**:
   (a) A samples $k_A \leftarrow \mathbb{Z}_q$ and computes
   $$K_A = k_A \cdot G.$$
   It then calls the random oracle on the entire transcript to obtain the randomizer μ_k.
   (b) A computes
   $$K = K_{B,0} + \mu_k \cdot K_{B,1} + K_A.$$
   It then calls $\mathcal{H}(K, X, \mathsf{msg})$ and receives the challenge e.
   (c) A computes the response
   $$z_A = k_A + e \cdot x_A,$$
   and sends K_A, z_A to B.
   (d) B also calls the random oracle to obtain μ_k, e and recomputes K. It then computes
   $$z = z_A + k_0 + \mu_k \cdot k_1 + e \cdot x_B,$$
   and outputs the Schnorr signature (z, e).

As in the ECDSA setting, we make use of $\mathcal{F}_{\mathsf{TAHE}}$ to comply with the 2PC-MPC design. In contrast to the ECDSA protocol, here we do not need zero-knowledge proofs to enforce honest behavior of A in the signing phase (beyond what is already required for key generation). This is inherent to the structure of Schnorr signatures: the message-dependent part of A's contribution already constitutes a proof of knowledge of $\log_G K_A$.

Distributed Presign Protocol We now describe the distributed presign protocol in the multi-party setting. The presign protocol consists of a single aggregation round executed by the distributed party B, which collectively generates

$$K_{B,0},\ K_{B,1},\ \mathsf{ct}_{k_0} = \mathsf{Enc}(k_0),\ \mathsf{ct}_{k_1} = \mathsf{Enc}(k_1).$$

Only the group elements $K_{B,0}$ and $K_{B,1}$ are sent to A. The ciphertexts ct_{k_0} and ct_{k_1} remain local to B and are used in the signing protocol to produce the distributed party's encrypted contribution to the signature.[5]

Notably, the presign output is independent of the public key X. Hence, if the distributed party B is a system that interacts with several clients A_i, then the same presign output can be used with any such client to sign (distinct) messages. Each presign, however, must still be used at most once.

Distributed Signing Protocol In the distributed signing protocol, A behaves exactly as in the honest two-party protocol described above. Upon receiving (K_A, z_A) and the message msg, the distributed party B proceeds as follows:

- It first verifies that
$$K_A + eX_A \stackrel{?}{=} z_A \cdot G.$$
If the check fails, it aborts.
- It then uses ct_{k_0}, ct_{k_1}, and $\mathsf{ct}_{\mathsf{key}}$ to compute its encrypted share of the signature, and decrypts it using $\mathcal{F}_{\mathsf{TAHE}}$ to obtain z_B.
- Finally, B computes the response
$$z = z_A + z_B$$
and outputs the signature (z, e).

G.2 Oracle Security of the Schnorr Signing Oracle

In this section we analyze the security of the Schnorr signing oracle and reduce signature forgery to breaking the Algebraic One More Discrete Log (AOMDL) problem.

We consider the slightly enhanced Schnorr-based signing oracle $\mathcal{G}^*_{\mathsf{SE\text{-}Sch}}$ (Functionality 12) and the corresponding existential unforgeability experiment $\mathsf{Exp}^{\mathcal{G}^*_{\mathsf{SE\text{-}Sch}}}_{\mathsf{EU}}$. We show that any efficient forger against this oracle can be turned into an efficient adversary against the AOMDL experiment $\mathsf{Exp}_{\mathsf{AOMDL}}$ (Experiment 15).

In the AOMDL security game, the challenger generates $n_{\mathsf{DL}} + 1$ group elements and then answers at most n_{DL} queries from the adversary, where each query requests a linear combination of the discrete logs of these group elements. To win, the adversary must output all $n_{\mathsf{DL}} + 1$ discrete logs.

Theorem 12 *If there is a PPT algorithm $\mathcal{A}^{n_s,n_{kg},n_r}_{\mathsf{SE-Sch-Forger}}$ that satisfies $\mathsf{Exp}^{\mathcal{G}^*_{\mathsf{SE\text{-}Sch}}}_{\mathsf{EU}}(\mathcal{A}, 1^\kappa, n_s, n_{kg})$ with probability at least ϵ, and uses at most n_r queries to the random oracle, then there*

[5] In the purely two-party case, where B is a centralized entity, these ciphertexts are not needed.

exists a PPT algorithm $\mathcal{A}'_{2n_s+n_{kg}}$ *that satisfies* $\mathsf{Exp}^{2n_s+n_{kg}}_{\mathsf{AOMDL}}(\mathcal{A}, 1^\kappa, (\mathbb{G}, G, q))$ *with probability at least*

$$\epsilon\left(\frac{\epsilon}{n_r} - \frac{1}{|H|}\right)\left(1 - \frac{n_s}{|H|}\right),$$

and expected running time

$$2\ TIME(\mathcal{A}^{n_s,n_{kg},n_r}_{\mathsf{SE-Sch-Forger}}) + \mathcal{O}(n_r + n_s + n_{kg}).$$

Intuitively, the reduction works as follows. The adversary $\mathcal{A}'_{2n_\mathsf{s}+n_\mathsf{kg}}$ simulates $\mathcal{G}^*_{\mathsf{SE\text{-}Sch}}$ when interacting with $\mathcal{A}^{n_\mathsf{s},n_\mathsf{kg},n_\mathsf{r}}_{\mathsf{SE-Sch-Forger}}$, and uses its access to the AOMDL challenger as a discrete-log oracle on carefully chosen linear combinations.

The reduction receives $n_{\mathsf{DL}} + 1 = 2n_\mathsf{s} + n_\mathsf{kg}$ group elements from the AOMDL challenger. The first $2n_\mathsf{s}$ elements are used as the nonces for presign queries, and the last n_kg elements are used as public keys. Upon receiving a signing request from $\mathcal{A}^{n_\mathsf{s},n_\mathsf{kg},n_\mathsf{r}}_{\mathsf{SE-Sch-Forger}}$, the reduction computes the randomizer μ_k and the challenge e using the random oracle, and then observes that the resulting signature z is a fixed linear combination in $(1, \mu_k, e)$ of the discrete logs of the corresponding presigns and public key. A single discrete-log query to the AOMDL challenger on this linear combination suffices to produce a correct signature.

To break AOMDL, we then extract new discrete logs from a forgery of $\mathcal{A}^{n_\mathsf{s},n_\mathsf{kg},n_\mathsf{r}}_{\mathsf{SE-Sch-Forger}}$. A Schnorr forgery $(K^*, X, \mathsf{msg}^*, e^*, z^*)$ yields a linear equation in the discrete logs of the public key and the (possibly new) nonce K^*. Since K^* may not appear among the original AOMDL challenges, we treat its discrete log k^* as a fresh variable. By rewinding $\mathcal{A}^{n_\mathsf{s},n_\mathsf{kg},n_\mathsf{r}}_{\mathsf{SE-Sch-Forger}}$ to the point where e^* was obtained from the random oracle and returning an independently sampled challenge $e^{**} \neq e^*$, we obtain (with non-negligible probability, by the Generalized Forking Lemma [9, Lemma 1]) a second forgery with the same public key and the same ephemeral key K^*. From the two linearly independent equations we can extract the discrete log of the public key, thereby winning the AOMDL game.

Supporting Definitions and Lemma We now detail the ingredients used in the proof of Theorem 12. We first define the slightly enhanced Schnorr-based oracle and its non-biased counterpart, as well as the unforgeability and AOMDL experiments. We also recall the Generalized Forking Lemma.

Lemma 13 (Generalized Forking Lemma [9, Lemma 1]) *Let* $n_r \geq 1$ *be an integer. Let* $\mathcal{A}_{n_r}(inp, h_1, \ldots, h_{n_r}; \rho)$ *be a randomized algorithm with randomness* ρ*, and assume that it outputs either* (ℓ, out) *for some* $1 \leq \ell \leq n_r$ *or* $\perp$*. Let* H *be the set of possible responses of the random oracle, and assume that the random oracle samples uniformly from* H*. Let* $\mathsf{acc}(\mathcal{A}_{n_r})$ *denote the probability, over* $inp \leftarrow \mathsf{InpGen}$*,* $h_1, \ldots, h_{n_r} \leftarrow H$*, and* $\rho \leftarrow R$*, that* $\mathcal{A}_{n_r}(inp, h_1, \ldots, h_{n_r}; \rho)$ *does not return* $\perp$*. Consider* $\mathsf{Fork}_{\mathcal{A}_{n_r}}(h_1, h'_1, \ldots, h_{n_r}, h'_{n_r})$ *as in Simulation 14. Then the probability that* $\mathsf{Fork}_{\mathcal{A}_{n_r}}(h_1, h'_1, \ldots, h_{n_r}, h'_{n_r})$ *does not return* $\perp$*, where* $h_1, \ldots, h_{n_r}$ *and* $h'_1, \ldots, h'_{n_r}$ *are sampled uniformly from* H*, is at least*

$$\mathsf{acc}(\mathcal{A}_{n_r})\left(\frac{\mathsf{acc}(\mathcal{A}_{n_r})}{n_r} - \frac{1}{|H|}\right).$$

Proof of Theorem 12 We now give the detailed proof of Theorem 12.

Proof (Proof of Theorem 12). Let $\mathcal{A}^{n_s,n_{kg},n_r}_{\mathsf{SE-Sch-Forger}}$ be a PPT algorithm that satisfies $\mathsf{Exp}^{\mathcal{G}^*_{\text{Non-Biased-SE-Sch}}}_{\mathsf{EU}}(\mathcal{A}, 1^\kappa, n_s, n_{kg})$ with probability at least ϵ, and that uses at most n_r calls to the random oracle $\mathcal{H}$. We define the associated algorithms $\mathcal{S}^{\mathsf{Exp}_{\mathsf{AOMDL}}}_{\mathsf{Exp}_{\mathsf{EU}}(\mathcal{A}^{n_s,n_{kg},n_r}_{\mathsf{SE-Sch-Forger}})}$ and $\mathcal{A}^{\mathcal{S}_{\mathsf{Exp}_{\mathsf{EU}}}}_{\mathsf{AOMDL-Sch}}$ as in the reduction, and observe that $\mathcal{A}^{\mathcal{S}_{\mathsf{Exp}_{\mathsf{EU}}}}_{\mathsf{AOMDL-Sch}}$ up to command line 8 is equivalent to the forking algorithm $\mathsf{Fork}_{\mathcal{S}^{\mathsf{Exp}_{\mathsf{AOMDL}}}_{\mathsf{Exp}_{\mathsf{EU}}(\mathcal{A}^{n_s,n_{kg},n_r}_{\mathsf{SE-Sch-Forger}})}}$.

Hence, by the Generalized Forking Lemma (Lemma 13), the success probability of $\mathcal{A}^{\mathcal{S}_{\mathsf{Exp}_{\mathsf{EU}}}}_{\mathsf{AOMDL-Sch}}$ up to command line 8 is at least

$$\epsilon\left(\frac{\epsilon}{n_r} - \frac{1}{|H|}\right).$$

It remains to analyze the probability that the linear systems produced by the two successful executions of $\mathcal{S}^{\mathsf{Exp}_{\mathsf{AOMDL}}}_{\mathsf{Exp}_{\mathsf{EU}}(\mathcal{A}^{n_s,n_{kg},n_r}_{\mathsf{SE-Sch-Forger}})}$ are linearly independent, which in turn depends on whether some presign nonce shares the same value of μ in both executions.

The sampled vectors are linearly dependent if and only if a presign has the same value of μ for the two signatures. Since the value μ is obtained from the random oracle and lives in H, the probability that such a collision occurs across n_s signing queries is at most $n_s/|H|$. Therefore, the overall success probability of $\mathcal{A}^{\mathcal{S}_{\mathsf{Exp}_{\mathsf{EU}}}}_{\mathsf{AOMDL-Sch}}$ is at least

$$\epsilon\left(\frac{\epsilon}{n_r} - \frac{1}{|H|}\right)\left(1 - \frac{n_s}{|H|}\right).$$

The expected running time of $\mathcal{A}^{\mathcal{S}_{\mathsf{Exp}_{\mathsf{EU}}}}_{\mathsf{AOMDL-Sch}}$ is bounded by

$$2\,\mathrm{TIME}(\mathcal{A}^{n_s,n_{kg},n_r}_{\mathsf{SE-Sch-Forger}}) + \mathcal{O}(n_r + n_s + n_{kg}),$$

as required.

G.3 UC-Realization of the Schnorr-Based Protocol

We now turn to the UC-security of the Schnorr-based threshold protocol in the $\mathcal{F}_{\mathsf{TAHE}}$-hybrid model.

Theorem 14 *The protocols Π_{keygen}, $\Pi^{Schnorr}_{presign}$, and $\Pi^{Schnorr}_{sign}$ UC-realize $\mathcal{F}^{\mathcal{G}^*_{SE\text{-}Sch},\binom{[n]}{t+1}}_{tsig}$ (Functionality 1 with Oracle 12) in the* broadcast model *and $\mathcal{F}_{TAHE}$-hybrid model.*

Proof of Theorem 14 We now provide the full proof of Theorem 14 by describing a simulator $\mathcal{S}$ and arguing indistinguishability of the real and ideal executions.

Proof. The simulator $\mathcal{S}$ interacts with the ideal functionality $\mathcal{F}^{\mathcal{G}^*_{\mathsf{SE\text{-}Sch}}}_{\mathsf{tsig}}$ and with the adversary, and internally emulates the honest parties.

Presign Simulation. The simulator $\mathcal{S}$ sends $(\mathsf{pres}, \mathsf{sid}, \mathsf{ssid}, i)$ to $\mathcal{F}^{\mathcal{G}^*_{\mathsf{SE\text{-}Sch}}}_{\mathsf{tsig}}$ for each $i \in U$ and receives $(\mathsf{pres}, \mathsf{sid}, \mathsf{ssid}, K_0, K_1)$ in return.

Case 1: Malicious B. We assume, without loss of generality, that all parties in B are corrupted. Therefore, the simulator only needs to simulate the behavior of

A. Upon receiving $(\mathsf{global\text{-}broadcast}, \mathsf{sid}, (\mathsf{pid}_A, K_{B,0}, K_{B,1}))$, the simulator verifies that $K_{B,0}, K_{B,1} \in \mathbb{G} \setminus \{0\}$; otherwise, it aborts.

Case 2: Malicious A_{pid_A}. In this case, the distributed party B is honest and must be simulated towards the adversarial centralized party. The simulator proceeds as follows:

1. For each $i \in B \setminus U$, the simulator samples
$$k_0^i, k_1^i \leftarrow \mathbb{Z}_q, \qquad \eta_0^i, \eta_1^i \leftarrow \mathcal{R}_{pk}.$$
It sets
$$K_{B,0}^i = K_0 + k_0^i \cdot G, \qquad K_{B,1}^i = K_1 + k_1^i \cdot G.$$
The simulator then invokes the zero-knowledge property to generate simulated (fake) proofs for the aggregation protocol and uses these values as inputs for the aggregation protocol with the adversary.
2. During the aggregation protocol, $\mathcal{S}$ receives $\mathsf{ct}_{k_0}^i, \mathsf{ct}_{k_1}^i$ for each $i \in U$, accompanied by zero-knowledge proofs, which it verifies. If a proof fails, the simulator flags i as malicious. Otherwise, it decrypts the ciphertexts to obtain k_0^i and k_1^i.
3. The simulator emulates $\mathcal{F}_{\mathsf{ACS}}$ and receives an authorized subset S_B from the adversary. This subset contains either malicious parties with validated proofs or honest parties. The simulator sets
$$k_0' = \sum_{i \in S_B} k_0^i, \qquad k_1' = \sum_{i \in S_B} k_1^i.$$
4. The simulator emulates $\mathcal{F}_{\mathsf{global\text{-}broadcast}}$ and sends the correct values $(K_{B,0}, K_{B,1})$ (as outputted by the aggregation protocol) to the adversary.

Simulation for Signing.

Case 1: Malicious B.

1. The simulator sends $(\mathsf{sign}, \mathsf{msg}, \mathsf{sid}, \mathsf{ssid}, (1,0,1,0))$ to $\mathcal{F}_{\mathsf{tsig}}^{\mathcal{G}^*_{\mathsf{SE\text{-}Sch}}}$ and receives a Schnorr signature $\sigma = (z, e)$ with public nonce K.
2. The simulator samples $\mu_k, e, z_A \leftarrow \mathbb{Z}_q$ uniformly at random, sets
$$K_A = z_A \cdot G - e \cdot X_A,$$
and defines
$$K = K_{B,0} + \mu_k \cdot K_{B,1} + K_A.$$
It programs the random oracle $\mathcal{H}$ to return μ_k on input $(\mathsf{sid}, X, \mathsf{pres}_{X,\mathsf{sid}}, K_A, \mathsf{msg})$ and e on input (K, X, msg).
3. The simulator sends $(\mathsf{broadcast}, \mathsf{sid}, (z_A, K_A))$ to the adversary.
4. Upon receiving the final signature σ from the ideal functionality, the simulator verifies its validity. If the signature fails verification, it aborts.

Case 2: Malicious A_{pid_A}.

1. Upon receiving (z_A, K_A) from the adversary, the simulator checks that
$$z_A \cdot G \stackrel{?}{=} K_A + e \cdot X_A.$$
If the check fails, it aborts.

2. The simulator calculates
$$k_A = z_A - ex_A,$$
where x_A was previously extracted during the key generation simulation. It then computes the coefficients
   (a) $\alpha_0 = n_H$,
   (b) $\alpha_1 = n_H$,
   (c) $\beta_0 = k'_0$,
   (d) $\beta_1 = k'_1$,
   corresponding to the aggregation of honest parties' contributions.
3. The simulator sends $(\mathsf{sign}, \mathsf{msg}, \mathsf{sid}, \mathsf{ssid}, (\alpha_0, \beta_0, \alpha_1, \beta_1))$ to $\mathcal{F}_{\mathsf{tsig}}^{\mathcal{G}^*_{\text{SE-Sch}}}$ and receives a valid signature (z, e).
4. The simulator emulates $\mathcal{F}_{\mathsf{TAHE}}$ to return z as the plaintext to the honest parties.
5. Finally, the simulator emulates $\mathcal{F}_{\text{global-broadcast}}$ to send $(\mathsf{global\text{-}broadcast}, \mathsf{sid}, (z, e))$ to the adversary.

Indistinguishability.

The adversary's view in the simulated execution consists of

$$(X, \pi_{\mathsf{DL}}, z_A, K_A)$$

together with the random oracle outputs e and μ_k. All these values are uniformly distributed (subject to the algebraic relations enforced by the protocol) and satisfy the adversary's checks, in particular that the zero-knowledge proof π_{DL} verifies and that

$$z_A \cdot G = K + e \cdot X_A.$$

Thus, the only potential distinguishing advantage comes from the distribution of abort events.

The only difference in abort behavior between the real and ideal executions can occur during verification of the final signature, that is, when the adversary forges a valid Schnorr signature that is not consistent with the ideal functionality. Assuming such an adversary exists, we can construct a forger that breaks the existential unforgeability of the underlying Schnorr scheme. Since the messages in the protocol are indistinguishable from those produced by a standard Schnorr signing oracle, any valid signature on a message $\mathsf{msg}' \neq \mathsf{msg}$ that was not legitimately signed can be used to break the unforgeability assumption.

We conclude that the view of any environment in the real execution is indistinguishable from its view in the ideal execution, and therefore the protocols Π_{keygen}, $\Pi^{\mathsf{Schnorr}}_{\mathsf{presign}}$, and $\Pi^{\mathsf{Schnorr}}_{\mathsf{sign}}$ UC-realize $\mathcal{F}_{\mathsf{tsig}}^{\mathcal{G}^*_{\text{SE-Sch}}, \binom{[n]}{t+1}}$ in the stated hybrid model.

Protocol 4: Sign $\Pi_{\mathsf{Sign}}^{\mathsf{ECDSA}}$
$\mathsf{Sign}(\mathbb{G}, G, H, q, \mathsf{sid}, \Gamma_B, \mathsf{pk}, X_A, \mu_x^0, \mu_x^1, \mu_x^G, \mathsf{global\text{-}keygen}, \mathsf{pres}, \mathsf{msg})$

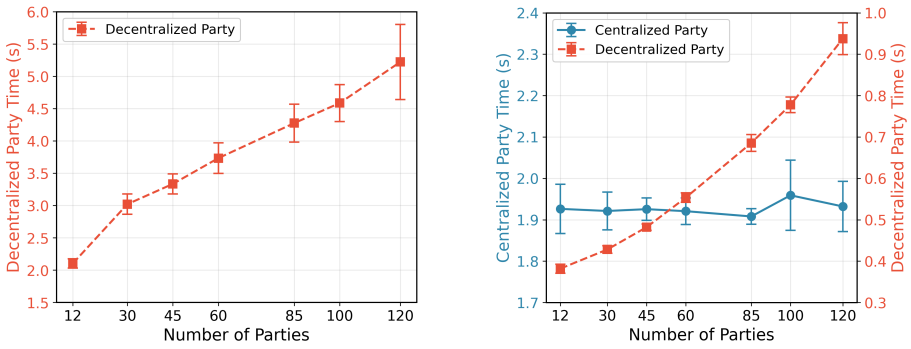
1. A_{pid_A} **does as follows**:
   (a) Calculate $\mathsf{ct}_{\gamma\cdot\mathsf{key}} = \mu_x^0 \odot \mathsf{ct}_{\gamma_B\cdot\mathsf{key},0} \oplus \mu_x^1 \odot \mathsf{ct}_{\gamma_B\cdot\mathsf{key},1} \oplus \mu_x^G \odot \mathsf{ct}_\gamma$ and $X = X_A + \mu_x^0 X_{0,B} + \mu_x^1 \cdot X_{1,B} + \mu_x^G \cdot G$.
   (b) Call the random oracle $\mathcal{H}$ on msg and receive m.
   (c) Sample $k_A, \alpha, \beta \leftarrow \mathbb{Z}_q$ and $\rho_0, \rho_1, \rho_2, \rho_3 \leftarrow \mathcal{R}_{\mathsf{pp}}$, and computes:
      i. $C_k = \mathsf{Pedersen.Com}_{G,H}(k_A; \rho_0)$
      ii. $C_\alpha = \mathsf{Pedersen.Com}_{G,H}(\alpha; \rho_1)$
      iii. $C_\beta = \mathsf{Pedersen.Com}_{G,H}(\beta; \rho_2)$
      iv. $C_{kx} = \mathsf{Pedersen.Com}_{X_A,H}(k_A; \rho_3)$
   (d) Run the protocols $\Pi_{\mathsf{zk}}^{L_{\mathsf{Dcom}}[C_k,H]}(G; k_A^{-1}, -k_A^{-1}\rho_0)$, $\Pi_{\mathsf{zk}}^{L_{\mathsf{Dcom}}[C_\alpha,H]}(G; (\alpha^{-1}, -\alpha^{-1}\rho_1)$ and $\Pi_{\mathsf{zk\text{-}uc}}^{L_{\mathsf{Dcom}}[G,H]}(C_\beta; \beta, \rho_2)$ generating proofs π_k and π_α, π_β.
   (e) We denote $\mathsf{pubdata} = \mathsf{sid}, \mathsf{msg}, \mathbb{G}, G, q, H, X, \mathsf{pres}_{X,\mathsf{sid}}$. Call the random oracle on $\mathcal{H}(\mathsf{pubdata}, C_k, C_{kx}, X_A, C_\alpha, C_\beta, \pi_k, \pi_\alpha, \pi_\beta)$, and receives $\mu_k^0, \mu_k^1, \mu_k^G$. It then computes:
      – $\mathsf{ct}_{\gamma\cdot k} = (\mu_k^0 \odot \mathsf{ct}_{\gamma\cdot k_0}) \oplus (\mu_k^1 \odot \mathsf{ct}_{\gamma\cdot k_1}) \oplus (\mu_k^G \odot \mathsf{ct}_\gamma)$
      – $R'_B = (\mu_k^0 R_{B,0}) + (\mu_k^1 \cdot R_{B,1}) + \mu_k^G \cdot G$
   (f) Sample $\eta_0, \eta_1 \leftarrow \mathcal{R}_{pk}$ and computes:
      i. $\mathsf{ct}_{\alpha,\beta} = \mathsf{AHE.Eval}(\mathsf{pk}, (\mathsf{ct}_\gamma, \mathsf{ct}_{\gamma\cdot k}), (0, \beta, \alpha); \eta_0)$
      // In honest protocol encrypts $\gamma \cdot (\alpha \cdot (\mu_k^0 k_{B,0} + \mu_k^1 k_{B,1} + \mu_k^G) + \beta) := \gamma \cdot k_B$
      ii. $R_B = (\alpha \cdot R'_B) + (\beta \cdot G)$
      iii. $R = k_A^{-1} \cdot R_B$ and $r = R_{x-axis}$
      iv. $a_1 = r \cdot k_A \cdot x_A + m \cdot k_A$ and $a_2 = r \cdot k_A$
      v. $\mathsf{ct}_A = \mathsf{AHE.Eval}(\mathsf{pk}, (\mathsf{ct}_\gamma, \mathsf{ct}_{\gamma\cdot\mathsf{key}}), (0, a_1, a_2); \eta_1)$
      // In honest protocol encrypts $\gamma \cdot k_A(m + rx)$
   (g) Generate the following proofs:
      i. $\pi_{kx} \leftarrow \Pi_{\mathsf{zk}}^{L_{\mathsf{DcomEq}}[(G,H),(X_A,H)]}(C_k, C_{kx}; k_A, \rho_0, \rho_3)$
      ii. $\pi_R \leftarrow \Pi_{\mathsf{zk}}^{L_{\mathsf{DComDL}}[G,H,(\mathbb{G},R,q)]}(C_k, R_B; k_A, \rho_0)$
      iii. $\pi_{R_B} \leftarrow \Pi_{\mathsf{zk}}^{L_{\mathsf{VecDComDL}}[(G,H),(\mathbb{G},(R'_B,G),q)]}(C_\alpha, C_\beta), R_B; (\alpha, \beta), \rho_1, \rho_2)$
      iv. $\pi_{\mathsf{ct}_{\alpha,\beta}} \leftarrow \Pi_{\mathsf{zk}}^{L_{\mathsf{DComEval}}[G,H,\mathsf{pk},(\mathsf{ct}_\gamma,\mathsf{ct}_{\gamma\cdot k},(\mathbb{G},G,q))]}(\mathsf{ct}_{\alpha,\beta}, (C_\beta, C_\alpha); (\beta, \alpha), \rho_0, \rho_1, \eta_0)$
      v. $\pi_{\mathsf{ct}_A} \leftarrow \Pi_{\mathsf{zk}}^{L_{\mathsf{DComEval}}[G,H,\mathsf{pk},(\mathsf{ct}_\gamma,\mathsf{ct}_{\gamma\cdot\mathsf{key}}),(\mathbb{G},G,q)]}(\mathsf{ct}_A, (C_1, C_2); (a_1, a_2), \rho_3 r + \rho_0 m, r\rho_0, \eta_1)$
   (h) Denote $T_A = (R, R_B, C_k, C_\alpha, C_\beta, C_{kx}, \mathsf{ct}_A, \mathsf{ct}_{\alpha,\beta}, \pi_k, \pi_\alpha, \pi_\beta, \pi_{kx}, \pi_R, \pi_{R_B}, \pi_{\mathsf{ct}_{\alpha,\beta}}, \pi_{\mathsf{ct}_A})$ and send $(\mathsf{broadcast}, \mathsf{sid}, T_B)$ to $\mathcal{F}_{\mathsf{broadcast}}$.
2. **Each** B_i **does as follows**:
   - **Round 1**:
   (a) Calculate $\mathsf{ct}_{\gamma\cdot\mathsf{key}} = \mu_x^0 \odot \mathsf{ct}_{\gamma_B\cdot\mathsf{key},0} \oplus \mu_x^1 \odot \mathsf{ct}_{\gamma_B\cdot\mathsf{key},1} \oplus \mu_x^G \odot \mathsf{ct}_\gamma$ and $X = X_A + \mu_x^0 X_{0,B} + \mu_x^1 \cdot X_{1,B} + \mu_x^G \cdot G$.
   (b) Call the random oracle $\mathcal{H}(\mathsf{msg})$ and receives m.
   (c) Receive $(\mathsf{broadcast}, \mathsf{sid}, (R, R_B, C_k, C_\alpha, C_\beta, C_{kx}, \mathsf{ct}_A, \mathsf{ct}_{\alpha,\beta}, \pi_k, \pi_\alpha, \pi_\beta, \pi_{kx}, \pi_R, \pi_{R_B}, \pi_{\mathsf{ct}_{\alpha,\beta}}, \pi_{\mathsf{ct}_A}))$ from $\mathcal{F}_{\mathsf{broadcast}}$.
   (d) Call the random oracle $\mathcal{H}(\mathsf{pubdata}, C_k, C_{kx}, X_A, C_\alpha, C_\beta, \pi_k, \pi_\alpha, \pi_\beta)$ and compute:
      i. $\mathsf{ct}_{\gamma\cdot k} = (\mu_k^0 \odot \mathsf{ct}_{\gamma\cdot k_0}) \oplus (\mu_k^1 \odot \mathsf{ct}_{\gamma\cdot k_1}) \oplus (\mu_k^G \odot \mathsf{ct}_\gamma)$
      ii. $R'_B = (\mu_k^0 R_{B,0}) + (\mu_k^1 \cdot R_{B,1}) + \mu_k^G \cdot G$
      iii. $C_1 = (r \odot C_{kx}) \oplus (m \odot C_k)$
      iv. $C_2 = r \odot C_k$
   (e) Verify the proofs. If fail abort and consider A_{pid_A} malicious.
   (f) Send $(\mathsf{decrypt}, \mathsf{pk}, \mathsf{ct}_A)$ and $(\mathsf{decrypt}, \mathsf{pk}, \mathsf{ct}_{\alpha,\beta})$ to $\mathcal{F}_{\mathsf{TAHE}}$.
   - **Broadcast Round**:
   (a) Receive $(\mathsf{decrypted}, \mathsf{pk}, \mathsf{ct}_A, \mathsf{pt}_A)$ and $(\mathsf{decrypted}, \mathsf{pk}, \mathsf{ct}_{\alpha,\beta}, \mathsf{pt}_4)$ from $\mathcal{F}_{\mathsf{TAHE}}$.
   (b) Compute $s' = \mathsf{pt}_4^{-1} \cdot \mathsf{pt}_A \mod q$ and $s = \min\{s', q - s'\}$.
   (c) Send $(\mathsf{global\text{-}broadcast}, \mathsf{sid}, i, (r, s))$ to $\mathcal{F}_{\mathsf{global\text{-}broadcast}}^{\Gamma_B}$
3. **Output**:
   (a) A_{pid_A} receives $(\mathsf{global\text{-}broadcast}, \mathsf{sid}, (r, s))$ from $\mathcal{F}_{\mathsf{global\text{-}broadcast}}^{\Gamma_B}$.
   (b) A_{pid_A} verifies that (r, s) is a valid ECDSA signature, if the verification fails it aborts and considers B malicious.
   (c) Both A_{pid_A} and B outputs (r, s).

Experiment 1: $\mathcal{G}^*$-Existential Unforgeability Experiment $\mathsf{Exp}^{\mathcal{G}^*}_{\mathsf{EU}}(\mathcal{A}, 1^\kappa, n_{\mathsf{s}}, n_{\mathsf{kg}})$ following [10]

The experiment interacts with an adversary $\mathcal{A}$ and is parameterized with 1^κ.

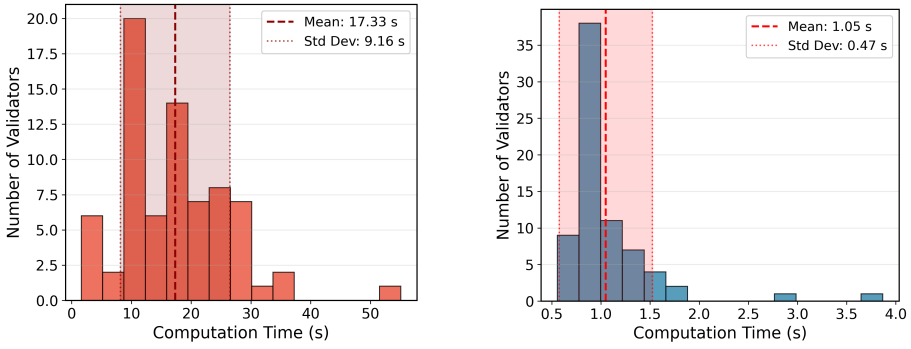
1. Call $\mathcal{G}^*$ with (setup) and hand $(\mathbb{G}, G, q)$ to $\mathcal{A}$.
2. The adversary $\mathcal{A}$ makes at most $n_{\mathsf{kg}} = \mathsf{poly}(\kappa)$ adaptive keygen calls to $\mathcal{G}^*$ and at most $n_{\mathsf{s}} = \mathsf{poly}(\kappa)$ adaptive pres or sign calls to $\mathcal{G}^*$ in the following way:
   - Call $\mathcal{G}^*$ with $(\mathsf{keygen}, \mathsf{sid})$ and hand X to $\mathcal{A}$.
   - Call $\mathcal{G}^*$ with $(\mathsf{pres}, \mathsf{ssid})$ and hand K_0 and K_1 to $\mathcal{A}$.
   - Call $\mathcal{G}^*$ with $(\mathsf{sign}, \mathsf{sid}, \mathsf{ssid}, \mathsf{msg})$ and hand $(\mathsf{sid}, \mathsf{ssid}, \mathsf{msg}, \sigma)$ to $\mathcal{A}$.
3. $\mathcal{A}$ outputs $(\mathsf{sid}, \mathsf{msg}, \sigma)$.

The experiment's output is 1 iff $\mathsf{Verify}(\mathsf{sid}, \mathsf{msg}, \sigma) = 1$ and m was not queried to $\mathcal{G}^*$, otherwise output 0.



(a) Presign. **(b)** Sign.

Fig. 1: 2PC-MPC performance scaling with number of parties from local benchmarks.



(a) Presign. **(b)** Sign.

Fig. 2: Protocol time distribution across validators. Performed over Narwhal consensus with $n = 85$ validators.

FUNCTIONALITY 1 (*Internal Broadcast Functionality: $\mathcal{F}_{\mathsf{broadcast}}$ from [73]*)

The functionality interacts a set of parties $B = \{B_i\}_{i\in[n]}$ and an adversary $\mathcal{A}$.

- Upon receiving $(\mathsf{broadcast}, \mathsf{sid}, i, \mathsf{msg})$ send $(\mathsf{broadcast}, \mathsf{sid}, i, \mathsf{msg})$ to $\mathcal{A}$ and record.
- Upon receiving $(\mathsf{req\text{–}output}, \mathsf{sid}, i, j)$ from $\mathcal{A}$ check if $(\mathsf{broadcast}, \mathsf{sid}, i, \mathsf{msg})$ was recorded, if so send $(\mathsf{broadcast}, \mathsf{sid}, i, \mathsf{msg})$ to B_j.

FUNCTIONALITY 2 (*Global Broadcast Functionality: $\mathcal{F}^{\Gamma_B}_{\mathsf{global\text{-}broadcast}}$*)

The functionality interacts two sets of parties $A = \{A_{\mathsf{pid}_A}\}$, $B = \{B_i\}_{i\in[n]}$ and an adversary $\mathcal{A}$.

- Upon receiving $(\mathsf{global\text{-}broadcast}, \mathsf{sid}, i, \mathsf{msg}_i)$ send $(\mathsf{global\text{-}broadcast}, \mathsf{sid}, i, \mathsf{msg}_i)$ to $\mathcal{A}$ and record.
- Upon receiving $(\mathsf{req\text{–}output}, \mathsf{sid})$ from $\mathcal{A}$, if there exist a message msg and a subset of parties $S_B \in \Gamma_B$, such that for each $i \in S_B$ there is a record $(\mathsf{global\text{-}broadcast}, \mathsf{sid}, i, \mathsf{msg})$ to all parties in A.

FUNCTIONALITY 3 (*Asynchronous Common Subset Functionality: $\mathcal{F}^{\Gamma_B}_{\mathsf{ACS}}$ from [72]*)

The functionality interacts with a set of parties $B = \{B_i\}_{i\in[n]}$ and an adversary $\mathcal{A}$.

- Upon receiving $(\mathsf{validate}, \mathsf{sid}, i, j)$ from B_i record and send $(\mathsf{validate}, \mathsf{sid}, i, j)$ to $\mathcal{A}$.
- Upon receiving $(\mathsf{req\text{–}output}, \mathsf{sid}, i, S_B)$ from $\mathcal{A}$:
  - If $S_B \notin \Gamma_B$, ignore the message.
  - Check that $(\mathsf{validate}, \mathsf{sid}, i, j)$ is recorded for each $B_j \in S_B$. Otherwise, ignore the message.
  - Record $(\mathsf{auth\text{–}subset}, \mathsf{sid}, S_B)$ and reject any future message for sid with a different subset.
  - Send $(\mathsf{auth\text{–}subset}, \mathsf{sid}, S_B)$ to B_i.

PROTOCOL 4 (*Aggregation Protocol: $\Pi^{\Gamma_B, L}_{\mathsf{agg}}$*)

The protocol interacts with a set of parties $B = \{B_i\}_{i\in[n]}$. It is parameterized by an access structure $\Gamma_B \subset 2^B$ along with a Maurer language L. Each party B_i holds a tuple $(\boldsymbol{Y}_i; \boldsymbol{w}_i) \in L$.

- **Each B_i does as follows:**
  1. Run a ZK protocol $\Pi^L_{\mathsf{zk}}(\boldsymbol{Y}_i; \boldsymbol{w}_i)$ generating a proof π_i. Send $(\mathsf{broadcast}, \mathsf{sid}, i, (\mathsf{proof}, \boldsymbol{Y}_i, \pi_i))$ to $\mathcal{F}_{\mathsf{broadcast}}$.
  2. Upon receiving $(\mathsf{broadcast}, \mathsf{sid}, (\mathsf{proof}, \boldsymbol{Y}_j, \pi_j))$ verify the proof π_j, if the proof verifies send $(\mathsf{validate}, \mathsf{sid}, i, j)$ to $\mathcal{F}_{\mathsf{ACS}}$ else consider B_j as malicious.
  3. Receive $(\mathsf{auth\text{–}subset}, \mathsf{sid}, S_B)$ from $\mathcal{F}_{\mathsf{ACS}}$ output $\sum_{j\in S_B} \boldsymbol{Y}_j$.

FUNCTIONALITY 5 (*TAHE: $\mathcal{F}_{\mathsf{TAHE}}$*)

The functionality interacts with a set of parties B and an adversary controlling a subset $U \subseteq B$. It is parameterized with an additive homomorphic encryption AHE along with an access structure Γ.

1. **Key Generation**: Upon receiving $(\mathsf{keygen}, \mathsf{pid}, \mathsf{sid})$ from a subset $S \in \Gamma$ for a fresh sid, run $(\mathsf{pk}; \mathsf{sk}) \leftarrow \mathsf{AHE.Gen}(1^\kappa)$, send pk to the adversary and receive the adversary's response $(\mathsf{continue}, U')$. If $U \cap U' = \emptyset$ then send $(\mathsf{pubkey}, \mathsf{sid}, \mathsf{pk})$ to all parties in B and store it to memory, otherwise send $(\mathsf{pubkey}, \mathsf{sid}, \perp, U \cap U')$ to all parties in B.
2. **Decryption:** Upon receiving a $(\mathsf{decrypt}, \mathsf{pid}, \mathsf{sid}, \mathsf{pk}, \mathsf{ct})$ check if $(\mathsf{pubkey}, \mathsf{sid}, \mathsf{pk})$ is stored in memory. If so let $S_{\mathsf{decrypt},\mathsf{sid},\mathsf{ct}} = S_{\mathsf{decrypt},\mathsf{sid},\mathsf{ct}} \cup \{\mathsf{pid}\}$ (if no request were recorded yet treat $S_{\mathsf{decrypt},\mathsf{sid},\mathsf{ct}}$ as the empty set) and do the following:
   (a) Check if $(\mathsf{decrypted}, \mathsf{sid}, \mathsf{ct}, \mathsf{pt})$ in memory else calculate $\mathsf{pt} \leftarrow \mathsf{AHE.Dec}(\mathsf{pk}, \mathsf{ct}; \mathsf{sk})$ and store $(\mathsf{decrypted}, \mathsf{sid}, \mathsf{ct}, \mathsf{pt})$ to memory.
   (b) Check if $S_{\mathsf{decrypt},\mathsf{sid},\mathsf{ct}} \in \Gamma$, if true, send $(\mathsf{decrypted}, \mathsf{pk}, \mathsf{ct}, \mathsf{pt})$ to the adversary and wait to the adversary's response $(\mathsf{continue}, U'')$ if $U \cap U'' = \emptyset$ send $(\mathsf{decrypted}, \mathsf{sid}, \mathsf{ct}, \mathsf{pt})$ to everyone, otherwise send $(\mathsf{decrypted}, \mathsf{pk}, \perp, U \cap U'')$.

SIMULATION 6 (*Slightly Enhanced ECDSA Lazy Simulation* $\mathcal{S}_{\mathsf{ECDSA}}$)

Key generation is applied immediately at the beginning. A biaskey request can be applied multiple times. Presign requests are global, and every sign request is applied with respect to a specific biaskey output and a specific presign value.

1. Key generation:
   (a) $\pi \leftarrow \{(0, \mathcal{O})\}$
   (b) $\tilde{d}_0, \tilde{d}_1 \leftarrow \mathbb{Z}_q^*$
   (c) Invoke $(\mathsf{map}, 1)$ to obtain G
   (d) Invoke $(\mathsf{map}, \tilde{d}_0)$ and $(\mathsf{map}, \tilde{d}_1)$ to obtain $\tilde{D}_0$ and $\tilde{D}_1$, respectively
   (e) $\mathsf{Keys} \leftarrow \emptyset$
   (f) $k \leftarrow 0; K \leftarrow \emptyset$
   (g) Return $(G, \tilde{D}_0, \tilde{D}_1)$
2. To process a key bias request $(\mathsf{biaskey}, \alpha_x^0, \beta_x^0, \alpha_x^1, \beta_x^1)$:
   (a) Set $d_0 \leftarrow \alpha_x^0 \cdot \tilde{d}_0 + \beta_x^0$ and $d_1 \leftarrow \alpha_x^1 \cdot \tilde{d}_1 + \beta_x^1$
   (b) Invoke (map, d_0) and (map, d_1) to obtain D_0 and D_1, respectively
   (c) $(\mu_x^0, \mu_x^1, \mu_x^G) \leftarrow \mathcal{H}_{\mathsf{key}}(D_0, D_1)$
   (d) $d \leftarrow \mu_x^0 \cdot d_0 + \mu_x^1 \cdot d_1 + \mu_x^G$
   (e) Invoke (map, d) to obtain D
   (f) $\mathsf{Keys} \leftarrow \mathsf{Keys} \cup \{(D, d)\}$
   (g) Return D
3. To process a group oracle query (map, i):
   (a) If $i \notin \mathsf{Domain}(\pi)$:
      i. $\mathcal{P} \leftarrow E^*$;
         while $\mathcal{P} \in \mathsf{Range}(\pi)$ do: $\mathcal{P} \leftarrow E^*$
      ii. Add $(i, \mathcal{P})$ and $(-i, -\mathcal{P})$ to π
   (b) Return $\pi(i)$
4. To process a group oracle query $(\mathsf{add}, \mathcal{P}_0, \mathcal{P}_1)$:
   (a) For $j = 0, 1$: if $\mathcal{P}_j \notin \mathsf{Range}(\pi)$:
      i. $i \leftarrow \mathbb{Z}_q^*$;
         while $i \in \mathsf{Domain}(\pi)$ do: $i \leftarrow \mathbb{Z}_q^*$
      ii. Add $(i, \mathcal{P}_j)$ and $(-i, -\mathcal{P}_j)$ to π
   (b) Invoke $(\mathsf{map}, \pi^{-1}(\mathcal{P}_0) + \pi^{-1}(\mathcal{P}_1))$ and return the result
5. To process a presignature request:
   (a) $k \leftarrow k + 1$
   (b) $\tilde{r}_k^0, \tilde{r}_k^1 \leftarrow \mathbb{Z}_q$
   (c) $\tilde{R}_k^0 \leftarrow (\mathsf{map}, \tilde{r}_k^0)$, $\tilde{R}_k^1 \leftarrow (\mathsf{map}, \tilde{r}_k^1)$
   (d) $K \leftarrow K \cup \{k\}$
   (e) Return $\tilde{R}_k^0, \tilde{R}_k^1$
6. To process a sign request $(\mathsf{sign}, D, k, e_k, m_k, \alpha_k^0, \beta_k^0, \alpha_k^1, \beta_k^1)$:
   (a) If $D \notin \mathsf{Keys}$ or $k \notin K$ then abort
   (b) $K \leftarrow K \setminus \{k\}$
   (c) Using Keys, extract d from D
   (d) Set $d_k = d + e_k$
   (e) $h_k \leftarrow \mathcal{H}_{\mathcal{M}}(m_k)$
   (f) Set $r_k^0 \leftarrow \alpha_k^0 \cdot \tilde{r}_k^0 + \beta_k^0$ and $r_k^1 \leftarrow \alpha_k^1 \cdot \tilde{r}_k^1 + \beta_k^1$
   (g) $R_k^0 \leftarrow (\mathsf{map}, r_k^0)$, $R_k^1 \leftarrow (\mathsf{map}, r_k^1)$
   (h) $(\mu_k^0, \mu_k^1, \mu_k^G) \leftarrow \mathcal{H}_k(D, e_k, R_k^0, R_k^1, m_k)$
   (i) $r_k \leftarrow \mu_k^0 r_k^0 + \mu_k^1 r_k^1 + \mu_k^G$
   (j) $R_k \leftarrow (\mathsf{map}, r_k)$
   (k) $t_k \leftarrow \bar{C}(R_k) \in \mathbb{Z}_q$; if $t_k = 0$ or $h_k + t_k d_k = 0$ then return fail.
   (l) $s_k \leftarrow r_k^{-1}(h_k + t_k d_k)$
   (m) Return (R_k, s_k, t_k)

SIMULATION 7 (*Slightly Enhanced ECDSA Symbolic Simulation*)

1. Key generation:
   (a) Set $\pi \leftarrow \{(0, \mathcal{O})\}$
   (b) Invoke $(\mathsf{map}, 1)$ to obtain G
   (c) Invoke $(\mathsf{map}, \tilde{\mathbf{d}}_0)$ and $(\mathsf{map}, \tilde{\mathbf{d}}_1)$ to obtain $\tilde{D}_0$ and $\tilde{D}_1$, respectively
   (d) $\mathsf{Keys} \leftarrow \emptyset$
   (e) $k \leftarrow 0; K \leftarrow \emptyset$
   (f) Return $(G, \tilde{D}_0, \tilde{D}_1)$
2. To process a key bias request $(\mathsf{biaskey}, \alpha_x^0, \beta_x^0, \alpha_x^1, \beta_x^1)$:
   (a) Set $d_0(\tilde{\mathbf{d}}_0) \leftarrow \alpha_x^0 \cdot \tilde{\mathbf{d}}_0 + \beta_x^0$ and $d_1(\tilde{\mathbf{d}}_1) \leftarrow \alpha_x^1 \cdot \tilde{\mathbf{d}}_1 + \beta_x^1$
   // These are functions of the existing symbolic variables and not new variables
   (b) Invoke $(\mathsf{map}, d_0(\tilde{\mathbf{d}}_0))$ and $(\mathsf{map}, d_1(\tilde{\mathbf{d}}_1))$ to obtain D_0 and D_1, respectively
   (c) $(\mu_x^0, \mu_x^1, \mu_x^G) \leftarrow \mathcal{H}_{\mathsf{key}}(D_0, D_1)$
   (d) $d(\tilde{\mathbf{d}}_0, \tilde{\mathbf{d}}_1) \leftarrow \mu_x^0 \cdot d_0(\tilde{\mathbf{d}}_0) + \mu_x^1 \cdot d_1(\tilde{\mathbf{d}}_1) + \mu_x^G$
   (e) Invoke $(\mathsf{map}, d(\tilde{\mathbf{d}}_0, \tilde{\mathbf{d}}_1))$ to obtain D
   (f) $\mathsf{Keys} \leftarrow \mathsf{Keys} \cup \{(D, d)\}$
   (g) Return D
3. To process a group oracle query (map, i):
   (a) If $i \notin \mathsf{Domain}(\pi)$:
      i. $\mathcal{P} \leftarrow E^*$;
         if $\mathcal{P} \in \mathsf{Range}(\pi)$ then abort
      ii. Add $(i, \mathcal{P})$ and $(-i, -\mathcal{P})$ to π
   (b) Return $\pi(i)$
4. To process a group oracle query $(\mathsf{add}, \mathcal{P}_0, \mathcal{P}_1)$:
   (a) For $j = 0, 1$: if $\mathcal{P}_j \notin \mathsf{Range}(\pi)$:
      i. $i \leftarrow \mathbb{Z}_q^*$;
         if $i \in \mathsf{Domain}(\pi)$ then abort
      ii. Add $(i, \mathcal{P}_j)$ and $(-i, -\mathcal{P}_j)$ to π
   (b) Invoke $(\mathsf{map}, \pi^{-1}(\mathcal{P}_0) + \pi^{-1}(\mathcal{P}_1))$ and return the result
5. To process a presignature request:
   (a) $k \leftarrow k + 1$
   (b) $\tilde{R}_k^0 \leftarrow (\mathsf{map}, \tilde{\mathbf{r}}_k^0)$, $\tilde{R}_k^1 \leftarrow (\mathsf{map}, \tilde{\mathbf{r}}_k^1)$
   (c) $K \leftarrow K \cup \{k\}$
   (d) Return $\tilde{R}_k^0, \tilde{R}_k^1$
6. To process a sign request $(\mathsf{sign}, D, k, e_k, m_k, \alpha_k^0, \beta_k^0, \alpha_k^1, \beta_k^1)$:
   (a) If $D \notin \mathsf{Keys}$ or $k \notin K$ then abort
   (b) $K \leftarrow K \setminus \{k\}$
   (c) Using Keys, extract d from D
   (d) Set $d_k = d + e_k$
   (e) $h_k \leftarrow \mathcal{H}_{\mathcal{M}}(m_k)$
   (f) Set $r_k^0(\tilde{\mathbf{r}}_k^0) \leftarrow \alpha_k^0 \cdot \tilde{\mathbf{r}}_k^0 + \beta_k^0$ and $r_k^1(\tilde{\mathbf{r}}_k^1) \leftarrow \alpha_k^1 \cdot \tilde{\mathbf{r}}_k^1 + \beta_k^1$
   // These are functions of the existing symbolic variables and not new variables
   (g) $R_k^0 \leftarrow (\mathsf{map}, r_k^0(\tilde{\mathbf{r}}_k^0))$, $R_k^1 \leftarrow (\mathsf{map}, r_k^1(\tilde{\mathbf{r}}_k^1))$
   (h) $(\mu_k^0, \mu_k^1, \mu_k^G) \leftarrow \mathcal{H}_k(D, e_k, R_k^0, R_k^1, m_k)$
   (i) $r_k(\tilde{\mathbf{r}}_k^0, \tilde{\mathbf{r}}_k^1) \leftarrow \mu_k^0 \cdot r_k^0(\tilde{\mathbf{r}}_k^0) + \mu_k^1 \cdot r_k^1(\tilde{\mathbf{r}}_k^1) + \mu_k^G$
   (j) $R_k \leftarrow (\mathsf{map}, r_k)$
   (k) $t_k \leftarrow \bar{C}(R_k) \in \mathbb{Z}_q$; if $t_k = 0$ then abort
   (l) $s_k \leftarrow \mathbb{Z}_q^*$
   (m) Substitute $\left(\alpha_k^0\right)^{-1}\left(\left(\mu_k^0\right)^{-1}\left(-\mu_k^1 r_k^1 + s_k^{-1} t_k d_k + s_k^{-1} h_k - \mu_k^G\right) - \beta_k^0\right)$ for the variable $\tilde{\mathbf{r}}_k^0$ throughout $\mathsf{Domain}(\pi)$, to obtain $r_k(\tilde{\mathbf{r}}_k^0, \tilde{\mathbf{r}}_k^1) = s_k^{-1}(h_k + t_k d_k)$, and abort if any two polynomials collapse (i.e., become identical after the substitution)
   (n) Return (R_k, s_k, t_k)

SIMULATION 8 (*Slightly Enhanced ECDSA Modified Symbolic Simulation*)

Key generation is applied immediately at the beginning. A $\mathsf{biaskey}$ request can be applied only once, and all presign and sign requests must be applied afterward.

1. Key generation:
   (a) Set $\pi \leftarrow \{(0, \mathcal{O})\}$
   (b) Invoke $(\mathsf{map}, 1)$ to obtain G
   (c) Invoke $(\mathsf{map}, \tilde{\mathbf{d}}_0)$ and $(\mathsf{map}, \tilde{\mathbf{d}}_1)$ to obtain $\tilde{D}_0$ and $\tilde{D}_1$, respectively
   (d) $\mathsf{Keys} \leftarrow \emptyset$
   (e) $k \leftarrow 0; K \leftarrow \emptyset$
   (f) Return $(G, \tilde{D}_0, \tilde{D}_1)$
2. To process a key bias request $(\mathsf{biaskey}, \mathsf{id}_x)$:
   (a) $(\mu_x^0, \mu_x^1, \mu_x^G) \leftarrow \mathcal{H}_{\mathsf{key}}(\tilde{D}_0, \tilde{D}_1, \mathsf{id}_x)$
   (b) $d(\tilde{\mathbf{d}}_0, \tilde{\mathbf{d}}_1) \leftarrow \mu_x^0 \tilde{\mathbf{d}}_0 + \mu_x^1 \tilde{\mathbf{d}}_1 + \mu_x^G$
   (c) Invoke $(\mathsf{map}, d(\tilde{\mathbf{d}}_0, \tilde{\mathbf{d}}_1))$ to obtain D
   (d) $\mathsf{Keys} \leftarrow \mathsf{Keys} \cup \{(D, d\}$
   (e) Return D
3. To process a group oracle query (map, i):
   (a) If $i \notin \mathsf{Domain}(\pi)$:
      i. $\mathcal{P} \leftarrow E^*$;
         if $\mathcal{P} \in \mathsf{Range}(\pi)$ then abort
      ii. Add $(i, \mathcal{P})$ and $(-i, -\mathcal{P})$ to π
   (b) Return $\pi(i)$
4. To process a group oracle query $(\mathsf{add}, \mathcal{P}_0, \mathcal{P}_1)$:
   (a) For $j = 0, 1$: if $\mathcal{P}_j \notin \mathsf{Range}(\pi)$:
      i. $i \leftarrow \mathbb{Z}_q^*$;
         if $i \in \mathsf{Domain}(\pi)$ then abort
      ii. Add $(i, \mathcal{P}_j)$ and $(-i, -\mathcal{P}_j)$ to π
   (b) Invoke $(\mathsf{map}, \pi^{-1}(\mathcal{P}_0) + \pi^{-1}(\mathcal{P}_1))$ and return the result
5. To process a presignature request:
   (a) $k \leftarrow k + 1$
   (b) $\tilde{R}_k^0 \leftarrow (\mathsf{map}, \tilde{\mathbf{r}}_k^0)$, $\tilde{R}_k^1 \leftarrow (\mathsf{map}, \tilde{\mathbf{r}}_k^1)$
   (c) $K \leftarrow K \cup \{k\}$
   (d) Return $\tilde{R}_k^0, \tilde{R}_k^1$
6. To process a sign request $(\mathsf{sign}, D, k, e_k, m_k, \mathsf{id}_k)$:
   (a) If $D \notin \mathsf{Keys}$ or $k \notin K$ then abort
   (b) $K \leftarrow K \setminus \{k\}$
   (c) Using Keys, extract d from D
   (d) Set $d_k = d + e_k$
   (e) $h_k \leftarrow \mathcal{H}_{\mathcal{M}}(m_k)$
   (f) $(\mu_k^0, \mu_k^1, \mu_k^G) \leftarrow \mathcal{H}_k(D, e_k, \tilde{R}_k^0, \tilde{R}_k^1, m_k, \mathsf{id}_k)$
   (g) $r_k(\tilde{\mathbf{r}}_k^0, \tilde{\mathbf{r}}_k^1) \leftarrow \mu_k^0 \tilde{\mathbf{r}}_k^0 + \mu_k^1 \tilde{\mathbf{r}}_k^1 + \mu_k^G$
   (h) $R_k \leftarrow (\mathsf{map}, r_k)$
   (i) $t_k \leftarrow \bar{C}(R_k) \in \mathbb{Z}_q$; if $t_k = 0$ then abort
   (j) $s_k \leftarrow \mathbb{Z}_q^*$
   (k) Substitute $\left(\mu_k^0\right)^{-1}\left(-\mu_k^1 \tilde{\mathbf{r}}_k^1 + s_k^{-1} t_k d_k + s_k^{-1} h_k - \mu_k^G\right)$ for $\tilde{\mathbf{r}}_k^0$ throughout $\mathsf{Domain}(\pi)$, to obtain $r_k(\tilde{\mathbf{r}}_k^0, \tilde{\mathbf{r}}_k^1) = s_k^{-1}(h_k + t_k d_k)$, and abort if any two polynomials collapse (i.e., become identical after the substitution)
   (l) Return (R_k, s_k, t_k)

PROTOCOL 9 (*ala ElGamal TAHE asynchronous key generation* $\Pi_{\textit{async-keygen}}$*:* $\mathit{TAHE.Gen}(1^\kappa, i, t, n, \mathit{aux})$)

The protocol is parameterized by a computational security parameter κ, a party indicator i, a threshold t, the number of parties n and an auxiliary information aux containing an efficient description of finite abelian groups $\mathbb{H}$ and $\mathbb{F} \leq \mathbb{H}$, a distribution $\mathcal{D}_\kappa$ and public parameters for a commitments scheme $\mathsf{pp} = g$ which is a random element in $\mathbb{G}/\mathbb{H}$. The subgroup $\mathbb{H}$ is of unknown order and the subgroup $\mathbb{F}$ is of known order M and admits an efficient algorithm for computing discrete logs in the subgroup. The protocol assumes that the parties have runs the setup phase of the PVSS scheme and that g is indeed randomized. The parties work as follows:

1. $B_i \in B$ **does as follows**:
   (a) Sample a contribution to the secret key $\alpha_i \leftarrow \mathcal{D}_\kappa$.
   (b) $(\{\mathsf{ct}^{i,j}_{\mathsf{Share}}, C_{i,j}\}_{j\in[n]}, \pi^i_{\mathsf{Share}}) \leftarrow \mathsf{PVSS.Dist}(\mathsf{pp}, \{\mathsf{pk}_j\}_{j\in[n]}, \alpha_i, g)$.
   (c) Send $(\mathsf{broadcast}, \mathsf{sid}, i, (\{\mathsf{ct}^{i,j}_{\mathsf{Share}}, C_{i,j}\}_{j\in[n]}, \pi^i_{\mathsf{Share}}))$ to $\mathcal{F}_{\mathsf{broadcast}}$.
2. $B_i \in B$ **does as follows**:
   (a) Upon receiving $(\{\mathsf{ct}^{j,j'}_{\mathsf{Share}}, C_{j,j'}\}_{j'\in[n]}, \pi^j_{\mathsf{Share}})$ verify the proofs π^j_{Share} and send $(\mathsf{validate}, \mathsf{sid}, i, j)$ to $\mathcal{F}^{\binom{[n]}{t+1}_B}_{\mathsf{ACS}}$. Else consider B_j malicious.
3. **Output**:
   (a) Receive S_B from $\mathcal{F}_{\mathsf{ACS}}$ and set $\mathsf{pk} = \Pi_{j\in S_B} C_{j,0}$.
   (b) Set $[\alpha_j]_i \leftarrow \mathsf{PVSS.DecShare}(\mathsf{pp}, \mathsf{pk}_i, \mathsf{sk}_i, \mathsf{ct}^{j,i}_{\mathsf{Share}})$.
   (c) $[\mathsf{sk}]_i = \sum_{j\in S}[\alpha_i]_j$ and $\mathsf{vk}_j = \Pi_{j'\in S_B} C_{j',j}$.
   (d) record $(\mathsf{pk}, \{\mathsf{vk}_j\}_{j\in[n]}; [\mathsf{sk}]_i)$.

PROTOCOL 10 (*Schnorr Based Presign* $\Pi^{\textit{Schnorr}}_{\textit{pres}}$*:* $\mathsf{SchPresign}(\mathbb{G}, G, q, \Gamma_B, \mathit{sid}, \mathit{pk}$)

The protocol is parameterized with an elliptic curve group description $(\mathbb{G}, G, q)$ along with an access structure Γ_B, a unique session identifier sid, an encryption key to a TAHE scheme pk. The protocol interacts with a centralized party A_{pid_A} and a set of parties $B = \{B_i\}_{i\in[n]}$. The set of parties B can only send messages to parties in A through the functionality $\mathcal{F}_{\mathsf{global\text{-}broadcast}}$. Each party B_i outputs:

- $K_{B,0}, K_{B,1}$ - points on the curve equal to $k_0 \cdot G$ and $k_1 \cdot G$ respectively.
- The distributed parties also output $\mathsf{ct}_{k_0}, \mathsf{ct}_{k_1}$ - encryptions of k_0, k_1 respectively.

1. **Each** B_i **does as follows**:
   - **Round 1**
   (a) B_i samples $k^i_0, k^i_1 \leftarrow \mathbb{Z}_q$ and $\eta^i_0, \eta^i_1 \leftarrow \mathcal{R}_{pk}$ and computes:
      i. $K^i_{B,0} = k^i_0 \cdot G$
      ii. $K^i_{B,1} = k^i_1 \cdot G$
      iii. $\mathsf{ct}^i_{k_0} = \mathsf{AHE.Enc}(\mathsf{pk}, k^i_0; \eta^i_0)$
      iv. $\mathsf{ct}^i_{k_1} = \mathsf{AHE.Enc}(\mathsf{pk}, k^i_1; \eta^i_0)$
   (b) Run $\Pi^{L_{\mathsf{EncDL}}[\mathsf{pk},(\mathbb{G},G,q)]}_{\mathsf{agg}}$ on inputs $(\mathsf{ct}^i_{k_1}, K^i_{B,0}; k^i_0, \eta^i_0)$ and $(\mathsf{ct}^i_{k_1}, K^i_{B,1}; k^i_1, \eta^i_1)$ receiving $(\mathsf{ct}_{k_0}, K_{B,0}), (\mathsf{ct}_{k_1}, K_{B,1})$.
   - **Broadcast Round**:
   (a) Send $(\mathsf{global\text{-}broadcast}, \mathsf{sid}, i, (\mathsf{pid}_A, K_{B,0}, K_{B,1}))$ to $\mathcal{F}_{\mathsf{global\text{-}broadcast}}$.
2. **Output.**
   - A_{pid_A} receives $(\mathsf{global\text{-}broadcast}, \mathsf{sid}, , (\mathsf{pid}_A, K_{B,0}, K_{B,1}))$ from $\mathcal{F}_{\mathsf{global\text{-}broadcast}}$.
   - A_{pid_A} Verifies that $K_{B,0}, K_{B,1} \in \mathbb{G} \setminus \{0\}$ elset it aborts and considers B malicious.
   - Both parties output $K_{B,0}, K_{B,1}$.
   - Parties in B also record $\mathsf{ct}_{k_0}, \mathsf{ct}_{k_1}$.

PROTOCOL 11 (*Schnorr Based Sign* $\Pi_{sign}^{Schnorr}$: $\mathsf{SchSign}(\mathbb{G}, G, q, sid, pk, X, ct_{key}, pres_X)$)

The protocol is parameterized with an elliptic curve group description $(\mathbb{G}, G, q)$ along with an access structure Γ_B, a unique session id sid, an encryption key to a TAHE scheme pk, a verification key X, an encryption of the share of the distributed party of the signing key $\mathsf{ct_{key}}$ and the output of a unique presign $\mathsf{pres}_X = (K_{B,0}, K_{B,1})$. The protocol interacts with a centralized party A_{pid_A} and a set of parties $B = \{B_i\}_{i\in[n]}$. The set of parties B can only send messages to parties in A through the functionality $\mathcal{F}_{\mathsf{global\text{-}broadcast}}$. The parties output a valid Schnorr signature (z, e) to a verification key X.

1. A_{pid_A} **does as follows**:
   (a) Samples $k_A \leftarrow \mathbb{Z}_q$ and computes $K_A = k_A \cdot G$.
   (b) Call the random oracle $\mathcal{H}(\mathsf{sid}, X, \mathsf{pres}_{X,\mathsf{sid}}, K_A, \mathsf{msg})$, and receive μ_k.
   (c) Compute $K = (K_{B,0}) + (\mu_k \cdot K_{B,1}) + K_A$.
   (d) Call the random oracle $\mathcal{H}(K, X, \mathsf{msg})$, and receive e.
   (e) Compute $z_A = k_A + e \cdot x_A$.
   (f) Send $(\mathsf{broadcast}, \mathsf{sid}, (z_A, K_A))$ to $\mathcal{F}_{\mathsf{broadcast}}$
2. **Each** B_i **does as follows**:
   - **Round 1**:
   (a) Receive $(\mathsf{broadcast}, \mathsf{sid}, (z_A, K_A))$ from $\mathcal{F}_{\mathsf{broadcast}}$ and verify that $K_A \in \mathbb{G}$ else consider A_{pid_A} malicious and abort.
   (b) Check that $z_A G = K_A + e \cdot X_A$. If not, it consider A_{pid_A} malicious and abort.
   (c) Call the random oracle $\mathcal{H}(\mathsf{sid}, X, \mathsf{pres}_{X,\mathsf{sid}}, K_A, \mathsf{msg})$, and receive μ_k.
   (d) B_i computes:
      i. $\mathsf{ct}_k = \mathsf{ct}_{k_0} \oplus \mu_k \odot \mathsf{ct}_{k_1}$
      ii. Compute $K = (K_{B,0}) + (\mu_k \cdot K_{B,1}) + K_A$.
   (e) Call the random oracle $\mathcal{H}(K, X, \mathsf{msg})$, and receive e.
   (f) Compute $\mathsf{ct}_B = \mathsf{ct}_k \oplus (e \cdot \mathsf{ct_{key}}) \oplus z_A$.
   (g) Send $(\mathsf{decrypt}, \mathsf{pk}, \mathsf{ct}_B)$ to $\mathcal{F}_{\mathsf{TAHE}}$.
   - **Broadcast Round**
   (a) receive $(\mathsf{decrypted}, \mathsf{pk}, \mathsf{ct}_B, \mathsf{pt}_B)$ from $\mathcal{F}_{\mathsf{TAHE}}$.
   (b) set $\sigma = (\mathsf{pt}_B, e)$.
   (c) Send $(\mathsf{global\text{-}broadcast}, \mathsf{sid}, i, \sigma)$ to $\mathcal{F}_{\mathsf{global\text{-}broadcast}}$.
3. **Output**:
   (a) Party A_{pid_A} receives $(\mathsf{global\text{-}broadcast}, \mathsf{sid}, \sigma)$ from $\mathcal{F}_{\mathsf{global\text{-}broadcast}}$.
   (b) Party A_{pid_A} verifies that σ is a valid signature if falis it aborts and considers B malicious.
   (c) Both parties output σ.

FUNCTIONALITY 12 (*Slightly Enhanced Schnorr Signing Oracle:* $\mathcal{G}^*_{\mathsf{SE\text{-}Sch}}$)

1. On input $(\mathsf{keygen}, \mathsf{sid})$, set $x \leftarrow \mathbb{Z}_q$ and $X = x \cdot G$, record $(\mathsf{sid}, X; x)$ and send X.
2. On input $(\mathsf{pres}, \mathsf{ssid})$, sample $k_0, k_1 \leftarrow \mathbb{Z}_q$, compute $K_0 = k_0 \cdot G$ and $K_1 = k_1 \cdot G$. Record $(\mathsf{ssid}, K_0, K_1; k_0, k_1)$ and send $(\mathsf{ssid}, K_0, K_1)$.
3. On input $(\mathsf{sign}, \mathsf{sid}, \mathsf{ssid}, \mathsf{msg}; \alpha_{\mathsf{key}}, \beta_{\mathsf{key}}, \alpha_{\mathsf{pres},0}, \beta_{\mathsf{pres},0}, \alpha_{\mathsf{pres},1}, \beta_{\mathsf{pres},1})$ do:
   (a) Retrieve $(\mathsf{sid}, X; x)$ and $(\mathsf{ssid}, K_0, K_1; k_0, k_1)$. If no such ssid or sid exist, ignore.
   (b) Compute $x' \leftarrow \alpha_{\mathsf{key}} \cdot x + \beta_{\mathsf{key}}$, $k'_0 \leftarrow \alpha_{\mathsf{pres},0} \cdot k_0 + \beta_{\mathsf{pres},0}$, and $k'_1 \leftarrow \alpha_{\mathsf{pres},1} \cdot k_1 + \beta_{\mathsf{pres},1}$.
   (c) Set $X' = x' \cdot G$, $K'_0 = k'_0 \cdot G$ and $K'_1 = k'_1 \cdot G$.
   (d) Set $\mu = \mathcal{H}(X', K'_0, K'_1, \mathsf{msg}; \alpha_{\mathsf{key}}, \beta_{\mathsf{key}}, \alpha_{\mathsf{pres},0}, \beta_{\mathsf{pres},0}, \alpha_{\mathsf{pres},1}, \beta_{\mathsf{pres},1})$ and set $e = \mathcal{H}(X', K'_0 + \mu \cdot K'_1, \mathsf{msg})$.
   (e) compute $z = k'_0 + \mu k'_1 + ex' \mod q$.
   (f) Erase $(\mathsf{ssid}, K_0, K_1; k_0, k_1)$ from memory and return $(\mathsf{sid}, \mathsf{ssid}, \mathsf{msg}; \alpha_{\mathsf{key}}, \beta_{\mathsf{key}}, \alpha_{\mathsf{pres},0}, \beta_{\mathsf{pres},0}, \alpha_{\mathsf{pres},1}, \beta_{\mathsf{pres},1}, K_0 + \mu \cdot K_1, z)$.

FUNCTIONALITY 13 (*Slightly Enhanced Schnorr Signing Oracle without biases :* $\mathcal{G}^*_{\mathsf{Non\text{-}Biased\text{-}SE\text{-}Sch}}$)

Parameters. A Random Oracle function $\mathcal{H} : \{0,1\}^* \to \mathbb{Z}_q$. Slightly Enhanced Schnorr oracle works as follows:

Operation.

1. On input $(\mathsf{keygen}, \mathsf{sid}, (\mathbb{G}, G, q))$, sample $x \leftarrow \mathbb{Z}_q$ and return $X = x \cdot G$. Record $(\mathsf{sid}, X; x)$.
2. On input $(\mathsf{pres}, \mathsf{ssid})$, sample $k_0, k_1 \leftarrow \mathbb{Z}_q$ and return $K_0 = k_0 \cdot G$ and $K_1 = k_1 \cdot G$. Record $(\mathsf{ssid}, K_0, K_1; k_0, k_1)$ and standby.
3. On input $(\mathsf{sign}, \mathsf{sid}, \mathsf{ssid}, \mathsf{msg})$ do:
   (a) Retrieve $(\mathsf{ssid}, K_0, K_1; k_0, k_1)$ and $(\mathsf{sid}, X; x)$ from memory. If no such ssid or sid exist, ignore.
   (b) Set $\mu = \mathcal{H}(X, K_0, K_1, \mathsf{msg})$ and set $e = \mathcal{H}(X, K_0 + \mu \cdot K_1, \mathsf{msg})$.
   (c) Set $z = k_0 + \mu k_1 + ex \mod q$.
   (d) Erase $(\mathsf{ssid}, K_0, K_1; k_0, k_1)$ from memory and return $(\mathsf{sid}, \mathsf{ssid}, \mathsf{msg}, K_0 + \mu \cdot K_1, z)$.

ALGORITHM 14 (*Forking Algorithm:* $\mathsf{Fork}_{\mathcal{A}}(inp, h_1, h'_1, \ldots, h_{n_r}, h'_{n_r})$)

Parameters. A forking algorithm is parameterized by $\mathcal{A}$ and its random space $\mathcal{R}$. The algorithm works as follows:

Operation.

1. Sample $\rho \leftarrow \mathcal{R}$.
2. Call $\mathcal{A}(inp, h_1, \ldots h_{n_r}; \rho)$. If $\mathcal{A}$ outputs $\perp$, then return $\perp$, otherwise retrieve (ℓ, out).
3. Call $\mathcal{A}(inp, h_1, \ldots, h_{\ell-1}, h'_\ell, \ldots, h'_{n_r}; \rho)$. If $\mathcal{A}$ outputs $\perp$, then return $\perp$, otherwise retrieve (ℓ', out').
4. If $\ell \neq \ell'$ or $h_\ell = h'_{\ell'}$ then return $\perp$.
5. Return (ℓ, out, out').

EXPERIMENT 15 (*Algebraic One More Discrete Log* $\mathsf{Exp}^{n_{DL}}_{\mathsf{AOMDL}}(\mathcal{A}, 1^\kappa, (\mathbb{G}, G, q))$ *[11]*)

The experiment interacts with an adversary $\mathcal{A}$ and is parameterized with 1^κ, $(\mathbb{G}, G, q)$, and n_{DL}.

1. The challenger:
   (a) Samples $k_1, \ldots, k_{n_{\mathsf{DL}}} \leftarrow \mathbb{Z}_q$, computes $K_\ell = k_\ell \cdot G$ for every $\ell \leq n_{\mathsf{DL}}$,
   (b) Sends $K_1, \ldots, K_{n_{\mathsf{DL}}}$ to the adversary.
2. $\mathcal{A}$ repeats the following process $n_{\mathsf{DL}} - 1$ times:
   (a) Chooses $\alpha_0, \ldots, \alpha_{n_{\mathsf{DL}}}$ from $\mathbb{Z}_q$ and sends them to the challenger.
   (b) Receives from the challenger $\alpha_1 \cdot k_1 + \cdots + \alpha_{n_{\mathsf{DL}}} \cdot k_{n_{\mathsf{DL}}}$.
3. $\mathcal{A}$ sends to the challenger $k'_1, \ldots, k'_{n_{\mathsf{DL}}}$.

   The experiment's output is 1 if and only if $k_\ell = k'_\ell$ for every $\ell \leq n_{\mathsf{DL}}$, otherwise output 0.

SIMULATION 16 (*Hybrid Simulation of Experiment 1 with predetermined keys and presigns* $\mathcal{S}^{Exp_{AOMDL}}_{Exp_{EU}(\mathcal{A}^{n_s,n_{kg},n_r}_{SE-Sch-Forger})}(1^\kappa,(\mathbb{G},G,q),h_1,\ldots,h_{n_r+2n_s};\rho)$)

This hybrid simulates the challenger in Experiment 1 facing $\mathcal{A}^{n_s,n_{kg},n_r}_{SE-Sch-Forger}$ and simulates $\mathcal{A}$ in $\mathsf{Exp}^{2n_s+n_{kg}}_{AOMDL}(\mathcal{A},1^\kappa,(\mathbb{G},G,q))$ facing the challenger from Experiment 15. The hybrid uses predetermined values for: $\mathcal{A}^{n_s,n_{kg},n_r}_{SE-Sch-Forger}$'s random tape and $\mathcal{H}$ queries output. The hybrid outputs a forged signature based on the predetermined random oracle queries output. It does not try to succeed in Experiment 15.

1. Receive $K_1,\ldots K_{2n_s+n_{kg}}$ from the challenger.
2. Replace the notation $X_\ell = K_{2n_s+\ell}$ for $\ell \leq n_{kg}$.
3. Run $\mathcal{A}^{n_s,n_{kg},n_r}_{SE-Sch-Forger}$ with randomness ρ as follows:
   - Upon receiving $(\mathsf{keygen},\mathsf{sid}$ from $\mathcal{A}^{n_s,n_{kg},n_r}_{SE-Sch-Forger}$:
     (a) Let X_ℓ be the next one not in memory. If all of X_ℓ are used, abort.
     (b) Keep $(\mathsf{keygen},\mathsf{sid},X_\ell)$ in memory and send it to $\mathcal{A}^{n_s,n_{kg},n_r}_{SE-Sch-Forger}$.
   - Upon receiving $(\mathsf{pres},\mathsf{ssid})$ from $\mathcal{A}^{n_s,n_{kg},n_r}_{SE-Sch-Forger}$:
     (a) Let K_ℓ be the next one not in memory. If all of K_ℓ are in memory, abort.
     (b) Keep $(\mathsf{pres},\mathsf{ssid},K_\ell,K_{\ell+1})$ in memory and send it to $\mathcal{A}^{n_s,n_{kg},n_r}_{SE-Sch-Forger}$.
   - Upon Receiving calling the random oracle $\mathcal{H}(C)$ do the following process denoted by $RO(C)$:
     - If there is h such that $(\mathsf{RandomOracle},C,h)$ is in memory, send h to $\mathcal{A}^{n_s,n_{kg},n_r}_{SE-Sch-Forger}$ and break.
     - Otherwise, let h_ℓ be the minimal ℓ such that there is no h_ℓ in memory.
     - Keep $(\mathsf{RandomOracle},C,h)$ in memory and send h to $\mathcal{A}^{n_s,n_{kg},n_r}_{SE-Sch-Forger}$.
   - Upon receiving $(\mathsf{sign},\mathsf{sid},\mathsf{ssid},\mathsf{msg})$ do:
     - If there is z such that $(\mathsf{sign},\mathsf{sid},\mathsf{ssid},\mathsf{msg},z)$ is in memory, send $(\mathsf{sign},\mathsf{sid},\mathsf{ssid},\mathsf{msg},z)$ to $\mathcal{A}^{n_s,n_{kg},n_r}_{SE-Sch-Forger}$.
     - If either sid or ssid are not in the memory, send $\perp$ to the challenger and abort.
     - Otherwise:
       (a) Denote by K_ℓ and $K_{\ell+1}$ the pair such that $(\mathsf{pres},\mathsf{ssid},K_\ell,K_{\ell+1})$ is in memory. If there is none, abort.
       (b) Denote by $X_{\ell'}$ the value such that $(\mathsf{keygen},\mathsf{sid},X_{\ell'})$ is in memory.
       (c) Set $\mu = RO(X,K_\ell,K_{\ell+1},\mathsf{msg})$ and $e = RO(X_{\ell'},K_\ell+\mu\cdot K_{\ell+1},\mathsf{msg})$, where $RO(C)$ is the process described earlier, and do not send its output to $\mathcal{A}^{n_s,n_{kg},n_r}_{SE-Sch-Forger}$.
       (d) Set $\alpha_\ell = 1$, $\alpha_{\ell+1} = \mu$, $\alpha_{2n_s+\ell'} = e$, and otherwise $\alpha_s = 0$
       (e) Send $\alpha_1,\ldots,\alpha_{2n_s+n_{kg}}$ to the challenger.
       (f) Receive z from the challenger.
       (g) keep $(\mathsf{sign},\mathsf{sid},\mathsf{ssid},\mathsf{msg},z)$ in memory and send it to $\mathcal{A}^{n_s,n_{kg},n_r}_{SE-Sch-Forger}$.
   - Upon receiving $\perp$ from $\mathcal{A}^{n_s,n_{kg},n_r}_{SE-Sch-Forger}$, abort.
   - Upon receiving $(\mathsf{sid},\mathsf{msg},K,z)$ from $\mathcal{A}^{n_s,n_{kg},n_r}_{SE-Sch-Forger}$ do:
     (a) If there are ssid and z such that $(\mathsf{sign},\mathsf{sid},K,\mathsf{msg},z)$ is in memory abort.
     (b) Retrieve X such that $(\mathsf{keygen},\mathsf{sid},X)$ is in memory.
     (c) Retrieve ℓ such that $(\mathsf{RandomOracle},(X,K,\mathsf{msg}),h_\ell)$ is in memory. If there is none, abort.
     (d) output $(\ell,(\mathsf{msg},X,K,z))$.

ALGORITHM 17 (*AOMDL adversary* $\mathcal{A}^{\mathcal{S}_{Exp_{EU}}}_{\mathsf{AOMDL-Sch}}(1^\kappa, (\mathbb{G}, G, q))$)

This algorithm is a PPT adversary $\mathcal{A}$ for $\mathsf{Exp}^{2n_s+n_{kg}}_{\mathsf{AOMDL}}$ (Experiment 15) that reduces AOMDL to forging sigantures. It uses $\mathcal{S}_{\mathsf{Exp}_{\mathsf{EU}}}$ to simulate the challenger in Experiment 1 and retrieve a forged signature from $\mathcal{A}^{n_s,n_{kg},n_r}_{\mathsf{SE-Sch-Forger}}$. It uses a fork to forge two signatures with the same nonce, in order to break AOMDL.

1. Sample $h_1, h'_1, \ldots, h_{n_r+2n_s}, h'_{n_r+2n_s}$ from H.
2. Receive $K_1, \ldots, K_{2n_s+n_{kg}}$ from the challenger.
3. Sample $\rho \leftarrow \mathcal{R}$. *// the random space of* $\mathcal{A}^{n_s,n_{kg},n_r}_{\mathsf{SE-Sch-Forger}}$
4. Simulate $\mathcal{S}_{\mathsf{Exp}_{\mathsf{EU}}}(1^\kappa, (\mathbb{G}, G, q), h_1, \ldots, h_{n_r+2n_s}; \rho)$ as follows:
   - Send $K_1, \ldots, K_{2n_s+n_{kg}}$ to $\mathcal{S}_{\mathsf{Exp}_{\mathsf{EU}}}$.
   - Upon receiving $\alpha_1, \ldots, \alpha_{2n_s+n_{kg}}$ from $\mathcal{S}_{\mathsf{Exp}_{\mathsf{EU}}}$:
     (a) If there is z such that $((\alpha_1, \ldots, \alpha_{2n_s+n_{kg}}), z)$ is in memory, send z to $\mathcal{S}_{\mathsf{Exp}_{\mathsf{EU}}}$ and break.
     (b) Otherwise, send $\alpha_1, \ldots, \alpha_{2n_s+n_{kg}}$ to the challenger.
     (c) Receive z from the challenger.
     (d) Keep $((\alpha_1, \ldots, \alpha_{2n_s+n_{kg}}), z)$ in memory.
     (e) Send z to $\mathcal{S}_{\mathsf{Exp}_{\mathsf{EU}}}$.
   - Upon receiving $\perp$ from $\mathcal{S}_{\mathsf{Exp}_{\mathsf{EU}}}$, send $\perp$ to the challenger and abort.
5. Receive $(\ell_0, (\mathsf{msg}, X, K, z_0))$ from $\mathcal{S}_{\mathsf{Exp}_{\mathsf{EU}}}$ and keep it in memory.
6. Simulate $\mathcal{S}_{\mathsf{Exp}_{\mathsf{EU}}}(1^\kappa, (\mathbb{G}, G, q), h_1, \ldots, h_{\ell_0-1}, h'_{\ell_0}, \ldots, h'_{n_r+2n_s}; \rho)$:
   - Send $K_1, \ldots, K_{2n_s+n_{kg}}$ to $\mathcal{S}_{\mathsf{Exp}_{\mathsf{EU}}}$.
   - Upon receiving $\alpha_1, \ldots, \alpha_{2n_s+n_{kg}}$ from $\mathcal{S}_{\mathsf{Exp}_{\mathsf{EU}}}$:
     (a) If there is z such that $((\alpha_1, \ldots, \alpha_{2n_s+n_{kg}}), z)$ is in memory, send z to $\mathcal{S}_{\mathsf{Exp}_{\mathsf{EU}}}$ and break.
     (b) Otherwise, send $\alpha_1, \ldots, \alpha_{2n_s+n_{kg}}$ to the challenger.
     (c) Receive z from the challenger.
     (d) Keep $((\alpha_1, \ldots, \alpha_{2n_s+n_{kg}}), z)$ in memory.
     (e) Send z to $\mathcal{S}_{\mathsf{Exp}_{\mathsf{EU}}}$.
   - Upon receiving $\perp$ from $\mathcal{S}_{\mathsf{Exp}_{\mathsf{EU}}}$, send $\perp$ to the challenger and abort.
7. Receive $(\ell_1, (\mathsf{msg}', X', K', z_1))$ from $\mathcal{S}_{\mathsf{Exp}_{\mathsf{EU}}}$.
8. If $\ell_0 \neq \ell_1$, or $h_{\ell_0} = h'_{\ell_0}$, or $\mathsf{msg} \neq \mathsf{msg}'$, or $K \neq K'$, or $X \neq X'$ then send $\perp$ to the challenger.
9. Retrieve $\ell > 2n_s$ such that $K_\ell = X$. If there is none, abort.
10. Check that $\log_G K_\ell = \frac{z_0 - z_1}{h_{\ell_0} - h'_{\ell_0}}$. If not, then the signatures are incorrect. Return $\perp$ and abort.
11. For $\ell' \leq 2n_s + n_{kg}$, set $\alpha_{\ell'}$ to 1 if $\ell' = 2n_s + \ell$, and to 0 otherwise.
12. Keep $((\alpha_1, \ldots, \alpha_{2n_s+n_{kg}}), \frac{z_0 - z_1}{h_{\ell_0} - h'_{\ell_0}})$ in memory.
13. Let $\{v_{\ell'}\}_{\ell'=1}^{2n_s+n_{kg}} \subset H^{2n_s+n_{kg}}$ be a set of vectors such that:
    - For every $((\alpha_1, \ldots, \alpha_{2n_s+n_{kg}}), z)$ in memory, there is ℓ' such that $v_{\ell'} = (\alpha_1, \ldots, \alpha_{2n_s+n_{kg}})$.
    - The set $\{v_{\ell'}\}_{\ell'=1}^{2n_s+n_{kg}}$ is linearly independent.
14. If there is no such set, send $\perp$ to the challenger and abort.
15. Send the challenger $v_{\ell'}$ if and only if there is no z such that $(v_{\ell'}, z)$ is in memory.
16. Retrieve $\log_G K_1, \ldots, \log_G K_{2n_s+n_{kg}}$ using the linear combinations $\{v_{\ell'}\}_{\ell'=1}^{2n_s+n_{kg}}$ as the result of each linear combination of them is already known.
17. Send $\log_G K_1, \ldots, \log_G K_{2n_s+n_{kg}}$ to the challenger.

SIMULATION 18 (*Simulation of Slightly Enhanced Schnorr Signing Oracle (with multiplication):* $\mathcal{S}_{\mathsf{SE\text{-}Sch}}$)

$\mathcal{S}_{\mathsf{SE\text{-}Sch}}$ simulates the biased signing oracle $\mathcal{G}^*_{\mathsf{SE\text{-}Sch}}$ given oracle access to the unbiased signing oracle $\mathcal{G}^*_{\mathsf{Non\text{-}Biased\text{-}SE\text{-}Sch}}$ (and its random oracle $\mathcal{H}$). **Parameters.** A cyclic group $(\mathbb{G}, G, q)$, a Random Oracle function $\mathcal{H} : \{0,1\}^* \to \mathbb{Z}_q$. The bias oracle uses the variables denoted by $'$, while the unbiased variables are denoted without it. Slightly Enhanced Schnorr oracle works as follows:

Operation.

1. On input $(\mathsf{keygen}, \mathsf{sid})$, call $\mathcal{G}^*_{\mathsf{Non\text{-}Biased\text{-}SE\text{-}Sch}}$ with $(\mathsf{keygen}, \mathsf{sid})$, receive (sid, X), record and send it.
2. On input $(\mathsf{pres}, \mathsf{ssid})$, call $\mathcal{G}^*_{\mathsf{Non\text{-}Biased\text{-}SE\text{-}Sch}}$ with $(\mathsf{pres}, \mathsf{ssid})$. Receive $(\mathsf{ssid}, K_0, K_1)$, record and send it.
3. On input $(\mathsf{query}, X', K'_0, K'_1, \mathsf{msg}'; \alpha_{\mathsf{key}}, \beta_{\mathsf{key}}, \alpha_{\mathsf{pres},0}, \beta_{\mathsf{pres},0}, \alpha_{\mathsf{pres},1}, \beta_{\mathsf{pres},1})$, do as follows:
   (a) Set $X \leftarrow \alpha_{\mathsf{key}}^{-1} \cdot (X' - \beta_{\mathsf{key}} \cdot G)$, $K_0 \leftarrow \alpha_{\mathsf{pres},0}^{-1} \cdot (K'_0 - \beta_{\mathsf{pres},0} \cdot G)$, and $K_1 \leftarrow \alpha_{\mathsf{pres},1}^{-1} \cdot (K'_1 - \beta_{\mathsf{pres},1} \cdot G)$.
   (b) If there is no record of (sid, X) or of $(\mathsf{ssid}, K_0, K_1)$, query $\mathcal{H}(X, K_0, K_1, \mathsf{msg}; \alpha_{\mathsf{key}}, \beta_{\mathsf{key}}, \alpha_{\mathsf{pres},0}, \beta_{\mathsf{pres},0}, \alpha_{\mathsf{pres},1}, \beta_{\mathsf{pres},1})$, record and return its output.
   (c) Else, if there is no record of $(\mathsf{map}, X, X', K_0, K_1, K'_0, K'_1, K_0 + \mu \cdot K_1, K'_0 + \mu' \cdot K'_1, \mathsf{msg}, \mathsf{msg}'; \alpha_{\mathsf{key}}, \beta_{\mathsf{key}}, \alpha_{\mathsf{pres},0}, \beta_{\mathsf{pres},0}, \alpha_{\mathsf{pres},1}, \beta_{\mathsf{pres},1}, \mu, \mu')$:
      i. Sample $\mathsf{msg} \leftarrow \{0,1\}^\kappa$
      ii. query $\mathcal{H}(X, K_0, K_1, \mathsf{msg})$, receive μ and compute $\mu' = \frac{\mu \cdot \alpha_{\mathsf{pres},0}}{\alpha_{\mathsf{pres},1}}$.
      iii. Record $(\mathsf{map}, X, X', K_0, K_1, K'_0, K'_1, K_0 + \mu \cdot K_1, K'_0 + \mu' \cdot K'_1, \mathsf{msg}, \mathsf{msg}'; \alpha_{\mathsf{key}}, \beta_{\mathsf{key}}, \alpha_{\mathsf{pres},0}, \beta_{\mathsf{pres},0}, \alpha_{\mathsf{pres},1}, \beta_{\mathsf{pres},1}, \mu, \mu')$
   (d) Return μ'.
4. On input $(\mathsf{query}, X', K', \mathsf{msg}')$, do as follows:
   (a) If there is no record of $(\mathsf{map}, X, X', K_0, K_1, K'_0, K'_1, K_0 + \mu \cdot K_1, K'_0 + \mu' \cdot K'_1, \mathsf{msg}, \mathsf{msg}'; \alpha_{\mathsf{key}}, \beta_{\mathsf{key}}, \alpha_{\mathsf{pres},0}, \beta_{\mathsf{pres},0}, \alpha_{\mathsf{pres},1}, \beta_{\mathsf{pres},1}, \mu, \mu')$, query $\mathcal{H}(X', K', \mathsf{msg}')$ and return its output.
   (b) Else, retrieve X, K, msg, α_{key}, and $\alpha_{\mathsf{pres},0}$, and query $e = \mathcal{H}(X, K, \mathsf{msg})$.
   (c) Compute, record and return $e' = \frac{e \cdot \alpha_{\mathsf{pres},0}}{\alpha_{\mathsf{key}}}$.
5. On input $(\mathsf{sign}, \mathsf{sid}, \mathsf{ssid}, \mathsf{msg}'; \alpha_{\mathsf{key}}, \beta_{\mathsf{key}}, \alpha_{\mathsf{pres},0}, \beta_{\mathsf{pres},0}, \alpha_{\mathsf{pres},1}, \beta_{\mathsf{pres},1})$ do:
   (a) Retrieve (sid, X) and $(\mathsf{ssid}, \mathsf{pres}, K_0, K_1)$ from memory.
   (b) Compute $X' \leftarrow \alpha_{\mathsf{key}} \cdot X + \beta_{\mathsf{key}} \cdot G$, $K'_0 \leftarrow \alpha_{\mathsf{pres},0} \cdot K_0 + \beta_{\mathsf{pres},0} \cdot G$, and $K'_1 \leftarrow \alpha_{\mathsf{pres},1} \cdot K_1 + \beta_{\mathsf{pres},1} \cdot G$.
   (c) Set μ' and e' by self-calling $(\mathsf{query}, X', K'_0, K'_1, \mathsf{msg}'; \alpha_{\mathsf{key}}, \beta_{\mathsf{key}}, \alpha_{\mathsf{pres},0}, \beta_{\mathsf{pres},0}, \alpha_{\mathsf{pres},1}, \beta_{\mathsf{pres},1})$ and $(\mathsf{query}, X', K', \mathsf{msg}')$, respectively.
   (d) Retrieve $(\mathsf{map}, X, X', K_0, K_1, K'_0, K'_1, K_0 + \mu \cdot K_1, K'_0 + \mu' \cdot K'_1, \mathsf{msg}, \mathsf{msg}'; \alpha_{\mathsf{key}}, \beta_{\mathsf{key}}, \alpha_{\mathsf{pres},0}, \beta_{\mathsf{pres},0}, \alpha_{\mathsf{pres},1}, \beta_{\mathsf{pres},1}, \mu, \mu')$ from memory.
   (e) Call $\mathcal{G}^*_{\mathsf{Non\text{-}Biased\text{-}SE\text{-}Sch}}$ with $(\mathsf{sign}, \mathsf{sid}, \mathsf{ssid}, \mathsf{msg})$ and receive z.
   (f) Compute $z_A = \beta_{\mathsf{pres},0} + \mu' \cdot \beta_{\mathsf{pres},1} + e' \beta_{\mathsf{key}}$.
   (g) Record and return $z' = \alpha_{\mathsf{pres},0} \cdot z + z_A$.