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

Offir Friedman[1], Avichai Marmor[1], Dolev Mutzari[1], Yehonatan C. Scaly[1], and Yuval Spiizer[1]

dWallet Labs, `research@dwalletlabs.com`

Abstract. Threshold signatures have become a critical tool in cryptocurrency systems, offering enhanced security by distributing the signing process among multiple signers. In this work, we distribute this process between a client and a *permissionless decentralized blockchain*, and present novel protocols for ECDSA and EdDSA/Schnorr signatures in this setting. Typical threshold access architectures used by trusted custodians suffer from the *honeypot problem*, wherein the more assets the custodian holds, the greater the incentive of compromising it.

Implementing threshold signatures over permissionless blockchains poses a few challenges. First, existing networks typically work over an *asynchronous reliable broadcast communication channel.* Accordingly, our protocol is implemented over such a channel. As a result, it also benefits from *identifiable abort*, *public verifiability*, and *guaranteed output delivery*, and the client benefits from *censorship resistance* of blockchain systems. Second, upon signing each block, the participating quorum may *dynamically change* and is *post-determined.* Therefore, we design a *fluid* protocol, that supports a post-determined dynamic quorum in each communication round, thereby complying with existing broadcast channel implementations. Third, in permissionless networks, parties may join, leave, and change their stake. Therefore, we offer protocols for *network reconfiguration*, with complexity independent of the number of clients in the system, and our protocol efficiently supports a *weighted threshold access structure* for the network. Specifically, the complexity of distributed key generation and presign only depends on the number of parties and not on the overall weight, and the amortized cost of sign only depends on the individual weight.

Furthermore, our protocol introduces key improvements, including the removal of zero-knowledge proofs towards the client, and presigns with a *non-interactive client.* For Schnorr, the presigns are *client-independent*, and can be collected by the blockchain in a common pool, available for all clients in the system. These optimizations reduce communication overhead and improve the system's ability to handle traffic spikes during high-demand periods.

Our protocol is UC-secure, and is therefore natively designed for multiple clients to use the system in parallel. Notably, we propose a novel assumption, *Slightly-Enhanced ECDSA Unforgeability*, offering concrete security for 256-bit elliptic curves for threshold ECDSA with support for parallel execution of presigns.

In addition to securing cryptocurrency wallets, we demonstrate how our protocol enables various cross-chain applications, such as *decentralized bridges*, *future transactions*, and *wallet transfer*. Our system is designed for *interoperability* across multiple blockchains, enhancing security, scalability, and flexibility for decentralized finance (DeFi) ecosystems.

1 Introduction

In recent years, there has been growing interest in thresholdizing digital signatures [EKR20, BD23, BLT+24], distributing the signing process among multiple parties. This is primarily because digital signatures serve as an authentication mechanism in Bitcoin [Nak08], Ethereum [B+13], 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 [JMV01] in past years [GGN16, Lin17, BGG17, LN18, GG19, DKLS19, CGG+20, DOK+20, DJN+20, GKSS20, CCL+20, ANO+22, BMP22, XAX+22, DKLS23, ZYP23, WMYC23, CCL+23, CDKS24], due to its widespread use in cryptocurrency.

Threshold signature based solutions have been widely deployed in the cryptocurrency industry and focus on mainly two use cases: distributed custody (e.g. Coinbase Wallet [Lin23], Qredo [KZe19b], Zengo [KZe19a], Fireblocks [Fir23], Copper [Cop21], Web3 Auth [Lab24] and Lit Protocol [CD24b]), and decentralized bridges (e.g. THORChain [Tho20], Keep Network [LP18], Near Network [IA21] and Internet Computer [T+22]). Distributed custody uses threshold signature protocols to mitigate self-custodial risks such as loss and theft by distributing the private signing key of the client, which verifies ownership of the funds, across multiple signers. In some applications, the key is shared between a client and a *centralised* or *trusted custodian* [Lin23, KZe19a, Fir23, Cop21, Lab24].[1] Other applications hand the key to a *trusted authorized custodian*, and the client uses an identification mechanism to invoke signing on transactions by the custodian [KZe19b, CD24b]. Ultimately, the signing key may be shared between the client and a *distributed permissionless custodian* in a way that requires the active consent of both in order to produce a signature. By permissionless, we mean that anyone can join the custodian network, similar to permissionless blockchains, and unlike trusted custodians or permissioned blockchains. Currently, only trusted custodians are deployed. If the client is involved, the custodians are typically centralized due to inherent limitations of prior approaches.

Decentralized bridges consist of a network of nodes that distributedly manage multiple signing keys for different wallets, each holding a pool of funds on different blockchains to form cross-chain liquidity pools. This enables a single entity (the network) to own, manage, and transfer funds (and information) across different blockchains. When a client owns wallets on two different blockchains and wishes to transfer funds between them, it can send funds to one end of the

[1] In some configurations in e.g. [Cop21], the client may use a client-trusted third party and sign with a 2-out-of-3 access structure.

bridge's liquidity pool and request a withdrawal of an equivalent amount to the client's wallet on the other end. The bridge network then creates and distributedly signs a corresponding transaction that the client can broadcast to the other blockchain.

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 [T+22]). 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 [Eth24]). Recently, [FMM+24] has taken a pivotal step toward increasing the number of nodes for producing ECDSA signatures, 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 [FMM+24], 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 (i.e., maintaining a high number of sessions concurrently) and a message complexity quadratic in the number of parties, as each party computes and sends a message to every other party. Instead, [FMM+24] opts to rely on a *broadcast channel*, and reduce 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. decentralized permissionless blockchains also provide *censorship resistance* for the client, as opposed to trusted custodians.

Nevertheless, some key aspects remain unaddressed. Primarily, their choice of working over an *ideal* broadcast channel, avoids a lot of difficulties that arises once one wishes to implementing their protocol over existing broadcast channel implementations. First, existing networks typically work over an *asynchronous 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 and did not send a message during some round. Second, upon signing each block, the participating quorum may *dynamically change* and is *post-determined*. Therefore, parties who participate in one communication round may not be available for the next one. Section 2 comprehensively explores these gaps and situates our approach within the broader context of threshold signature research. Building on [FMM+24], we address these critical challenges and provide an affirmative answer to the question below:

Can we design an asynchronous, permissionless, decentralized network capable of concurrently servicing numerous clients requesting digital signatures on their transactions?

Thresholdizing EdDSA. Recently, Bitcoin added support for EdDSA [BDL$^+$12], a derivative of Schnorr signatures [Sch90]. Other cryptocurrencies such as Stellar [KAS19], Monero [Cry13] and Solana [Yak18] have adopted EdDSA exclusively.[2] This has spurred interest in threshold Schnorr protocols [KG21, CKM21, Lin22, BTZ22, RRJ$^+$22, CGRS23, CDSG$^+$24]. Nonetheless, the vast majority supports either ECDSA or EdDSA. To support interoperability between the bulk of deployed blockchains, in this work we consider both threshold ECDSA and Schnorr signatures.

While threshold protocols for Schnorr are already deployed in production with up to 150 parties (Chainflip [Har23], Stacks [Ali20], Threshold [Wil22]), these protocols rely on traditional threshold access structure. Whenever the network consists of a large number of nodes, the client is not involved in producing the signatures and, in particular, has to rely on the network to safely store the private signing key. This stands in contrast to the *Zero Trust* principle [Sta20], which has been foundational in blockchain ecosystems since the inception of Bitcoin. Under this principle, no one is trusted, not even miners or validators, and every client calculates and verifies the state themselves.

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. This has resulted in billions of dollars in losses, hampering the mass adoption of Web3 and the transfer of real economic value to blockchain applications. For instance, Wormhole [Gra22], a platform that mints and burns WETH on Solana, has already been exploited by an attacker in 02/02/2022 to mint 120,000 WETH [Wor22] - worth at the time over 320 million USD, which were used to withdraw real ETH on Ethereum, leading to one of the largest hacks in Web3's history (see [LMDG23] for a recent review of cross-chain bridge hacks).

In order to resolve those issues and comply with the Zero Trust architecture, [FMM$^+$24] proposed the *2PC-MPC* framework, which we follow and 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 signature raises many practical difficulties. First, the access structure is inherently hierarchical. Prior solutions that used a thresh-

[2] Cardano [Hos17] has recently added support for ECDSA [BA23].

old 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 even not be aware of their identities. Exposing such communication channels may pose additional cyber-security threat for parties in the network, and would also significantly increase the communication overhead of the client.

With the above issues in mind, [FMM$^+$24] added on top of 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.[3] Hence the name: 2PC-MPC.

Another important aspect of a decentralized system is the number of clients it can concurrently service. This metric was put explicitly on the table in [FMM$^+$24], and we make further improvements in this metric. Prior works on threshold signatures are typically focused on a single key, and do not consider the system as a whole. Therefore, each node in the decentralized network must hold a private share per each wallet of each client in the system. Instead, in [FMM$^+$24], 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) [CDN01]. 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.

In this work, we also identify that this design potentially allows for parties to join and leave the network efficiently. In the naive approach, the network had to re-distribute the network share of the signing key of each client. In contrast, in the above design, whenever the network changes its configuration, only this secret decryption key must be reshared. We propose such a protocol, and point out that this allows for the decentralized network to be permissionless.

1.1 Our Contribution

Our primary contribution is the design and implementation of a 2PC-MPC protocol for an asynchronous, permissionless, decentralized, weighted network, servicing numerous clients concurrently to securely sign messages, using ECDSA and EdDSA/Schnorr. 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. Section 2 also provides extensive

[3] 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.

comparison with prior works, and [FMM+24] in particular, further highlighting the challenges addressed in this work and the advantage of our approach.

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 [FMM+24]. 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 sign 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 preprocess can reduce computation and latency during high-demand periods. When folded back to the standard threshold access structure (removing the client), DKG is one-round, presign is one-round for EdDSA/Schnorr and two rounds for ECDSA, and sign is a single round. This makes it on-par with state of the art protocols for Schnorr [CDSG+24] and ECDSA [CDKS24].

In addition, we alleviate the need for *zero-knowledge* (ZK) proofs towards the client, that were used in [FMM+24] to prove honest behaviour. In order to admit the 2PC-MPC framework, [FMM+24] offered to aggregate the proofs from each party in the network. However, when the plaintext space of the TAHE is not aligned with the ECDSA/Schnorr group order q, these proofs in [FMM+24] use range proofs (specifically, Bulletproofs [BBB+18]). While these proofs can be aggregated in the sense that the aggregated proof size is logarithmic with the number of parties n, verification time by the client remains linear with n. By removing the need for zk-proofs, we resolve this issue. We believe our technique can be applied to their protocol as well.

Furthermore, for Schnorr signatures, presign is client-independent. Namely, a presign is generated by the network alone, and can be leveraged by *any* client during the online signing phase. By computing presigns during periods of *low load* or *idle time*, the system can efficiently handle sudden *traffic spikes*, ensuring minimal disruption when demand surges. In addition, this saves the cost associated with generation of client-dedicated presigns that may never be used.

Moreover, for Schnorr signatures, we also support *soft key derivation* as specified in the BIP32 standard [Wui13] for hierarchical deterministic wallets. This standard allows the derivation of a child public key from a master public key, enabling child keys to inherit permissions from their parent, while optionally being restricted to more limited actions. We refer to Section 3 for more details.

In terms of security, our protocol for ECDSA circumvents the *Enhanced-ECDSA unforgeability* assumption (proposed in [CGG+20]), even when allowing presigns. Enhanced ECDSA was proven secure in the EC-GGM model (see Section 3) in [GS22], but the concrete parameters of the reduction do not suffice when working over 256-bit elliptic curves commonly used nowadays (e.g., secp256k1 [Bal24] 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 not only corrupt part of the network participants, but also has control over the network communication channel. This introduces challenges in the UC-simulation, as we cover in Section 3.1.

Additionally, compared to [FMM$^+$24], our protocol UC-realizes a more robust and natural ideal functionality (see Functionality 4.1). In [FMM$^+$24], a weaker functionality that was first introduced by [CGG$^+$20] is realized, which apart from ensuring unforgeability, allows the adversary to control the distribution of the public key and the signature. We believe [FMM$^+$24] may also be adjusted to realize our functionality, that generates the public key and signature according to an ideal signing oracle. Notably, we are only able to realize "slightly-enhanced" signing oracles, which allow the adversary to inject a linear biases. Nevertheless, we prove this signing oracles to be secure.

In Section 1.2 we delve deeper into the concrete design of our proposed system, followed by Section 1.3 where we explore the range of applications that can be built upon it.

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 [LRP20].[4] In turn, during each reconfiguration validators may also withdraw part of their stake, or increase their stake, which could change their voting power accordingly.[5] In this context, when referring to blockchains we also consider *DAG-based cryptocurrencies* [LeM17], *DAG-based consensus platforms* and *blockchain-DAG hybrids* [BCD$^+$23b] (see [PMIH18, GZT$^+$19, WYY$^+$22, WYCX23] for recent surveys). DAG-based designs allow validators to propose blocks in parallel. This improves the system *latency*, that is, the time to create a wallet or sign a transaction, as well as the *throughput*, that is, the number of transactions (or wallets) produced per unit time. As observed by [FMM$^+$24], this can be utilized to increase the throughput of the online-signing phase. Essentially, the bottleneck in this phase is the aggregation

[4] If the blockchain is based on *Proof of Work (PoW)*, each *miner* has a voting power proportional to its computational power.

[5] In PoW blockchains, the estimated computational power of a miner can be adjusted based on its performance during the last epoch, and its voting power is adjusted accordingly.

of decryption shares. Therefore, each validator can aggregate decryption shares of a different subset of signing request, amortizing this cost.

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 channel* [FLP85, BT83, BT85] used for producing blocks.[6] The *block proposer* validator is responsible for aggregating the last MPC round, compute the message to send to the client, and include it in its proposed block. Oftentimes, such communication channel is either asynchronous (as in Narwhal-Tusk [DKKSS22]) or *partially synchronous* (as in Narwhal-Bullshark [SGSKK22]).

Policies. Upon creation, wallets will specify policies, either by using *smart contracts* (see [MPJ18, WYW$^+$18] and [TLL$^+$21, HZL$^+$21] for recent overviews and surveys) implemented on-chain, or using light-clients [CBC22] 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 1.3, we illustrate several key use-cases enabled by the flexibility, security, and scalability of the above design.

1.3 Applications

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 this date.[7]

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

[6] In our case, *reliable broadcast* [Bra87] is sufficient, as we do not require agreement on the ordering of MPC messages. Consensus is often implemented on top of a reliable broadcast channel, e.g., Narwhal-Tusk [DKKSS22], where Tusk implements asynchronous consensus over Narwhal which implements a reliable broadcast channel.

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

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* [Dev23], wherein the transaction is executed only after a specified period of time/blocks.

Bridging Blockchains. We identify *bridges* as an appealing private 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 [SG20, ACK$^+$20] 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) [BAJ23]. 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*.[8] 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.

Decentralized Autonomous Organization (DAO). By supporting BIP32 [Wui13], DAOs can manage a common wallet associated with a master public key, and delegate permissions on it. For instance, the master key may have

[8] 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.

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.

1.4 Paper Organization

In Section 2 we cover the limitations of prior works including [FMM$^+$24]. Section 3 contains a technical overview of our protocol. It pinpoints the main technical challenges addressed in this work and outlines our proposed solutions. Section 4 contains preliminaries and definitions. Our 2PC-MPC ECDSA and Schnorr protocols are presented in Section 5 and Section 6 respectively. The protocols for reconfiguration are presented in Section 7. In Section 8 we analyze the security of our protocol. Implementation details and evaluation are provided in Section 9.

2 Comparison to [FMM$^+$24] & Related Work

This section provides a detailed comparison of our work with the foundational 2PC-MPC framework proposed by [FMM$^+$24]. While we build on their design, their instantiation faces significant limitations in real-world applicability. Below, we outline those key limitations, discuss relevant related works, and highlight the practical challenges our approach addresses.

First and foremost, the work in [FMM$^+$24] is over an *ideal* broadcast channel. In reality, consensus protocols implement a broadcast channel over P2P channels, e.g. via gossiping [AYSS09]. This distinction is crucial, since implementing an ideal broadcast channel is theoretically impossible [FLP85]. This abstraction overlooks three major challenges when resorting to off-the-shelf broadcast channel implementations, all of which we address in this work.

First, existing implementations realize more relaxed functionalities, e.g. *consistent broadcast*, *reliable broadcast*, *consensus* or *atomic broadcast* (see [CGR11] for an overview). Each of those provides different security guarantees, potentially compromising on latency or bandwidth. In this work, we rely on consistent broadcast for security, and require reliable broadcast to ensure correctness properties such as guaranteed output delivery.

Second, they assume a *synchronous communication channel*, meaning that all parties have access to a global clock, and in particular, can decide and agree if a party does not broadcast a message. This also occurs in [RRJ$^+$22], a recent adaptation of [DKLS23] that achieves identifiable abort by adding a synchronous broadcast channel. Synchronous communication has proven to be a rather stringent requirement [WGD22], and blockchains typically work over the more relaxed *partially-synchronous* communication channel [DLS88], or preferably, over an *asynchronous* communication channel. Indeed, synchronizing on a global clock can significantly affect network latency. In this work, the protocols are defined over an asynchronous channel, which supports existing implementations of broadcast channels (e.g., Narwhal-Tusk [DKKSS22], Narwhal-Bullshark [SGSKK22], Mysticeti [BCD+23a]) and provides a significantly lower latency. In asynchronous communication, parties compute and broadcast their messages for the next round as soon as a threshold of parties from the previous round broadcasts their messages, without having to coordinate and agree on which parties failed to participate. Worth mentioning here is ROAST [RRJ+22], an adaptation of FROST1 [KG21] that works over an asynchronous communication channel. However, their solution requires running multiple sessions of FROST in parallel, up to the number of honest parties in the network, making it asymptotically more expensive. Whenever a party comes back online, it must complete all the sessions that it missed to ensure delivery. This makes it difficult for an offline party to catch-up, and also consumes a significant amount of storage for managing those parallel sessions with respect to each sign request.

Third, their work requires the same subset of parties to participate in all rounds of each protocol (DKG, presign and sign). This is not on-par with existing implementations of broadcast channels [DKKSS22, SGSKK22, BCD+23a], which only ensure that an arbitrary threshold of parties is online during each communication round. The notion of *fluid MPC* [CGG+21], which allows the subset of participating in an MPC protocol to dynamically change between rounds, was recently proposed. However, current research is mostly theoretic. Our protocol achieves fluidity, and is therefore aligned with the premises of existing broadcast channel implementations. In particular, the parties in the protocol are essentially *stateless*. Notably, [RRJ+22] also resolves the above issue, but indirectly. Their transformation ensures that there will be one session in which a fixed threshold of parties participate in all rounds. In our approach, one session can have a different subset of parties in each round, leading to better performance. One reason that flexibility might be challenging is that we cannot have commitment rounds, as parties may not be available in the next round to open them. Our solution to this borrows ideas from FROST1 [KG21].

Permissionless Networks. Although the 2PC-MPC design is suitable for letting the set of parties in the network to change, their work neither mentions this nor offers a concrete protocol for network reconfiguration. Indeed, with the 2PC-MPC framework, since all network signing key shares are encrypted under the same public key of a TAHE, resharing the secret decryption key of the TAHE with the new committee is sufficient for network reconfiguration. Nevertheless, we find it to be quite challenging, since when working over a TAHE based on a hidden-order group (e.g. Tiresias [FMM+23] based on Paillier [Pai99], or [BDO23, BCD+24] based on the Class-Groups encryption scheme by [CLT18] in the CL framework [CL15]), the secret decryption key is shared over the integers [Sha79b]. Unfortunately, current re-configuration protocols [YXXM23, LRU22] assume the sharing is over a finite field and cannot be directly applied. The main concern is that when working over the integers, the bit-length of the secret shares will increase in every reconfiguration, making it impractical. We propose two re-configuration protocols for secrets shared over the integers, each of which may be preferable depending on the use-case.

In this context, many works (e.g., [CMP20, BMP22, CCL+23]) have considered proactive security [CGHN97], where the adversary can adaptively choose which subset of parties to corrupt (and recover). This is often achieved by re-sharing the private signing key, which is somewhat similar to reconfiguration. Nevertheless, this is different from allowing the threshold and the set of parties themselves to change as well. A more closely related work is Dynamic-FROST [CDSG+24], the first Schnorr threshold signature scheme that accommodates changes in both the committee and the threshold value without relying on a trusted third party. However, such works directly reshare the private signing key. As a result, when considering a system with numerous wallets, whenever the committee changes, such protocol must be executed with respect to each wallet separately, and so does not scale with the number of clients in the system.

Weighted Access Structure. Oftentimes, the access structure of the decentralized network is *weighted*. This is the case in Proof-of-Stake (PoS) blockchains, wherein each validator has a voting power proportional to its stake, and this is also the case in Proof-of-Work (PoW) blockchains, wherein each miner has a chance to propose a new block proportional to its computational power. To align with the financial incentives and security premises of the underlying blockchain, the secret key of the TAHE should also be shared according to the same weighted access structure. In this context, [DPTX24] proposed a scheme for weighted Verifiable Random Functions in which signing and verification time, as well as the signature size, are independent of the total weight of the parties. However, their construction relies heavily on the signature being based on pairings. A more related work is WSTS [Yan23] which reduces the bandwidth during the presign and sign rounds when parties control multiple keys. In our protocol, bandwidth and computation during the DKG is independent of the total number of parties as well, as a consequence of the 2PC-MPC design. Indeed, only the DKG of the underlying TAHE depends on the overall voting power (and reconfiguration). Additionally, the amortized computational cost during the online signing phase is proportional to the average voting power rather than the total voting power. We also remark that if the set of parties participating in the online signing phase is known in advance, the computational cost of each party is proportional to its individual voting power, without amortization. This assumption was taken in [Yan23], but was not fully exploited since the verification of the signature shares by the aggregator takes work proportional to the total voting power. Instead, we utilize the idea in [FMM+24], suggesting that it is sufficient to verify the aggregated signature in order to verify the correctness of the signature shares, reducing this additional cost.

Finally, beyond practical implementation challenges, we also address theoretical concerns in securing presign protocols. Specifically, proving security when allowing presign is known to be challenging. Up until this work, threshold ECDSA protocols that enabled presigns (e.g., [CGG+20, DOK+20, DKLS23, CCL+23, FMM+24]) have assumed the Enhanced ECDSA Unforgeability assumption [CGG+20], which essentially states that forging ECDSA signatures is hard even if the adversary is allowed to query the nonce part of multiple signatures before deciding on the messages it wants to sign. The hardness of Enhanced ECDSA Unforgeability was established in the *Elliptic Curve Generic Group Model* (EC-GGM) proposed in [GS22]. However, the complexity of their reduction is cubic, which is insufficient for concrete security for 256-bit elliptic curves. In this work, we propose the Slightly-Enhanced ECDSA Unforgeability assumption, and follow similar techniques to those employed in [GS22] to analyze its security. In our protocol, the presign is slightly different compared to prior works, and consists of a pair of nonces rather than one. At the online signing phase, the nonces are randomly combined based on the message to be signed, the client's input, the presign, and the public key. In this case, we achieve a quadratic complexity for the reduction, making the Slightly Enhanced assumption compatible with 256-bit elliptic curves. For our Schnorr protocol, we use a similar technique to the one used in FROST3 [CGRS23], and base our security on the *Algebraic One-More Discrete Log* (AOMDL) [NRS21] assumption. However, we take a more modular approach by first reducing the signing oracle itself to breaking AOMDL, and then simulating the protocol with calls to the signing oracle. This modular approach can be of independent interest since it can potentially simplify the simulation of other threshold Schnorr protocols.

3 Technical Overview

In this section, we discuss the technical challenges addressed in this work. In Section 3.1 we discuss our methods for implementing a typical threshold signature protocol in an asynchronous network model, and address the challenges in UC-simulation. In Section 3.2 we discuss our methods to support presigns for both ECDSA and Schnorr. In Section 3.3 we elaborate on our methods for allowing the network to dynamically change. Finally, in Section 3.4 we show how to comply with the 2PC-MPC framework.

3.1 Asynchronous protocol

When working over an asynchronous communication channel, one cannot simply rely on commitments, as a party that sends a commitment in one round may be unavailable in the next round to open it. Without commitments, values can be chosen adaptively by a rushing adversary. Another point of influence given to the adversary in this model is adaptively choosing the subset of participating parties based on their messages. The issue is further complicated by the need to conform to the UC model. To illustrate the above issues, consider a simplified DKG protocol for Schnorr:

1. Each party i samples $x_i \leftarrow \mathbb{Z}_q$ and broadcasts $X_i = x_i \cdot G$ along with a ZKP π_{DL}.

2. The parties agree on a subset S of $t+1$ parties who sent valid messages and set $X = \sum_{i \in S} X_i$.

As a first attempt, consider an ideal functionality that samples the public key X at random. Unfortunately, simulation here is impossible, even against a semi-honest adversary. Regardless of what the simulator sends on behalf of the honest parties, it cannot force the protocol to land on a specific X. Therefore, our idea is to explicitly give the adversary more freedom in the ideal world. As a second attempt, consider a modified functionality that allows the adversary to inject a bias β, resulting with i.e. $X + \beta \cdot G$. In this case, a standalone simulation is manageable:

1. The simulator picks an honest party i^* at random, sends $X + x_{i^*} \cdot G$ on its behalf, and simulates the corresponding ZK proof. On behalf of the rest, it acts honestly.
2. Upon receiving the messages from the malicious parties, it rewinds the adversary and extracts the discrete log of X_i for each malicious party.
3. When the adversary chooses the subset S, if $i^* \notin S$ it rewinds to the beginning of the simulation.
4. Finally, it sets $\beta = \sum_{i \in S} x_i$ and sends it to the ideal functionality.

However, moving to the UC model introduces additional complexities. Specifically, we cannot use rewinding in order to extract the secrets of the adversary or to guess i^* correctly. A naive solution for the former is to use UC-extractable ZK proofs, e.g. by applying Fischlin's transform [Fis05]. However, this costs around an order of magnitude in performance [CL24]. Instead, we utilize TAHE encryption for witness extraction. Using a UC-secure TAHE and simulating the TAHE's ideal functionality, the simulator holds the secret decryption key and extracts encrypted secrets. Computationally sound ZK proofs are still necessary to bind curve points to ciphertexts but they do not need to be UC-extractable.

As for the latter issue, we relax the functionality even further, and allow the adversary to apply an *affine transformation* $\alpha \cdot X + \beta \cdot G$ with $\alpha \neq 0$. In this case, the simulator can send $X + x_i \cdot G$ on behalf of each honest party. Then, when the adversary picks the subset S, the simulator sets α to be the number of honest parties in S, in order to land on the same public key as in the real execution.

As we refine the ideal functionality, in order to complete the security analysis we must analyze unforgeability with respect to the refined signing oracle. The unforgeability for Schnorr signatures is established in Theorem 8.2. On the other hand, it turns out that this functionality makes the adversary too powerful against ECDSA signatures, and allows an efficient forgery attack:

1. Receive X from the functionality.
2. Choose a message msg^* to be forged and calculate $m^* = \mathcal{H}(\mathsf{msg}^*)$.
3. Set $R^* = X$ and set r^* to be R^*'s X coordinate.
4. Send (R^*, r^*, r^*) (i.e., set $s^* = r^*$) as the signature to msg^* with respect to the public key $X - \frac{m^*}{r^*} G$.

One can easily verify that this forgery is successful. In order to prevent such attacks, we modify the functionality as follows: The functionality first samples two public keys X_0, X_1 and sends them to the adversary. The adversary can choose coefficients α_i, β_i (with $\beta_i \neq 0$) and set $X_i' = \alpha_i X_i + \beta_i G$. Finally, the functionality samples random coefficients $\mu_x^0, \mu_x^1, \mu_x^G$, and the final key is $\mu_x^0 X_0 + \mu_x^1 X_1 + \mu_x^G G$. (In fact, these coefficients are derived deterministically and not sampled, see Functionality 8.1 for details.) ECDSA is proven secure with respect to this functionality in Theorem D.3.

3.2 Adding presign support

Another challenge that we face is support of presigns. In this case, if we allow the adversary to inject arbitrary bias to the public key, then apparently the Enhanced ECDSA signing oracle is insecure [GS22]. Moreover, the Enhanced Schnorr signing oracle is insecure even without the above relaxation [BLL+22]. This is not surprising, as the adversary seems to have a lot of power. In order to blunt this control we use a technique inspired by FROST1 [KG21]. Essentially, the nonce part of the signature is derived from the presign output as well as the message to be signed. This idea gives rise to a modified signing oracle which we call the *Slightly Enhanced Signing Oracle*. As the name suggests, we aim to give evidence that this oracle provides better security than *Enhanced Signing Oracle* introduced in [CGG+20] which allows the adversary to see the public nonce before choosing the message. Apparently, this signing oracle is also secure when allowing the adversary to apply affine transformations on the public key and the presigns.

Next, we describe the Slightly Enhanced signing oracle for ECDSA and Schnorr signatures in more detail. Upon request of a presign, two public nonces R_0, R_1 are generated, rather than one. Then, when querying for a signature on some message msg, the public nonce R used for the signature is determined by a linear combination of the two nonces $R = \mu_0 R_0 + \mu_1 R_1 + \mu_G G$. The coefficients are derived from the public key X, the presign R_0, R_1, and the message msg. Intuitively, this suggests that the adversary cannot predict the nonce before it decides on a message. In addition, for the reasons mentioned in Section 3.1, we let the adversary pick invertible affine transformations for the public key (and in ECDSA, for its two parts), and for both nonces in the presign. Below, we briefly cover the security analysis of these slightly enhanced signing oracles.

Slightly Enhanced ECDSA Signing Oracle. We prove that Functionality 8.1 is secure in the EC-GGM model introduced by [GS22]. 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 preform computations over $\mathbb{G}$. We provide two main simulations of Functionality 8.1 in this model. In the first simulation, called lazy simulation, the mapping π is sampled "on the fly", and whenever a new value $x \in \mathbb{Z}_q$ is queried, a value $\pi(x) \in \mathbb{G}$ is randomly sampled, recorded and returned. In the second simulation, called symbolic simulation, the secret ECDSA signing key is never sampled, and instead a symbolic variable is used. We i) prove that our protocol UC realizes the lazy simulation in the EC-GGM model, ii) present a polynomial reduction between the lazy simulation and the symbolic simulation, and iii) show that the symbolic simulation is secure. 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.

Slightly Enhanced Schnorr Signing Oracle. We reduce an adversary that can forge a signature given access to the signing oracle in Functionality 8.2 to breaking *Algebraic One More Discrete Log* (AOMDL) [NRS21] game. For simplicity, we first reduce such forgery to forging signatures when adding biases to the public key and presign is not allowed (Functionality D.4). Then, we use a generalized forking lemma [BN06, Lemma 1] to reduce the latter forgery into winning the AOMDL security game.

Remarkably, Functionalities 8.2 and 8.1 provide greater versatility than what we have described above. Specifically, the adversary is allowed to specify a *tweak* $(\alpha_{\mathsf{key}}, \beta_{\mathsf{key}})$ that will be applied on the public key in the signing request itself. In other words, the adversary is allowed to add a different bias to the same public key, upon each signing request. This choice enables support for *soft key derivation* as specified in BIP32 standard [Wui13]. Crucially, the reason that despite all of these degrees of freedom the signing oracle stays secure, is that the functionality's coefficients are derived from a random oracle on all of the above values: the biased public key and presign, the tweaks, and the message. If any of them is removed, an attack becomes feasible.

In addition, Functionality 8.2 allows the adversary to ask for an arbitrary number of keys and presigns, and then adaptively choose the key-presign-message binding in parallel for all signing requests to follow. This ensures security for the use-case of a common pool of presigns for multiple clients.

3.3 Reconfiguration

Before addressing the client, one final issue remains regarding the network: enabling dynamic reconfiguration through a dedicated protocol. As previously mentioned, it suffices to reshare the secret decryption key of the TAHE scheme, denoted as sk. While asynchronous reconfiguration protocols exist in the literature [YXXM23, LRU22], to the best of our knowledge, they all assume secret sharing schemes over finite fields. Unfortunately, this assumption does not hold for two prevalent TAHE schemes based on Paillier and class groups, where the secret key sk resides in a hidden-order group. In these cases, sk is shared over the integers, introducing notable challenges. Specifically, integer-based secret shares are substantially larger than the original secret, with their bit-length growing super-linearly with the number of parties. Moreover, the naive approach of "resharing the shares" is impractical, since the size of the secret shares may grow indefinitely with each reconfiguration.

To address this, we propose two asynchronous reconfiguration protocols:

Protocol 1: Dynamic Weight and Distribution Adjustment - This protocol supports dynamic changes in both the total weight and its distribution among parties. It assumes the secret key is self-encrypted, $\mathsf{ct}_{\mathsf{sk}} := \mathsf{Enc}_{\mathsf{pk}}(\mathsf{sk})$, relying on *circular security* [CL01]. The protocol proceeds as follows:

1. Parties homomorphically add random masks to $\mathsf{ct}_{\mathsf{sk}}$ while simultaneously sharing these masks via a *Publicly Verifiable Secret Sharing* (PVSS) scheme.
2. The masked secret key, $\mathsf{ct}_{\mathsf{masked}} := \mathsf{ct}_{\mathsf{sk}} \oplus \mathsf{Enc}_{\mathsf{pk}}(\mathsf{mask})$, is then threshold-decrypted.
3. The sharing of the masked values is converted into a new sharing of the secret key.

The key challenge is that the secret key space differs from the plaintext space of TAHE. This is resolved using limb decomposition, which preserves the correctness of the sharing within the key space. Importantly, the share size remains constant, as they are always derived from the same "fresh" $\mathsf{ct}_{\mathsf{sk}}$.

Although *circular security* might seem like a strong assumption, it has been proven for AHEs based on hidden-order groups within a modified "ElGamal-like" subgroup indistinguishability framework [BG10]. Specifically, this variant applies to Paillier encryption [BG10] and may extend to class-group-based schemes [BCIL23].

This protocol variant, which allows threshold adjustments, is proven secure in the standalone model due to challenges similar to those in Section 3.1, particularly in verifying keys. Reconfiguration is often sequential, blocking other sessions during execution which would still allow to use this protocol. However, the protocol can achieve UC-security by generating new random public parameters for the commitment scheme. For Paillier and class-group-based encryption, this involves a ciphertext element known to the simulator. One possible solution is to use hash-to-group function, which is straightforward for Paillier but challenging for class groups [SBK24].Another optio is adding an extra protocol round where each party sends an encryption with a UC-proof of validity, enabling the simulator to extract the message, which suffices for simulation.

Protocol 2: Fixed Total Weight and Threshold with Flexible Distribution - This protocol maintains the total weight and threshold but allows arbitrary redistribution among parties. It is suitable when the overall voting power and threshold remain unchanged, as in the SUI blockchain [BCD$^+$23b]. The steps are as follows:

1. Parties share a random value to rerandomize the existing secret key sharing.
2. Adjusted shares are distributed among parties.
3. The new shares are combined and reconstructed under the new configuration.

The primary challenge is bounding the size of the new shares. Although reconstruction initially yields larger shares, careful analysis shows they are divisible by $n!$[3] over the integers. After division, each share is the sum of the original share and a share of zero. Consequently, after N_r reconfigurations, a share is only $N_r + 1$ times larger, with its bit-length increasing by $\log(N_r)$—a manageable growth. Regarding security, since the final sharing combines the previous sk sharing with a fresh zero-sharing, it effectively constitutes a new integer-based sharing.

3.4 Complying with the 2PC-MPC framework

Finally, we consider integrating the client into our protocol. In the 2PC-MPC framework, the client run-time and communication overhead should be independent of the network size. It turns out that the challenge lies in the zk-proofs, since the statements that parties in the network send are always summed-up in our protocols. Unless there is a way to aggregate the zk-proofs of parties in the network into a single proof for the aggregated statement, as in [FMM+24], the client cannot receive proofs from the distributed party. We find proof aggregation to be quite challenging in an asynchronous fluid framework.

Instead, we design the protocol in a way that does not require any proofs from the distributed party towards the client. The basic concept is to view the client as a signing oracle, and as such it would not need any proofs. While this concept somewhat works for EdDSA, it fails for ECDSA and allows for a myriad of non-trivial attacks.

In our ECDSA 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 so public $x \cdot R_B$), and nonce k_A sampled by party A alone. 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 exception that the adversary can pick the generator of the group R_B every time.

Unfortunately, the above protocol enables the following attack. Party B sends X instead of R_B.[9] It will get $k_A^{-1} \cdot X$ and (the encryption of) $\sigma_A = k_A(m + rx)$. It then multiply $k_A^{-1} X$ 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 [Che06].

In order to resolve this issue, one might consider letting the client add an additive mask to the nonce of B. However, in order to do that and still allow the distributed party to generate a signature, the client must also give away $\mathsf{Enc}(k_B + \beta)$, which allows the adversary to calculate β, bringing us back to square one. Instead, we observe that allowing the client to apply an affine transformation and sending $\mathsf{Enc}(\alpha \cdot k_B + \beta)$ prevents this issue.

Again, this seems to shift too much power to the client. However, as long as the protocol enforces that the rerandomization of the nonces (the μs) depends on α, β and k_A, this extra power is still captured by the slightly enhanced signing oracle, which also allows the adversary to bias the presigns. To this end, the client sends homomorphic commitments and zk-proofs to α, β and k_A, and the derivation of the nonces depends on those commitments as well. Notably, in order for the simulation to work without rewinding, Fischlin [Fis05] UC-extractable proofs are used, and the nonces are derived from the proofs as well. This is important as the simulator needs to extract α, β, k_A before it simulates them.

[9] Sending a proof of knowledge for R_B prevents this, but our goal is to avoid it.

Notably, this incurs a negligible cost on performance, as it is required solely for the proof sent by the client. Moreover, verifying proofs over elliptic curves is significantly faster than over class-group based encryption schemes.

4 Preliminaries

Notation. Group elements and sets are denoted by big 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 superscript 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 or a miner.

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 A.2).

The internal communication channel between parties in B is an asynchronous reliable broadcast channel, which is captured in $\mathcal{F}_{\mathsf{broadcast}}$ (Functionality A.2). This is the underlying communication channel used in blockchains in order to agree on transactions to sign. Some of the potential instantiations can be found in [DKKSS22, SGSKK22, BCD$^+$23a]. Following [SS24], 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, we 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 A.3). A potential instantiation can be found in [SS24]. 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 [SS24], while $\mathcal{F}_{\mathsf{ACS}}$ is a rephrasing from [Sho24].

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 it 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 4.1). 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, pre-signing, and signing operations from the oracle. Any other type of query is controlled by the adversary. Additionally, the functionality captures changes to the network structure. It allows an authorized subset in Γ_B to reconfigure the network. It also enables a client to share its wallet with others and, when the network is not corrupted, to transfer ownership completely. This distinction is crucial: in a malicious network, enforcement of policies cannot be trusted. Consequently, a malicious party may collaborate with the network to recover the private signing key and regain access. For simplicity, the functionality assumes that presigns are tied to a specific key X. That being said for Schnorr we support key-independent presigns.

Zero-Knowledge Proofs (ZKPs). We recall the formal definition of *Zero-Knowledge Proofs* (ZKPs) and *Zero-Knowledge Proofs of Knowledge* (ZKPoK) in Appendix A.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*.

FUNCTIONALITY 4.1 (*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 initial access structure Γ_B. The functionality starts either with a record $(\mathsf{sid}_B, \Gamma_B)$ if $U \notin \Gamma_B$ or with the record $(\mathsf{sid}_B, \Gamma_B \cup \{\mathcal{A}\})$ otherwise.

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.
   (c) If $A_{\mathsf{pid}_A} \notin U$ record $(X, \{\mathsf{pid}_A\})$ otherwise, record $(X, \{\mathsf{pid}_A, \mathcal{A}\})$ instead.
2. **Presign**: Upon receiving $(\mathsf{pres}, X, \mathsf{sid}, \mathsf{ssid}, \mathsf{pid}_A)$ from $A_{\mathsf{pid}_A} \in A_X$ for which (X, A_X) is recorded, along with $(\mathsf{pres}, \mathsf{sid}, \mathsf{ssid}, \mathsf{pid}_A, \mathsf{pid}_B)$ from a subset $S \in \Gamma_B$, do as follows:
   (a) Send $(\mathsf{pres}, \mathsf{sid}, \mathsf{ssid}, X)$ to $\mathcal{G}$.
   (b) Act as a proxy between $\mathcal{G}$ and $\mathcal{A}$ until the oracle outputs $(\mathsf{pres\text{-}output}, \mathsf{sid}, \mathsf{ssid}, \vec{K})$ 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.
4. **Reconfiguration**: Upon receiving messages $(\mathsf{reconfigure}, \mathsf{sid}_B, \mathsf{pid}_B, \Gamma_{B,\mathsf{new}})$ from an authorized subset $S \in \Gamma_B$ check if either $\{\mathcal{A}\} \in \Gamma_B$ or $U \in \Gamma_{B,\mathsf{new}}$. If so record $(\mathsf{sid}_B, \Gamma_{B,\mathsf{new}} \cup \{\{\mathcal{A}\}\})$, otherwise record $(\mathsf{sid}_B, \Gamma_{B,\mathsf{new}})$.
5. **Transfer**: Upon receiving messages $(\mathsf{transfer}, X, \mathsf{pid}_A, A_{X,\mathsf{new}})$ from A_{pid_A} such that $\mathsf{pid}_A \in A_X$ for a recorded (X, A_X) check if either $\mathcal{A} \in A_X$ or $A_{X,\mathsf{new}} \cap U \neq \emptyset$. If so record $(X, A_{X,\mathsf{new}} \cup \{\mathcal{A}\})$, otherwise record $(X, A_{X,\mathsf{new}})$.

All of the ZKPs used are non-interactive, either by applying the Fiat-Shamir transform [FS86], or by applying Fischlin's transform [Fis05] 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 $\Pi_{\mathsf{uc\text{-}zk}}^{L[\mathtt{params}]}$ for the latter case. Formal definitions of the languages used in our protocol are provided in Appendix B.

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 $(\vec{Y}_i; \vec{w}_i)$ and validates the proofs of the other parties. Then, using $\mathcal{F}_{\mathsf{ACS}}$ the parties agree on a subset of the statements, and using $\mathcal{F}_{\mathsf{global\text{-}broadcast}}$, the statement $\sum_j \vec{Y}_j$ is sent to the client.

Shamir Secret Sharing Over the Integers. Shamir's original secret sharing scheme [Sha79a] was presented over a finite field. However, when working over a group of unknown order, [Rab98] suggested to work with a secret sharing scehem over the integers. Later, [FMM$^+$23] offered a more tight analysis and improved bounds on the secret shares size. including the improved bound on the secret shares size, and we follow their notation below.

Let $s \in \mathbb{Z} \cap [-b, b]$ be an integer secret, and denote $\Delta_n = n!$. The algorithm $\mathsf{Share}_{t,n}(s)$ samples a degree t polynomial $p(x) = \Delta_n \cdot s + \sum_{i=1}^{t} a_i x^i$, where $a_1, \cdots, a_t \leftarrow [-I(\sigma, n, b), I(\sigma, n, b)]$. Here, $I(\sigma, n, b)$ is some bound on the absolute value of the coefficients of the polynomial that is chosen to statistically hide the integer secret s. It then outputs $[s] := ([s]_1, \ldots, [s]_n)$ where $[s]_j = p(j)$ is the *secret share* of party j.

Reconstruction works as follows. Given a set S of $t+1$ secret shares, we have $p(0) = \sum_{j \in S} \lambda_{0,j}^S [s]_j = \Delta_n s$, where $\{\lambda_{i,j}^S\}_{j \in S}$ is the set of Lagrange coefficients corresponding to the interpolation of point i using the subset of interpolations points S. However, since the Lagrange coefficients may not be integers, they are first multiplied by Δ_n, and so $\sum_{j \in S} \Delta_n \lambda_{0,j}^S [s]_j / \Delta_n^2 = s$. We denote the upper bound on the absolute value of the share on s by $D(\sigma, n, t, b)$. This will correspond to the bound on the secret key share size of each party in B.

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 Petersen Commitments [Ped91] on the appropriate Elliptic Curve. Homomorphic Addition is denote by $\oplus$ and multiplication by a scalar 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 [CL15] or by interpreting the plaintext as a $\mathbb{Z}$-module using the Paillier scheme [Pai99], as done in [FMM$^+$24]. 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 encryption, 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 $\mathcal{F}_{\mathsf{DAHE}}$

(Functionality A.5), which also allows for dynamic changes to the access structure. 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 [BDO23] or Paillier-based TAHE with circuit privacy ([FMM+24], following [FMM+23]), along with the reconfiguration protocols (see Section 7). Both threshold schemes are *asynchronous*, meaning that the set of decrypting parties may be unknown in advance during decryption. 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.

Publicly Verifiable Secret Sharing (PVSS). Publicly Verifiable Secret Sharing (PVSS) [Sta96] enables a dealer to share a secret among a set of recipients for some monotone access structure, while any recipient can verify the correctness of the shares of every other recipient without further communication. Typically, this is achieved by the dealer generating encryptions under the public keys of each recipient while proving, along with ZKPs that the sharing was done correctly. The main advantage of this approach is that it avoids the need for an accountability mechanism in case a recipient claims they received an incorrect share, or did not receive there share at all.

We use PVSS in our reconfiguration protocols, in order to reshare the secret decryption key. In this context, we only need the Dist algorithm, responsible for distributing the secret in a publicly verifiable manner. The reconstruction phase is never invoked, and is only describe for compliance.

Extended Preliminaries. We refer to Appendix A, for a comprehensive background, formal definitions on: communication model A.1; ZKPs A.2; aggregation protocol A.3; SSS over $\mathbb{Z}$ A.4; homomorphic commitments A.5; TAHE A.6; and PVSS A.7.

5 ECDSA Based Protocol

Next, we describe the ECDSA protocol, consisting of three sub-protocols: distributed key generation, presign, and sign. Full details and proofs of correctness appear in Appendix C.1. Intuitively, the protocol should be thought of as a two-party protocol, which is then transformed into a 2PC-MPC protocol using TAHE. Thus all messages sent by the client even if encrypted by the TAHE should not reveal any information on its secrets. Let us describe the honest two-party protocol in short:

1. **Distributed Key Generation**:
   (a) B samples $x_{0,B}, x_{1,B} \leftarrow \mathbb{Z}_q$ and sends $X_{0,B} := x_{0,B} \cdot G$ and $X_{1,B} := x_{1,B} \cdot G$ to A.
   (b) A samples $x_A \leftarrow \mathbb{Z}_q$ and sends $X_A = x_A \cdot G$ to B.
   (c) The parties then call the Random Oracle $\mathcal{H}$ on $X_A, X_{0,B}, X_{1,B}$ and other necessary values to receive $\mu_x^0, \mu_x^1, \mu_x^G$, and set $X = X_A + \mu_x^0 \cdot X_{0,B} + \mu_x^1 \cdot X_{1,B} + \mu_x^G \cdot G$. Additionally, B sets $x_B = \mu_x^0 \cdot x_{0,B} + \mu_x^1 \cdot x_{1,B} + \mu_x^G$.

2. **Presign**: B samples $\gamma, k_0, k_1 \leftarrow \mathbb{Z}_q$, and sends $R_0 = k_0 \cdot G, R_1 = k_1 \cdot G, \mathsf{ct}_\gamma = \mathsf{AHE.Enc}(\gamma), \mathsf{ct}_{\gamma \cdot k_0} = \mathsf{AHE.Enc}(\gamma k_0), \mathsf{ct}_{\gamma \cdot k_1} = \mathsf{AHE.Enc}(\gamma k_1)$, and $\mathsf{ct}_{\gamma \cdot \mathsf{key}} = \mathsf{AHE.Enc}(\gamma x_B)$ to A.
3. **Sign**:
   (a) A samples $k_A, \alpha, \beta \leftarrow \mathbb{Z}_q$ and sets $m = \mathcal{H}(\mathsf{msg})$. It then calls the random oracle on the entire transcript, including msg and commitments on α, β, and k_A, to get randomizers $\mu_k^0, \mu_k^1, \mu_k^G$.
   (b) A sets $R = k_A^{-1}(\alpha(\mu_k^0 \cdot R_0 + \mu_k^1 \cdot R_1 + \mu_k^G \cdot G) + \beta \cdot G)$ and extracts its X coordinate, denoted as r. It homomorphically computes the ciphertexts $\mathsf{ct}_{\alpha,\beta} = \mathsf{AHE.Enc}\left(\gamma\left(\alpha(\mu_k^0 k_0 + \mu_k^1 k_1 + \mu_k^G) + \beta\right)\right)$ and $\mathsf{ct}_A = \mathsf{AHE.Enc}\left(\gamma(k_A(m + r(x_A + x_B)))\right)$. It sends these values to B.
   (c) B decrypts $\mathsf{ct}_{\alpha,\beta}, \mathsf{ct}_A$ to get $\mathsf{pt}_4, \mathsf{pt}_A$ respectively and calculate $s = \mathsf{pt}_4^{-1} \cdot \mathsf{pt}_A$.

The scalars $\mu_x^0, \mu_x^1, \mu_x^G$ are used to randomize the secret key and to limit A's influence on it. Similarly, the scalars $\mu_k^0, \mu_k^1, \mu_k^G$ serve to restrict A's control over the random exponent k of the signature. In contrast, γ is employed for blinding, ensuring that publishing pt_4 and pt_A does not reveal any secret information. While this blinding seems to be redundant in the 2PC protocol, it will be needed in the 2PC-MPC protocol to prevent a malicious A to collude with a single party in B. The transformation between the protocols revolves around the distribution of B and the enforcement of honest behavior from A and B.

Key Generation. In the first step, B samples $x_{0,B}$ and $x_{1,B}$ using Protocol A.4 (the aggregation protocol). Then, in addition to sending the public values $X_{0,B}, X_{1,B}$, B also sends encryptions of the secret values $x_{0,B}, x_{1,B}$. These values will be used later to prove honest behavior of B. In the second step, A samples its share x_A and sends its part of the public key X_A along with a UC-extractable zk-proof.

Presign. The ECDSA presign protocol is executed only by B. Using the aggregation protocol B samples γ and generates $\mathsf{ct}_\gamma = \mathsf{Enc}(\gamma)$ and $\mathsf{ct}_{\gamma \cdot \mathsf{key}} = \mathsf{Enc}(\gamma x_B)$. In the next round, it generates $\mathsf{ct}_{\gamma \cdot k_0} = \mathsf{Enc}(\gamma k_0), \mathsf{ct}_{\gamma \cdot k_1} = \mathsf{Enc}(\gamma k_1)$ and $R_0 = k_0 \cdot G, R_1 = k_1 \cdot G$, using the aggregation protocol again.

Sign. The sign protocol is applied by A itself, and focus on enforcing its honest behavior. First, A samples k_A, α, β and computes corresponding homomorphic commitments C_k, C_α, C_β and corresponding ZK proofs π_k, π_α and a UC-extractable proof π_β. To prove that k_A and α are non-zero, we use a public parameter inversion trick (see Appendix D). Then, A calls the random oracle on the entire transcript, including all these commitments and proofs, to get the randomizers $\mu_k^0, \mu_k^1, \mu_k^G$. Next, A computes $\mathsf{ct}_{\alpha,\beta}$ and ct_A as in the two-party protocol, along with corresponding zk-proofs of honest behavior. Finally, the distributed party verifies the proofs and public computations, and uses $\mathcal{F}_{\mathsf{TAHE}}$ to decrypt $\mathsf{ct}_A, \mathsf{ct}_{\alpha,\beta}$ and compute the signature.

6 Schnorr-Based Protocol

Next, we describe the Schnorr-based protocol. We refer to Appendix C.2 for a detailed description of the distributed key generation, presign and sign protocols and proofs of their correctness. We begin with the honest two-party protocol:

1. **Distributed Key Generation**: (as in Section 5 with $x_{1,B} = 0, \mu_x^1 = 1, \mu_x^G = 0$)
2. **Presign**: B samples $k_0, k_1 \leftarrow \mathbb{Z}_q$ and calculates $K_{B,0} = k_0 \cdot G$ and $K_{B,1} = k_1 \cdot G$ which are sent 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 get 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 then 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 computes K. Then, it computes $z = z_A + k_0 + \mu_k \cdot k_1 + e \cdot x_B$, and outputs (z, e).

Again, we make use of $\mathcal{F}_{\mathsf{TAHE}}$ to comply with the 2PC-MPC design. In contrast to the ECDSA protocol, here we need not use zero-knowledge proofs to enforce honest behavior of A (except for key-gen). Essentially, this is due to the nature of Schnorr signatures. Specifically, the signature part of A consists a zero-knowledge proof of knowledge of $\log_G K_A$.

Presign. The presign protocol consists of a single aggregation round in B, that collectively generates $K_{B,0}$, $K_{B,1}$, $\mathsf{ct}_{k_0} = \mathsf{Enc}(k_0)$, and $\mathsf{ct}_{k_1} = \mathsf{Enc}(k_1)$. Only $K_{B,0}$ and $K_{B,1}$ are sent to A. The values of ct_{k_0} and ct_{k_1} will only be used by B in the signing protocol to produce the signature part of B.

Notably, the presign is independent of the public key X. Thus, in case the distributed party B is a system that interacts with several clients A_i, the presign output can be used by any of them for signing. Importantly, each presign still can only be used once.

Sign. Party A acts exactly as in the honest two-party protocol described above. Then, B first verifies that $K_A + eX_A = z_A \cdot G$. It then uses ct_{k_0} and ct_{k_1} and $\mathsf{ct}_{\mathsf{key}}$ to compute its encrypted share of the signature, decrypts it using $\mathcal{F}_{\mathsf{TAHE}}$, and computes the response part $z = z_A + z_B$.

7 Reconfiguration

As described in Section 1.2, reconfiguration allows the distributed party to be instantiated with a permissionless decentralized PoS blockchain, wherein the weight of each party corresponds to its voting power. At the end of each *epoch*, validators may stake, withdraw stake, join or leave, affecting the distribution of the overall voting power.

By the 2PC-MPC design, the only secret that is shared in the distributed party B is the TAHE's secret key sk. Therefore, reconfiguration boils down to resharing the secret key sk, a process whose complexity is independent of the number of clients in A.

We present two protocols for reconfiguration, that is, for resharing sk. The first one also supports changing the threshold. This is used to adjust the voting power distribution. Both protocols use an asynchronous PVSS scheme (see Definition A.6) such as [Sch99, GHL22, CD24a]. We choose PVSS over VSS since we already work over a broadcast channel. In our setting, the parties never reconstruct the secret and a correct decryption proof is not needed for the PVSS.

Instead, we ensure that the new verification keys of the TAHE scheme are consistent with the new shares of sk by adding a ZK-proof tying the encryptions of the shares to homomorphic commitments corresponding to the verification keys. This commitment provided by $\mathsf{PVSS.Dist}$ is denoted by C

Notably, in permissionless blockchains the joining validators are not considered part of the validator network until the very beginning of the new epoch, and therefore cannot actively participate in the reconfiguration protocol. Instead, they may only provide input to the reconfiguration protocol by locking their stake, and may receive output by querying the blockchain state. Intuitively, joining validators are only exposed to the client communication channel, and cannot directly communicate with other validators, not even via the internal broadcast channel. To this end, in both protocols the old quorum B computes a broadcast message which allows the parties in the new quorum B' to locally calculate their shares, and validate the verification keys.

Reconfiguration with a varying threshold. We observe that encryption under pk is actually a way to share the corresponding plaintext, as it can be reconstructed by applying threshold decryption. The idea is to share the TAHE secret key sk by encrypting it under pk. We denote this encryption by $\mathsf{ct}_{\mathsf{sk}} = \mathsf{Enc}_{\mathsf{pk}}(\mathsf{sk})$. The old quorum then creates an encryption of a random mask $\mathsf{ct}_{\mathsf{mask}} = \mathsf{Enc}(r)$ which is used to mask the secret key sk under the encryption. Simultaneously, they create a sharing of the mask $[r]$ that corresponds to the new access structure, and the two are tied using zk-proofs. The old quorum then preforms threshold decryption of $\mathsf{ct}_{\mathsf{sk}} \oplus \mathsf{ct}_{\mathsf{mask}}$ and broadcast $\mathsf{sk} + r$. The new quorum then derives from $[r]$ and $\mathsf{sk} + r$ a fresh sharing $[\mathsf{sk}]$ of the secret key. A detailed description is presented in Section C.3 and Protocol C.6.

Nevertheless, the key space $\mathcal{K}_\kappa$ and plaintext space $\mathcal{P}_{pk}$ of the TAHE need not be aligned, and therefore $\mathsf{ct}_{\mathsf{sk}}$ is not necessarily well-defined. In order to resolve this issue, we *decompose* the secret key into *limbs* small enough such that they can be encrypted and operated upon (to a certain extent) without going over the size of the plaintext space.

Reconfiguration with a constant threshold. The idea behind this protocol is to let a threshold of parties from the old quorum to share their shares with respect to the new quorum, and then let each party in the new quorum reconstruct its shares by applying Lagrange interpolation.

For security, it is important to rerandomize the shares, as otherwise an adversary can combine shares from previous configurations to extract the secret key. Therefore, parties in the old quorum first generate two sharings of the same random value r, $[r]_{\mathsf{old}}, [r]_{\mathsf{new}}$, where one is designated for the old quorum and the other for the new quorum. We stress that both shares are over the same overall voting power and threshold, and so the interpolation points are the same. The difference between the two shares is which party gets which shares.

The old quorum can then mask their key shares using the random shares and get $[\mathsf{sk} + r]_{\mathsf{old}}$. The masked key shares are subsequently reshared among the old quorum. By applying Lagrange interpolation, the old quorum can derive a sharing over each share $[\mathsf{sk} + r]_i^{\mathsf{old}}$. Then, they can send to each party i in the new quorum their share of $[\mathsf{sk}+r]_i^{\mathsf{old}}$. Finally, each party in the new quorum i can then reconstruct $[\mathsf{sk}+r]_i^{\mathsf{old}}$, and by subtracting $[r]_i^{\mathsf{new}}$, it gets $[\mathsf{sk}]_i^{\mathsf{new}}$. That is, its designated shares with respect to the new quorum and fresh randomness, of the secret sk.

The parties use deterministic commitment schemes (which are statistically binding and computationally hiding) to verify the consistency of the sharing. See Section C.3 and Protocol C.7 for full details.

The biggest complication in this framework arises from the fact that, in certain TAHE schemes, the sharing of the secret key is performed over the integers. This specifically affects schemes based on groups of unknown order, such as Paillier and Class Groups. During interpolation, the interpolated value may include a factor of Δ_n. This could theoretically cause a significant increase in share size (as we create a sharing of a share). To address this, we construct the sharing in a way that enables division by Δ^3, normalizing the share size, which thus grows minimally. In particular, we show that the new share is the sum of the old polynomial and the polynomial used to share 0 at the same point.

Another issue arises because the final verification key is actually received raised to the power of Δ^3. This can be resolved by adding a communication round in which each party sends the $\frac{1}{\Delta^3}$ root of the verification key, and the other parties can locally verify that it is indeed the correct root. Alternatively, the modified verification key can be used for the TAHE scheme, and the root can be sent during the next reconfiguration phase. This shows that the shares grow logarithmically with the number of reconfigurations. Therefore, we can effectively work with a constant bound on share size, e.g. by bounding the number of reconfigurations by the security parameter κ, with no significant overhead.

8 Security

In this section, we prove the security of our protocol for ECDSA (in 8.1), Schnorr (in 8.2) and reconfiguration (in 8.3). Omitted proofs are in Appendix D.

8.1 Analyzing Slightly Enhanced ECDSA

In this section, we provide a sketch of the security proof of the ECDSA signing oracle 8.1 in the EC-GGM model [GS22]. Our proof follows the steps taken in [GS22], and we refer to Appendix D.1 for a detailed proof.

As in [GS22], 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 [GS22], 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.

The security proof of the symbolic simulation is based on the symbolic verification equation of the forged signature and the equality of symbolic polynomials.

FUNCTIONALITY 8.1 (*Slightly Enhanced ECDSA Signing Oracle:* $\mathcal{G}^*_{\textsf{SE-ECDSA}}$)

1. On input $(\textsf{keygen}, \textsf{sid})$, sample $\tilde{x}_0, \tilde{x}_1 \leftarrow \mathbb{Z}_q$ and send $\tilde{X}_0 = \tilde{x}_0 \cdot G$ and $\tilde{X}_1 = \tilde{x}_1 \cdot G$. Upon receiving $(\textsf{biaskey}, \textsf{sid}, \alpha_x^{(0)}, \beta_x^{(0)}, \alpha_x^{(1)}, \beta_x^{(1)})$ 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 $(\mu_x^0, \mu_x^1, \mu_x^G) = \mathcal{H}(\textsf{sid}, X_0, X_1)$. 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 $(\textsf{sid}, X; x)$.
2. On input $(\textsf{pres}, \textsf{sid}, \textsf{ssid})$, 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 $(\textsf{sid}, \textsf{ssid}, \tilde{R}_0, \tilde{R}_1; \tilde{k}_0, \tilde{k}_1)$ and send $(\textsf{sid}, \textsf{ssid}, \tilde{R}_0, \tilde{R}_1)$.
3. On input $(\textsf{sign}, \textsf{sid}, \textsf{ssid}, \textsf{msg}; \beta'_{\textsf{key}}, \alpha_{\textsf{pres},0}, \beta_{\textsf{pres},0}, \alpha_{\textsf{pres},1}, \beta_{\textsf{pres},1})$ do:
   (a) Retrieve $(\textsf{sid}, X; x)$ and $(\textsf{sid}, \textsf{ssid}, \tilde{R}_0, \tilde{R}_1; \tilde{k}_0, \tilde{k}_1)$. If no such $\textsf{ssid}$ or $\textsf{sid}$ exist, ignore.
   (b) Compute $x' \leftarrow x + \beta'_{\textsf{key}}$ and $X' \leftarrow x' \cdot G$.
   (c) Compute $k_0 \leftarrow \alpha_{\textsf{pres},0} \cdot \tilde{k}_0 + \beta_{\textsf{pres},0}$ and $k_1 \leftarrow \alpha_{\textsf{pres},1} \cdot \tilde{k}_1 + \beta_{\textsf{pres},1}$.
   (d) Set $R_0 = k_0 \cdot G$ and $R_1 = k_1 \cdot G$.
   (e) Set $(\mu_k^0, \mu_k^1, \mu_k^G) = \mathcal{H}(\textsf{sid}, \textsf{ssid}, X, \beta'_{\textsf{key}}, R_0, R_1, \textsf{msg})$, and set $k = \mu_k^0 \cdot k_0 + \mu_k^1 \cdot k_1 + \mu_k^G$ and $R = k \cdot G$.
   (f) Set $r = R_{x-axis}$ and compute $s = k^{-1} \cdot (\mathcal{H}(\textsf{msg}) + r \cdot x')$.
   (g) Erase $(\textsf{sid}, \textsf{ssid}, \tilde{R}_0, \tilde{R}_1; \tilde{k}_0, \tilde{k}_1)$ from memory and return $(\textsf{sid}, \textsf{ssid}, \textsf{msg}; \beta'_{\textsf{key}}, \alpha_{\textsf{pres},0}, \beta_{\textsf{pres},0}, \alpha_{\textsf{pres},1}, \beta_{\textsf{pres},1}; R, s, r)$.

Then, following [GS22], 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 and deduce:

Theorem 8.1 (D.3) *Let $\mathcal{A}$ be an adversary to the existential unforgeability game D.6 with respect to the ECDSA oracle $\mathcal{G}^*_{\textsf{SE-ECDSA}}$ (Functionality 8.1) in the EC-GGM model (LazySimulation D.1), that makes at most N presignature, signing, hash, or group queries, when $\mathcal{H}_{\textsf{key}}, \mathcal{H}_{\mathcal{M}}$ and all $\mathcal{H}_k$ are modeled as independent Random Oracles. Then $\textsf{Adv}(\mathcal{A}, \textsf{Exp}_{\textsf{EU}}^{\mathcal{G}^*_{\textsf{SE-ECDSA}}}, \textit{lazy-sim}) \leq \mathcal{O}(\frac{N^2}{q})$.*

We note that our proof also handles Additive Key Derivation, see Appendix D.1 for details.

8.2 Analyzing Slightly Enhanced Schnorr

In this section, we provide a sketch of the proof of the security of the Schnorr signing oracle 8.2. Specifically, we reduce signature forgery into breaking the Algebraic One More Discrete Log problem (AOMDL) D.7. In AOMDL security game, the challenger generates $n_{\textsf{DL}} + 1$ group elements, then $\mathcal{A}$ asks for $n_{\textsf{DL}}$ linear combinations of their discrete log values, and then must output all $n_{\textsf{DL}} + 1$ discrete logs to win. The detailed proof is in Appendix D.2.

Theorem 8.2 *If there is a PPT algorithm $\mathcal{A}^{n_s, n_{kg}, n_r}_{\textsf{SE-Sch-Forger}}$ that satisfies $\textsf{Exp}_{\textsf{EU}}^{\mathcal{G}^*_{\textsf{SE-Sch}}}(\mathcal{A}, 1^\kappa, n_s, n_{kg})$ with probability at least ϵ, and uses at most n_r calls to the random oracle, then there is a PPT algorithm $\mathcal{A}'_{2n_s+n_{kg}}$ that satisfies $\textsf{Exp}_{\textsf{AOMDL}}^{2n_s+n_{kg}}(\mathcal{A}, 1^\kappa, (\mathbb{G}, G, q))$ with probability at least $\epsilon \left(\frac{\epsilon}{n_r} - \frac{1}{|H|}\right)\left(1 - \frac{n_s}{|H|}\right)$ with expected running time $2\,TIME(\mathcal{A}^{n_s, n_{kg}, n_r}_{\textsf{SE-Sch-Forger}}) + \mathcal{O}(n_r + n_s + n_{kg})$.*

FUNCTIONALITY 8.2 (*Slightly Enhanced Schnorr Signing Oracle:* $\mathcal{G}^*_{\textsf{SE-Sch}}$)

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 $(\mathsf{sid}, \mathsf{ssid}, \mathsf{msg}; \alpha_{\mathsf{key}}, \beta_{\mathsf{key}}, \alpha_{\mathsf{pres},0}, \beta_{\mathsf{pres},0}, \alpha_{\mathsf{pres},1}, \beta_{\mathsf{pres},1}, K_0 + \mu \cdot K_1, z)$.

Essentially, $\mathcal{A}'_{2n_{\mathsf{s}}+n_{\mathsf{kg}}}$ will simulate $\mathcal{G}^*_{\textsf{SE-Sch}}$ when interacting with $\mathcal{A}^{n_{\mathsf{s}},n_{\mathsf{kg}},n_{\mathsf{r}}}_{\mathsf{SE-Sch-Forger}}$, and will use discrete log queries to the AOMDL challenger in order to compute the signatures.

$\mathcal{A}'$ receives $n_{\mathsf{DL}} + 1 = 2n_{\mathsf{s}} + n_{\mathsf{kg}}$ group elements. The first $2n_{\mathsf{s}}$ are the nonce of the pre-sign queries and the last n_{kg} are the public keys. Upon receiving a signing request from $\mathcal{A}$, $\mathcal{A}'$ can compute the randomizer μ_k and the challenge e. The signature z is then the linear combination $(1, \mu_k, e)$ of the discrete logs of the corresponding presigns and public key. Therefore, $\mathcal{A}'$ can compute the signature with a proper query to the AOMDL challenger.

Next, $\mathcal{A}$ should output a forged signature (if not, $\mathcal{A}'$ aborts), which for Schnorr gives a linear equation over the discrete logs of the public key and the nonce K^* introduced by $\mathcal{A}$. Since K^* may not be part of the AOMDL challenges, the signature is a linear equation with the new variable k^*. Therefore, another such equation is obtained by rewinding $\mathcal{A}$ to when e^* was returned from the random oracle, and returning instead $e^{**} \neq e^*$ sampled at random. By the Generalized Forking Lemma [BN06, Lemma 1], we show that with non-negligible probability, $\mathcal{A}$ returns a second forgery using the same public key and the same ephemeral key K^*. Using the forgeries, $\mathcal{A}'$ extracts the discrete log of the corresponding public key, and wins the AOMDL game.

8.3 Protocol Simulations.

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

Signing Protocols. Both the ECDSA and Schnorr based protocols UC realize their corresponding threshold signing functionality (Functionality 4.1 with Oracle 8.1 and Oracle 8.2) respectively in the *reliable broadcast model* and $\mathcal{F}_{\mathsf{DAHE}}$ (Functionality A.5)-hybrid model.

Theorem 8.3 *The protocols Π_{keygen}, $\Pi^{\mathsf{ECDSA}}_{\mathsf{presign}}$, and $\Pi^{\mathsf{ECDSA}}_{\mathsf{sign}}$ UC realize $\mathcal{F}^{\mathcal{G}^*_{\textsf{SE-ECDSA}}, \binom{[n]}{t+1}}_{\mathsf{tsig}}$ in the* reliable broadcast model *and $\mathcal{F}_{\mathsf{DAHE}}$-hybrid model..*

Theorem 8.4 *The protocols Π_{keygen}, $\Pi^{\mathsf{Schnorr}}_{\mathsf{presign}}$, and $\Pi^{\mathsf{Schnorr}}_{\mathsf{sign}}$ UC-realize $\mathcal{F}_{\mathsf{tsig}}^{\mathcal{G}^*_{\mathsf{SE\text{-}Sch}},\binom{[n]}{t+1}}$ (Functionality 4.1 with Oracle 8.2) in the* broadcast model *and $\mathcal{F}_{\mathsf{DAHE}}$-hybrid model.*

We refer the reader to Appendix D.1 and D.2 for the proofs.

Reconfiguration Simulation. Both reconfiguration protocols may be combined with a wide class of TAHEs to realize a dynamical access structure TAHE (Functionality A.5) in our *reliable broadcast model*. There are two types of dynamical access structures that we address. One with a constant threshold (Protocol C.7) that our protocol UC realizes, and one with a dynamic threshold (Protocol C.6) that our protocol stand-alone realizes.

Theorem 8.5 *Let TAHE be a simulatable threshold encryption scheme, i.e. there exists simulator $\mathcal{S}_{\mathsf{keygen}}$ and $\mathcal{S}_{\mathsf{TDec}}$ simulating key generation and threshold decryption commands of $\mathcal{F}_{\mathsf{TAHE}}$. Then TAHE along with either of the protocols $\Pi^{\mathsf{var\text{-}th}}_{\mathsf{ReConfigure}}$ or $\Pi^{\mathsf{cons\text{-}th}}_{\mathsf{ReConfigure}}$ realizes $\mathcal{F}_{\mathsf{DAHE}}$ in the* reliable broadcast model.

We refer the reader to Appendix D.3 for the proof. For discussion about instantiating $\mathcal{F}_{\mathsf{DAHE}}$ with specific TAHE schemes we refer the reader to Appendix F.

9 Performance

We implemented our ECDSA protocol in Rust. The implementation and benchmark code will be released as open source. We instantiated our TAHE with threshold Paillier based on [FMM+23] and with Class Groups based on [BCD+24], all times are reported for $t = \frac{2}{3}n$. We note that our Class-Groups based implementation is about about 5x slower compared to [BCIL23] and it similarly uses vartime operations (moving to constant time should mostly effect the running time of the client). All experiments are conducted over a MacBook Pro Apple M1 Max with a 3.22GHz CPU on a single thread. We report the raw tables for our experiments in Appendix E.

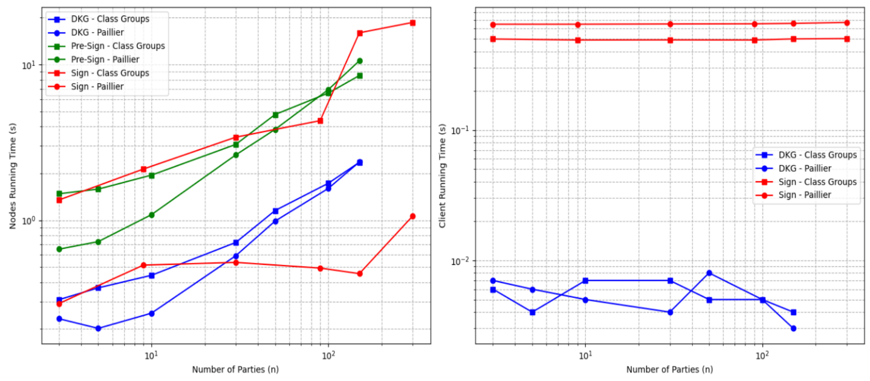

Fig. 1: Run-time of nodes and client vs number of parties in B. As expected, the client run-time is independent of network size. The total network size determines the cost of the the DKG and the pre-sign protocols, while the total share power determines the cost of the sign protocol. Thus, we are interested in the sign protocol performance for greater values of n. DKG is significantly faster as it does not involve encryptions. The node run-time in DKG and presign is linear with network size due to the verification of zk proofs from each party. The sign complexity is super-linear as the decryption share size also increases with the number of parties. However, with the amortized recombination optimization (Section F.3), the amortized cost is divided by $n/2$.

References

ACK$^+$20. Erdinc Akyildirim, Shaen Corbet, Paraskevi Katsiampa, Neil Kellard, and Ahmet Sensoy. The development of bitcoin futures: Exploring the interactions between cryptocurrency derivatives. *Finance Research Letters*, 34:101234, 2020.

Ali20. Muneeb Ali. Stacks 2.0 apps and smart contracts for bitcoin. *Recuperado el*, 8, 2020.

ANO$^+$22. Damiano Abram, Ariel Nof, Claudio Orlandi, Peter Scholl, and Omer Shlomovits. Low-bandwidth threshold ECDSA via pseudorandom correlation generators. In *SP*, pages 2554–2572. IEEE, 2022.

AYSS09. Tuncer Can Aysal, Mehmet Ercan Yildiz, Anand D Sarwate, and Anna Scaglione. Broadcast gossip algorithms for consensus. *IEEE Transactions on Signal processing*, 57(7):2748–2761, 2009.

B$^+$13. Vitalik Buterin et al. Ethereum white paper. *GitHub repository*, 1:22–23, 2013.

BA23. J. Kiguru B. Akolkar. Cardano: New upgrade to support ecdsa and schnorr cryptographic signatures to make it easier for developers to build cross-chain apps. *Crypto NewsFlash https://www.crypto-news-flash.com*, 2023.

BAJ23. P. BAJPAI. Cryptocurrency futures: Definition and how they work on exchanges. *Investopedia https://www.investopedia.com/articles/investing/012215/how-invest-bitcoin-exchange-futures.asp*, 2023.

Bal24. Kannan Balasubramanian. Security of the secp256k1 elliptic curve used in the bitcoin blockchain. *Indian Journal of Cryptography and Network Security (IJCNS)*, 4(1):1–5, 2024.

BBB$^+$18. Benedikt Bünz, Jonathan Bootle, Dan Boneh, Andrew Poelstra, Pieter Wuille, and Greg Maxwell. Bulletproofs: Short proofs for confidential transactions and more. In *2018 IEEE symposium on security and privacy (SP)*, pages 315–334. IEEE, 2018.

BBHP22. Michael Backes, Pascal Berrang, Lucjan Hanzlik, and Ivan Pryvalov. A framework for constructing single secret leader election from mpc. In *European Symposium on Research in Computer Security*, pages 672–691. Springer, 2022.

BCD$^+$23a. Kushal Babel, Andrey Chursin, George Danezis, Lefteris Kokoris-Kogias, and Alberto Sonnino. Mysticeti: Low-latency dag consensus with fast commit path. *arXiv preprint arXiv:2310.14821*, 2023.

BCD$^+$23b. Same Blackshear, Andrey Chursin, George Danezis, Anastasios Kichidis, Lefteris Kokoris-Kogias, Xun Li, Mark Logan, Ashok Menon, Todd Nowacki, Alberto Sonnino, et al. Sui lutris: A blockchain combining broadcast and consensus. *arXiv preprint arXiv:2310.18042*, 2023.

BCD$^+$24. Lennart Braun, Guilhem Castagnos, Ivan Damgård, Fabien Laguillaumie, Kelsey Melissaris, Claudio Orlandi, and Ida Tucker. An improved threshold homomorphic cryptosystem based on class groups. *Cryptology ePrint Archive*, 2024.

BCIL23. Cyril Bouvier, Guilhem Castagnos, Laurent Imbert, and Fabien Laguillaumie. I want to ride my bicycl: Bicycl implements cryptography in class groups. *Journal of Cryptology*, 36(3):17, 2023.

BD23. LTAN Brandão and Michael Davidson. Notes on threshold eddsa/schnorr signatures, 2023.

BDL$^+$12. Daniel J Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, and Bo-Yin Yang. High-speed high-security signatures. *Journal of cryptographic engineering*, 2(2):77–89, 2012.

BDO23. Lennart Braun, Ivan Damgård, and Claudio Orlandi. Secure multiparty computation from threshold encryption based on class groups. In *Annual International Cryptology Conference*, pages 613–645. Springer, 2023.

BG10. Zvika Brakerski and Shafi Goldwasser. Circular and leakage resilient public-key encryption under subgroup indistinguishability: (or: Quadratic residuosity strikes back). In *Advances in Cryptology–CRYPTO 2010: 30th Annual Cryptology Conference, Santa Barbara, CA, USA, August 15-19, 2010. Proceedings 30*, pages 1–20. Springer, 2010.

BGG17. Dan Boneh, Rosario Gennaro, and Steven Goldfeder. Using level-1 homomorphic encryption to improve threshold DSA signatures for bitcoin wallet security. In *LATINCRYPT*, volume 11368 of *Lecture Notes in Computer Science*, pages 352–377. Springer, 2017.

BGG$^+$18. Dan Boneh, Rosario Gennaro, Steven Goldfeder, Aayush Jain, Sam Kim, Peter MR Rasmussen, and Amit Sahai. Threshold cryptosystems from threshold fully homomorphic encryption. In *Advances in Cryptology–CRYPTO 2018: 38th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 19–23, 2018, Proceedings, Part I 38*, pages 565–596. Springer, 2018.

BLL$^+$22. Fabrice Benhamouda, Tancrède Lepoint, Julian Loss, Michele Orrù, and Mariana Raykova. On the (in)security of ros. *Journal of Cryptology*, 35(4):25, 2022.

BLT$^+$24. Renas Bacho, Julian Loss, Stefano Tessaro, Benedikt Wagner, and Chenzhi Zhu. Twinkle: Threshold signatures from ddh with full adaptive security. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 429–459. Springer, 2024.

BMP22. Constantin Blokh, Nikolaos Makriyannis, and Udi Peled. Efficient asymmetric threshold ecdsa for mpc-based cold storage. *Cryptology ePrint Archive*, 2022.

BN06. Mihir Bellare and Gregory Neven. 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*, pages 390–399, 2006.

Bol02. Alexandra Boldyreva. Threshold signatures, multisignatures and blind signatures based on the gap-diffie-hellman-group signature scheme. In *International Workshop on Public Key Cryptography*, pages 31–46. Springer, 2002.

Bra87. Gabriel Bracha. Asynchronous byzantine agreement protocols. *Information and Computation*, 75(2):130–143, 1987.

BT83. Gabriel Bracha and Sam Toueg. Resilient consensus protocols. In *Proceedings of the second annual ACM symposium on Principles of distributed computing*, pages 12–26, 1983.

BT85. Gabriel Bracha and Sam Toueg. Asynchronous consensus and broadcast protocols. *Journal of the ACM (JACM)*, 32(4):824–840, 1985.

BTZ22. Mihir Bellare, Stefano Tessaro, and Chenzhi Zhu. Stronger security for non-interactive threshold signatures: Bls and frost. *Cryptology ePrint Archive*, 2022.

CBC22. Panagiotis Chatzigiannis, Foteini Baldimtsi, and Konstantinos Chalkias. Sok: Blockchain light clients. In *International Conference on Financial Cryptography and Data Security*, pages 615–641. Springer, 2022.

CCL$^+$20. Guilhem Castagnos, Dario Catalano, Fabien Laguillaumie, Federico Savasta, and Ida Tucker. Bandwidth-efficient threshold EC-DSA. In *PKC*, volume 12111 of *Lecture Notes in Computer Science*, pages 266–296. Springer, 2020.

CCL$^+$23. Guilhem Castagnos, Dario Catalano, Fabien Laguillaumie, Federico Savasta, and Ida Tucker. Bandwidth-efficient threshold ec-dsa revisited: Online/offline extensions, identifiable aborts proactive and adaptive security. *Theoretical Computer Science*, 939:78–104, 2023.

CD24a. Ignacio Cascudo and Bernardo David. Publicly verifiable secret sharing over class groups and applications to dkg and yoso. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 216–248. Springer, 2024.

CD24b. Cassano Chris and Sneider David. Lit protocol whitepaper. *Github Repository https://github.com/LIT-Protocol/whitepaper*, 2024.

CDKS24. Ran Cohen, Jack Doerner, Yashvanth Kondi, and Abhi Shelat. Secure multiparty computation with identifiable abort via vindicating release. In *Annual International Cryptology Conference*, pages 36–73. Springer, 2024.

CDN01. Ronald Cramer, Ivan Damgård, and Jesper B Nielsen. 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*, pages 280–300. Springer, 2001.

CDSG$^+$24. Annalisa Cimatti, Francesco De Sclavis, Giuseppe Galano, Sara Giammusso, Michela Iezzi, Antonio Muci, Matteo Nardelli, and Marco Pedicini. Dynamic-frost: Schnorr threshold signatures with a flexible committee. *Cryptology ePrint Archive*, 2024.

CGG$^+$20. Ran Canetti, Rosario Gennaro, Steven Goldfeder, Nikolaos Makriyannis, and Udi Peled. UC non-interactive, proactive, threshold ECDSA with identifiable aborts. In *CCS*, pages 1769–1787, 2020.

CGG$^+$21. Arka Rai Choudhuri, Aarushi Goel, Matthew Green, Abhishek Jain, and Gabriel Kaptchuk. Fluid mpc: secure multiparty computation with dynamic participants. In *Advances in Cryptology–CRYPTO 2021: 41st Annual International Cryptology Conference, CRYPTO 2021, Virtual Event, August 16–20, 2021, Proceedings, Part II 41*, pages 94–123. Springer, 2021.

CGHN97. Ran Canetti, Rosario Gennaro, Amir Herzberg, and Dalit Naor. Proactive security: Long-term protection against break-ins. *RSA Laboratories' CryptoBytes*, 3(1):1–8, 1997.

CGR11. Christian Cachin, Rachid Guerraoui, and Lus Rodrigues. *Introduction to Reliable and Secure Distributed Programming*. Springer Publishing Company, Incorporated, 2nd edition, 2011.

CGRS23. Hien Chu, Paul Gerhart, Tim Ruffing, and Dominique Schröder. Practical schnorr threshold signatures without the algebraic group model. In *Annual International Cryptology Conference*, pages 743–773. Springer, 2023.

Che06. Jung Hee Cheon. Security analysis of the strong diffie-hellman problem. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 1–11. Springer, 2006.

CHI$^+$21. Megan Chen, Carmit Hazay, Yuval Ishai, Yuriy Kashnikov, Daniele Micciancio, Tarik Riviere, Abhi Shelat, Muthu Venkitasubramaniam, and Ruihan Wang. Diogenes: lightweight scalable rsa modulus generation with a dishonest majority. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 590–607. IEEE, 2021.

CKM21. Elizabeth Crites, Chelsea Komlo, and Mary Maller. How to prove schnorr assuming schnorr: Security of multi-and threshold signatures. *Cryptology ePrint Archive*, 2021.

CL01. Jan Camenisch and Anna Lysyanskaya. An efficient system for non-transferable anonymous credentials with optional anonymity revocation. In *Advances in Cryptology—EUROCRYPT 2001: International Conference on the Theory and Application of Cryptographic Techniques Innsbruck, Austria, May 6–10, 2001 Proceedings 20*, pages 93–118. Springer, 2001.

CL15. Guilhem Castagnos and Fabien Laguillaumie. Linearly homomorphic encryption from. In *Cryptographers' Track at the RSA Conference*, pages 487–505. Springer, 2015.

CL24. Yi-Hsiu Chen and Yehuda Lindell. Optimizing and implementing fischlin's transform for uc-secure zero-knowledge. *Cryptology ePrint Archive*, 2024.

CLT18. Guilhem Castagnos, Fabien Laguillaumie, and Ida Tucker. Practical fully secure unrestricted inner product functional encryption modulo p. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 733–764. Springer, 2018.

CMP20. Ran Canetti, Nikolaos Makriyannis, and Udi Peled. UC non-interactive, proactive, threshold ECDSA. *IACR Cryptol. ePrint Arch.*, page 492, 2020.

CNS23. Anamaria Costache, Lea Nürnberger, and Tjerand Silde. Computational fhe circuit privacy for free. *Cryptology ePrint Archive*, 2023.

Cop21. Copper. Wallets as a service. *Whitepaper https://copper.co/en/products/wallets-as-a-service*, 2021.

Cry13. Nicolas van S Cryptonote. v2. 0. *HYPERLINK https://cryptonote.org/whitepaper.pdf*, 2013.

Dev23. Bitcoin Developer. Developer guides – locktime and sequence number. *Bitcoin Developer https://developer.bitcoin.org/devguide/transactions.html*, 2023.

DJN$^+$20. Ivan Damgård, Thomas Pelle Jakobsen, Jesper Buus Nielsen, Jakob Illeborg Pagter, and Michael Bæksvang Østergård. Fast threshold ECDSA with honest majority. In *SCN*, volume 12238 of *LNCS*, pages 382–400. Springer, 2020.

DKKSS22. George Danezis, Lefteris Kokoris-Kogias, Alberto Sonnino, and Alexander Spiegelman. Narwhal and tusk: a dag-based mempool and efficient bft consensus. In *Proceedings of the Seventeenth European Conference on Computer Systems*, pages 34–50, 2022.

DKLS19. Jack Doerner, Yashvanth Kondi, Eysa Lee, and Abhi Shelat. Threshold ECDSA from ECDSA assumptions: The multiparty case. In *SP 2019*, pages 1051–1066. IEEE, 2019.

DKLS23. Jack Doerner, Yashvanth Kondi, Eysa Lee, and Abhi Shelat. Threshold ECDSA in three rounds. *IACR Cryptol. ePrint Arch.*, page 765, 2023.

DLS88. Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *J. ACM*, 35(2):288–323, apr 1988.

DOK$^+$20. Anders P. K. Dalskov, Claudio Orlandi, Marcel Keller, Kris Shrishak, and Haya Shulman. Securing DNSSEC keys via threshold ECDSA from generic MPC. In *ESORICS*, volume 12309, pages 654–673. Springer, 2020.

DPTX24. Sourav Das, Benny Pinkas, Alin Tomescu, and Zhuolun Xiang. Distributed randomness using weighted vrfs. *Cryptology ePrint Archive*, 2024.

DT06. Ivan Damgård and Rune Thorbek. Linear integer secret sharing and distributed exponentiation. In *International Workshop on Public Key Cryptography*, pages 75–90. Springer, 2006.

EKR20. Sinan Ergezer, Holger Kinkelin, and Filip Rezabek. A survey on threshold signature schemes. *Network*, 49, 2020.

Eth24. Etherscan. Ethereum node tracker. *Etherscan https://etherscan.io/nodetracker*, 2024.

Fir23. 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.

Fis05. Marc Fischlin. Communication-efficient non-interactive proofs of knowledge with online extractors. In *Annual International Cryptology Conference*, pages 152–168. Springer, 2005.

FLP85. Michael J Fischer, Nancy A Lynch, and Michael S Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM (JACM)*, 32(2):374–382, 1985.

FMM$^+$23. Offir Friedman, Avichai Marmor, Dolev Mutzari, Yehonatan C Scaly, Yuval Spiizer, and Avishay Yanai. Tiresias: Large scale, maliciously secure threshold paillier. *Cryptology ePrint Archive*, 2023.

FMM$^+$24. Offir Friedman, Avichai Marmor, Dolev Mutzari, Omer Sadika, Yehonatan C Scaly, Yuval Spiizer, and Avishay Yanai. 2pc-mpc: Emulating two party ecdsa in large-scale mpc. *Cryptology ePrint Archive*, 2024.

FS86. Amos Fiat and Adi Shamir. How to prove yourself: Practical solutions to identification and signature problems. In *Conference on the theory and application of cryptographic techniques*, pages 186–194. Springer, 1986.

GG19. Rosario Gennaro and Steven Goldfeder. Fast multiparty threshold ECDSA with fast trustless setup. *IACR Cryptol. ePrint Arch.*, page 114, 2019.

GGN16. Rosario Gennaro, Steven Goldfeder, and Arvind Narayanan. Threshold-optimal DSA/ECDSA signatures and an application to bitcoin wallet security. In *International Conference on Applied Cryptography and Network Security*, pages 156–174. Springer, 2016.

GHL22. Craig Gentry, Shai Halevi, and Vadim Lyubashevsky. Practical non-interactive publicly verifiable secret sharing with thousands of parties. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 458–487. Springer, 2022.

GKSS20. Adam Gagol, Jedrzej Kula, Damian Straszak, and Michal Swietek. Threshold ECDSA for decentralized asset custody. *IACR Cryptol. ePrint Arch.*, page 498, 2020.

Gra22. Evan Gray. Governor. *GitHub Repository https://github.com/wormhole-foundation/wormhole/blob/main/whitepapers/0007_governor.md*, 2022.

GS22. Jens Groth and Victor Shoup. On the security of ecdsa with additive key derivation and presignatures. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 365–396. Springer, 2022.

GZT$^+$19. Zhengfeng Gao, Jilai Zheng, Shuyang Tang, Yu Long, Zhiqiang Liu, Zhen Liu, and Dawu Gu. State-of-the-art survey of consensus mechanisms on dag-based distributed ledger. *Journal of Software*, 31(4):1124–1142, 2019.

Har23. Simon Harman. Chainflip protocol whitepaper. *Whitepaper https://chainflip.io/whitepaper.pdf*, 2023.

Hos17. Charles Hoskinson. Why we are building cardano. *IOHK (accessed 18 December 2017) https://whycardano.com*, 2017.

HZL$^+$21. Bin Hu, Zongyang Zhang, Jianwei Liu, Yizhong Liu, Jiayuan Yin, Rongxing Lu, and Xiaodong Lin. A comprehensive survey on smart contract construction and execution: paradigms, tools, and systems. *Patterns*, 2(2), 2021.

IA21. Polosukhin Illia and Skidanov Alexander. The near white paper. *Whitepaper https://pages.near.org/papers/the-official-near-white-paper/*, 2021.

JMV01. Don Johnson, Alfred Menezes, and Scott Vanstone. The elliptic curve digital signature algorithm (ecdsa). *International journal of information security*, 1:36–63, 2001.

KAS19. Nida Khan, Tabrez Ahmad, and Radu State. Feasibility of stellar as a blockchain-based micropayment system. In *International Conference on Smart Blockchain*, pages 53–65. Springer, 2019.

KG21. Chelsea Komlo and Ian Goldberg. 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*, pages 34–65. Springer, 2021.

Klu22. Kamil Kluczniak. Circuit privacy for fhew/tfhe-style fully homomorphic encryption in practice. *Cryptology ePrint Archive*, 2022.

KZe19a. Team KZen. Bitcoin wallet powered by two-party ecdsa - extended abstract. *Whitepaper https://github.com/ZenGo-X/gotham-city/blob/master/white-paper/*, 2019.

KZe19b. Team KZen. Radical new infrastructure for digital asset ownership and blockchain interoperability. *Whitepaper https://www.qredo.com/qredo-white-paper.pdf*, 2019.

Lab24. Torus Labs. Web3 auth. *Github Repository https://github.com/web3auth*, 2024.

LeM17. Colin LeMahieu. Raiblocks: A feeless distributed cryptocurrency network. *URL https://raiblocks. net/media/RaiBlocks_Whitepaper__English. pdf*, 2017.

Lin17. Yehuda Lindell. Fast secure two-party ECDSA signing. In *Annual International Cryptology Conference*, pages 613–644. Springer, 2017.

Lin22. Yehuda Lindell. Simple three-round multiparty schnorr signing with full simulatability. *Cryptology eprint Archive*, 2022.

Lin23. Yehuda Lindell. Digital asset management with mpc. *Whitepaper https://www.coinbase.com/blog/digital-asset-management-with-mpc-whitepaper*, 2023.

LMDG23. Sung-Shine Lee, Alexandr Murashkin, Martin Derka, and Jan Gorzny. Sok: Not quite water under the bridge: Review of cross-chain bridge hacks. In *2023 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*, pages 1–14. IEEE, 2023.

LN18. Yehuda Lindell and Ariel Nof. Fast secure multiparty ECDSA with practical distributed key generation and applications to cryptocurrency custody. In *CCS*, pages 1837–1854. ACM, 2018.

LP18. Matt Luongo and Corbin Pon. The keep network: A privacy layer for public blockchains. *Keep Netw., Rep*, 2018.

LRP20. Stefanos Leonardos, Daniël Reijsbergen, and Georgios Piliouras. Weighted voting on the blockchain: Improving consensus in proof of stake protocols. *International Journal of Network Management*, 30(5):e2093, 2020.

LRU22. Christophe Levrat, Matthieu Rambaud, and Antoine Urban. Breaking the $t < n/3$ consensus bound: Asynchronous dynamic proactive secret sharing under honest majority. *Cryptology ePrint Archive*, 2022.

MBH23. Christian Mouchet, Elliott Bertrand, and Jean-Pierre Hubaux. An efficient threshold access-structure for rlwe-based multiparty homomorphic encryption. *Journal of Cryptology*, 36(2):10, 2023.

MPJ18. Bhabendu Kumar Mohanta, Soumyashree S Panda, and Debasish Jena. An overview of smart contract and use cases in blockchain technology. In *2018 9th international conference on computing, communication and networking technologies (ICCCNT)*, pages 1–4. IEEE, 2018.

Nak08. Satoshi Nakamoto. Bitcoin whitepaper. *URL: https://bitcoin.org/bitcoin.pdf-(:17.07.2019)*, 9:15, 2008.

NRS21. Jonas Nick, Tim Ruffing, and Yannick Seurin. Musig2: Simple two-round schnorr multi-signatures. In *Annual International Cryptology Conference*, pages 189–221. Springer, 2021.

Pai99. Pascal Paillier. 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*, pages 223–238. Springer, 1999.

Ped91. Torben Pryds Pedersen. Non-interactive and information-theoretic secure verifiable secret sharing. In *Annual international cryptology conference*, pages 129–140. Springer, 1991.

PMIH18. Huma Pervez, Muhammad Muneeb, Muhammad Usama Irfan, and Irfan Ul Haq. A comparative analysis of dag-based blockchain architectures. In *2018 12th International conference on open source systems and technologies (ICOSST)*, pages 27–34. IEEE, 2018.

Rab98. Tal Rabin. A simplified approach to threshold and proactive rsa. In *Advances in Cryptology—CRYPTO'98: 18th Annual International Cryptology Conference Santa Barbara, California, USA August 23–27, 1998 Proceedings 18*, pages 89–104. Springer, 1998.

RRJ$^+$22. Tim Ruffing, Viktoria Ronge, Elliott Jin, Jonas Schneider-Bensch, and Dominique Schröder. Roast: robust asynchronous schnorr threshold signatures. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, pages 2551–2564, 2022.

SBK24. István András Seres, Péter Burcsi, and Péter Kutas. How (not) to hash into class groups of imaginary quadratic fields? *Cryptology ePrint Archive*, 2024.

Sch90. Claus-Peter Schnorr. Efficient identification and signatures for smart cards. In *Advances in Cryptology—CRYPTO'89 Proceedings 9*, pages 239–252. Springer, 1990.

Sch99. Berry Schoenmakers. A simple publicly verifiable secret sharing scheme and its application to electronic voting. In *Annual International Cryptology Conference*, pages 148–164. Springer, 1999.

SG20. Helder Sebastião and Pedro Godinho. Bitcoin futures: An effective tool for hedging cryptocurrencies. *Finance Research Letters*, 33:101230, 2020.

SGSKK22. Alexander Spiegelman, Neil Giridharan, Alberto Sonnino, and Lefteris Kokoris-Kogias. Bullshark: Dag bft protocols made practical. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, pages 2705–2718, 2022.

Sha79a. Adi Shamir. How to share a secret. *Communications of the ACM*, 22(11):612–613, 1979.

Sha79b. Adi Shamir. How to share a secret. *Communications of the ACM*, 22(11):612–613, 1979.

Sho24. Victor Shoup. A theoretical take on a practical consensus protocol. *Cryptology ePrint Archive*, 2024.

SS24. Victor Shoup and Nigel P Smart. Lightweight asynchronous verifiable secret sharing with optimal resilience. *Journal of Cryptology*, 37(3):27, 2024.

Sta96. Markus Stadler. Publicly verifiable secret sharing. In *International Conference on the Theory and Applications of Cryptographic Techniques*, pages 190–199. Springer, 1996.

Sta20. V Stafford. Zero trust architecture. *NIST special publication*, 800:207, 2020.

T^+22. DFINITY Team et al. The internet computer for geeks. *Cryptology ePrint Archive*, 2022.

Tho20. Thorchain. Thorchain: A decentralised liquidity network. *Whitepaper https://github.com/thorchain/Resources/blob/master/Whitepapers/THORChain-Whitepaper-May2020.pdf*, 2020.

TLL^+21. Palina Tolmach, Yi Li, Shang-Wei Lin, Yang Liu, and Zengxiang Li. A survey of smart contract formal specification and verification. *ACM Computing Surveys (CSUR)*, 54(7):1–38, 2021.

WGD22. Francesc Wilhelmi, Lorenza Giupponi, and Paolo Dini. Analysis and evaluation of synchronous and asynchronous flchain. *Computer Networks*, 218:109390, 2022.

Wil22. MacLane Wilkison. What is threshold network? *Whitepaper https://docs.threshold.network/*, 2022.

WMYC23. Harry W. H. Wong, Jack P. K. Ma, Hoover H. F. Yin, and Sherman S. M. Chow. Real threshold ECDSA. In *NDSS*. The Internet Society, 2023.

Wor22. Wormhole. Wormhole incident report — 02/02/22. *Wormhole-crypto Medium https://wormholecrypto.medium.com/wormhole-incident-report-02-02-22-ad9b8f21eec6*, 2022.

Wui13. P. Wuille. Hierarchical deterministic wallets. *GitHub Repository https://github.com/bitcoin/bips/blob/master/bip-0032.mediawiki*, 2013.

WYCX23. Qin Wang, Jiangshan Yu, Shiping Chen, and Yang Xiang. Sok: Dag-based blockchain systems. *ACM Computing Surveys*, 55(12):1–38, 2023.

WYW^+18. Shuai Wang, Yong Yuan, Xiao Wang, Juanjuan Li, Rui Qin, and Fei-Yue Wang. An overview of smart contract: architecture, applications, and future trends. In *2018 IEEE Intelligent Vehicles Symposium (IV)*, pages 108–113. IEEE, 2018.

WYY^+22. Huan Yu Wu, Xin Yang, Chentao Yue, Hye-Young Paik, and Salil S Kanhere. Chain or dag? underlying data structures, architectures, topologies and consensus in distributed ledger technology: A review, taxonomy and research issues. *Journal of Systems Architecture*, 131:102720, 2022.

XAX^+22. Haiyang Xue, Man Ho Au, Xiang Xie, Tsz Hon Yuen, and Handong Cui. Efficient online-friendly two-party ECDSA signature. *IACR Cryptol. ePrint Arch.*, page 318, 2022.

Yak18. Anatoly Yakovenko. Solana: A new architecture for a high performance blockchain v0. 8.13. *Whitepaper*, 2018.

Yan23. J. Wiley Yandle. Weighted schnorr threshold signatures. *GitHub Repository https://trust-machines.github.io/wsts/wsts.pdf*, 2023.

YXXM23. Thomas Yurek, Zhuolun Xiang, Yu Xia, and Andrew Miller. Long live the honey badger: Robust asynchronous {DPSS} and its applications. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 5413–5430, 2023.

ZYP23. Guy Zyskind, Avishay Yanai, and Alex 'Sandy' Pentland. Unstoppable wallets: Chain-assisted threshold ECDSA and its applications. *IACR Cryptol. ePrint Arch.*, page 832, 2023.

A Extended Preliminaries

A.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 4).

FUNCTIONALITY A.1 (*Internal Broadcast Functionality: $\mathcal{F}_{\mathsf{broadcast}}$ from [SS24]*)

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 A.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 A.3 (*Asynchronous Common Subset Functionality: $\mathcal{F}^{\Gamma_B}_{\mathsf{ACS}}$ from [Sho24]*)

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.

A.2 Security Definitions for ZKPs

Definition A.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 Zer-Knowledge Proof of Knowledge if it is also* Knowledge sound*.*

Definition A.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 A.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-Knowlege (HVZK)

Definition A.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 A.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-negligble 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* $\mathcal{P}^*$*.*

A.3 Statement Aggregation Protocol

We present an aggregation protocol for brevity, which will be referred to later in our protocols. This is a simple protocol consists of sending a statement with a proof, agreeing on a subset in the access structure that has sent valid proofs and calculating the aggregated statement. In the case of voting power there is no need to send individual statements, each party can create, send and prove a single statement and agreeing on a subset with enough voting power.

PROTOCOL A.4 (*Aggregation Protocol:* $\Pi_{\mathsf{agg}}^{\Gamma_B, L}$)

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 $(\vec{Y}_i; \vec{w}_i) \in L$.

- **Each** B_i **does as follows:**
  1. Run a ZK protocol $\Pi_{\mathsf{zk}}^{L}(\vec{Y}_i; \vec{w}_i)$ generating a proof π_i. Send $(\mathsf{broadcast}, \mathsf{sid}, i, (\mathsf{proof}, \vec{Y}_i, \pi_i))$ to $\mathcal{F}_{\mathsf{broadcast}}$.
  2. Upon receiving $(\mathsf{broadcast}, \mathsf{sid}, (\mathsf{proof}, \vec{Y}_j, \pi_j))$ 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 $\sum_{j \in S_B} \vec{Y}_j$.

A.4 Security Definitions For Shamir Secret Sharing Over The Integers

The security definition below is taken from [DT06] (Definition 4).

Definition A.6. ***Privacy of a Secret Sharing Scheme****: A Linear Secret Sharing scheme is private, if for any two secrets* s, s'*, independent random coins* r, r' *and any set* U *with at most* t *shareholders, the distribution of* $\{s_i(s, r, k)\}_{i \in U}$ *and* $\{s_i(s', r, k)\}_{i \in U}$ *are statistically indistinguishable.*

Definition A.7. ***Correctness of a Secret Sharing Scheme****: A Linear Secret Sharing scheme is correct, if the secret is reconsructed from shares* $\{s_i\}_{i \in S}$ *where* $S \in \binom{[n]}{t+1}$*, by taking an integer linear combination of the shares, with coefficients that depend only on the index set* S*.*

Importantly note that in the correctness definition for Integers Secret Sharing the secrets are $\Delta_n^2 s$ instead of s.

A.5 Homomorphic Commitment Schemes

Definition A.8. *A non-interactive commitment scheme consists of a pair of PPT algorithms* $(\mathsf{setup}, \mathsf{Com})$*. The setup algoirthm* $\mathsf{pp} \leftarrow \mathsf{setup}$ *outputs the spaces* $\mathcal{M}_{\mathsf{pp}}$*,* $\mathcal{R}_{\mathsf{pp}}$ *and* $\mathcal{C}_{\mathsf{pp}}$ *along with any other public values needed for the scheme. The commitment algorithm* $\mathsf{Com}_{\mathsf{pp}}$ *defines a function* $\mathcal{M}_{\mathsf{pp}} \times \mathcal{R}_{\mathsf{pp}} \to \mathcal{C}_{\mathsf{pp}}$*. For a message* $m \in \mathcal{M}_{\mathsf{pp}}$*, the algirthm draws* ρ*, and computes commitment* $C = \mathsf{Com}_{\mathsf{pp}}(m; \rho)$*. Whenever the public parameters are clear from the contexst we write* Com *instead of* $\mathsf{Com}_{\mathsf{pp}}$*. Commitment schemes may also be deterministic in which case one can think of* ρ *as a trivial group and write* $\mathsf{Com}_{\mathsf{pp}}(m)$*.*

A commitment scheme has two properties, *hiding* and *binding*, in the case of Pedersen Commitment [Ped91] 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 $\mathsf{pp} \leftarrow \mathsf{Setup}(\cdot)$,

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

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

A.6 Formal Definitions For AHE and TAHE

AHE: The definition for AHE below is inspired from [CHI+21]

Definition A.1 (AHE) *An* additively homomorphic encryption scheme *is associated with an ensemble* $\{\mathcal{K}_\kappa\}_\kappa$ *and consists of four polynomial time algorithms:* $\mathit{AHE} = (\mathit{Gen}, \mathit{Enc}, \mathit{Dec}, \mathit{Add})$ *specified as follows:*

– $\mathit{Gen}(1^\kappa, \mathsf{aux}) \to (\mathit{pk}; \mathit{sk})$*. A probabilistic algorithm that is given a security parameter* 1^κ *and possibly some auxiliary information* aux *and samples a key-pair* $(\mathit{pk}, \mathit{sk})$ *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 purpose in this paper we require an AHE scheme to be associated with a language $L_{\mathsf{GenAHE}}[\kappa] = \{pk; \mathsf{aux} \mid (pk; sk) = Gen(1^\kappa, \mathsf{aux})\}$ *(see Eq. 13) that expresses the fact that* pk *was generated correctly.*

- $Enc(pk, pt; \eta_{\mathsf{enc}}) \to ct$*. A deterministic algorithm that on input a public key* pk*, a plaintext* $pt \in \mathcal{P}_{pk}$ *and randomness* $\eta_{\mathsf{enc}} \in \mathcal{R}_{pk}$*, outputs a ciphertext* $ct \in \mathcal{C}_{pk}$*. We define* $Enc(pk, pt)$ *as a probabilistic algorithm that first uniformly samples* $\eta_{\mathsf{enc}} \in \mathcal{R}_{pk}$ *and then outputs* $ct = Enc(pk, pt; \eta_{\mathsf{enc}}) \in \mathcal{C}_{pk}$*.*
- $Dec(sk, ct) \to pt$*. A deterministic algorithm that on input a secret key* sk *and a ciphertext* $ct \in \mathcal{C}_{pk}$ *outputs a plaintext* $pt \in \mathcal{P}_{pk}$*.*
- $Add(pk, ct_1, ct_2) \to ct_3$*. A deterministic algorithm that on input a public key* pk *and two ciphertexts* $ct_1, ct_2 \in \mathcal{C}_{pk}$ *outputs a ciphertext* $ct_3 \in \mathcal{C}_{pk}$ *such that if* $ct_1 = Enc(pk, m_1; \eta_1)$ *and* $ct_2 = Enc(pk, m_2; \eta_2)$ *then* $ct_3 = Enc(pk, m_1 + m_1; \eta_1 + \eta_2)$*, 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 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 $0 \leq i \leq \ell$) and ℓ ciphertexts $\mathsf{ct}_1, \ldots, \mathsf{ct}_\ell \in \mathcal{R}_{pk}$, Eval outputs a ciphertext ct.

Definition A.2 (Correctness) *AHE is* correct *if for every* κ*, every* $t \in poly(\kappa)$*, every* ℓ*-ary affine function* f *as above, and every plaintexts* $pt_1, \ldots, pt_t \in \mathcal{P}_{pk}$*,*

$$\Pr[Dec(sk, Eval(pk, f, (ct_1, \ldots, ct_\ell))) = f(pt_1, \ldots, pt_\ell)] \geq 1 - neg(\kappa)$$

where $(pk, sk) \leftarrow Gen(1^\kappa)$ *and* $ct_i \leftarrow Enc(pk, pt_i)$*, 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 A.3 (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* $pt \in \mathcal{P}_{pk}$*:*

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

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

Definition A.4 (Linear Circuit Privacy) *AHE has* Linear Circuit Privacy *if there exists a PPT algorithm $\mathcal{S}_{\mathit{Eval}}$ such that for every PPT adversary $\mathcal{A}$, every ℓ-ary affine function f as above, every (pk, sk) and every plaintexts $pt_1, \ldots, pt_\ell \in \mathcal{P}_{pk} \cup \{\perp\}$ and randomness $\eta_1, \ldots, \eta_\ell$,*

$$
\begin{aligned}
| \Pr[\mathcal{A}(1^\kappa, pk, sk, \{pt_i, \eta_i\}_{i=1}^{\ell}, \mathit{Eval}(pk, f, (ct_1, \ldots, ct_\ell))) = 1] \\
- \Pr[\mathcal{A}(1^\kappa, pk, sk, \{pt_i, \eta_i\}_{i=1}^{\ell}, \mathcal{S}_{\mathit{Eval}}(pk, f(pt_1, \ldots, pt_\ell))) = 1]| \leq \mathit{neg}(\kappa)
\end{aligned}
\tag{2}
$$

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

Remark A.1. We build upon the definition from [FMM+24] 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 [Pai99] this distinction is void. But it is not the case for Elgamal style encryption such as Class-Group [CL15] nor for lattice based AHE. See Appendix F for further discussion.

TAHE: A TAHE scheme is defined by a tuple of algorithms (Gen, Enc, Add, TDec, Rec). Gen, Enc, Add works similarly to the definition of AHE while TDec is an operation that each party runs locally on the chiphertext 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 A.5 and the AHE security definition.

Definition A.5 (TAHE) *Threshold additively homomorphic encryption scheme is associated with an ensemble $\{\mathcal{K}_\kappa\}_\kappa$ and consists of five polynomial time algorithms TAHE = (Gen, Enc, Add, TDec, Rec) specified as follows:*

– $(pk, vk_i; [sk]_i) \leftarrow \mathit{Gen}(1^\kappa, i, t, n, \mathsf{aux})$. *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 $(pk, sk) \in \mathcal{K}_\kappa$ such that $[sk]_i$ 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* $\{\mathsf{ds}_i\}_{i \in S} \in \mathcal{C}^{|S|}_{pk}$ *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.

Dynamical TAHE: In our use-case the parties need to be able to re-share their secret keys such that the distribution of the secret keys represent their current stake in the blockchain. To this end in addition to the above algorithms we add a reconfiguration protocol.
$\mathsf{ReConf}(1^\kappa, 1^\sigma, t, t', n, n', \mathsf{pk}, \mathsf{aux}; [\mathsf{sk}]_i)$ a probabilistic (possibly interactive) protocol that is given a security parameter 1^κ, a statistical security parameter 1^σ, the initial threshold t, the final threshold t', the initial number of parties n, possibly some auxiliary information aux and the output $(\mathsf{AHE}, [\mathsf{sk}]_i)$ of the Gen protocol. We will say that a dynamical TAHE is secure if it realizes the below ideal functionality A.5 $\mathcal{F}_{DAHE}$

A.7 PVSS Formal Definition

Definition A.6 *(PVSS [CD24a]) A PVSS scheme consists of the following algorithms:*

- *Setup:*
  - $\mathsf{pp} \leftarrow \mathsf{Setup}(1^\kappa, \mathsf{ip})$ *outputs public parameters* pp. *The initial parameters* ip *contain information about the number of parties, privacy and Reconstruction thresholds and spaces of secrets and shares. The public parameters include a description of the key space* $\mathcal{K}_\kappa$ *which should be thought of as a Cartesian product over the public key share and the private key share along with some relation they satisfy.*
  - $(\mathsf{pk}_i, \pi^i_{\mathsf{pk}}; \mathsf{sk}_i) \leftarrow \mathsf{Gen}(\mathsf{pp}, i)$, *where* $(\mathsf{pk}; \mathsf{sk}) \in \mathcal{K}_\kappa$ *and* π_{pk} *is a proof meant to assert that* pk *is a valid public key.*
  - $0/1 \leftarrow \mathsf{Verify}(\mathsf{pp}, j, \mathsf{pk}_j, \pi^j_{\mathsf{pk}})$, *a verification algorithm for the proof* Π^j_{pk}.

FUNCTIONALITY A.5 (*Dynamical Access Structure TAHE:* $\mathcal{F}_{\mathsf{DAHE}}$)

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 initial access structure Γ_{keygen}.

1. **Key Generation**: Upon receiving $(\mathsf{keygen}, \mathsf{pid}, \mathsf{sid})$ from a subset $S_{\mathsf{keygen}} \in \Gamma_{\mathsf{keygen}}$ 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, define $\Gamma = \Gamma_{\mathsf{keygen}}$ and store $(\mathsf{access}-\mathsf{structure}, \mathsf{sid}, \Gamma)$ to memory, otherwise send $(\mathsf{pubkey}, \mathsf{sid}, \perp, U \cap U')$ to all parties in B.
2. **Decryption:** Upon receiving a $(\mathsf{decrypt}, \mathsf{pid}, \mathsf{sid}, \mathsf{ct})$ let $S_{\mathsf{decrypt,sid,ct}} = S_{\mathsf{decrypt,sid,ct}} \cup \{\mathsf{pid}\}$ (if no request were recorded yet treat $S_{\mathsf{decrypt,sid,ct}}$ 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 $S_{\mathsf{decrypt,sid,ct}} \cup U \in \Gamma$ for $(\mathsf{access}-\mathsf{structure}, \mathsf{sid}, \Gamma)$ in memory, if true send $(\mathsf{decrypted}, \mathsf{sid}, \mathsf{ct}, \mathsf{pt})$ to U.
   (c) Check if $S_{\mathsf{decrypt,sid,ct}} \in \Gamma$ for $(\mathsf{access}-\mathsf{structure}, \mathsf{sid}, \Gamma)$, 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 $(\mathsf{decrypted}, \mathsf{pk}, \perp, U \cap U'')$.
3. **Re-configuration**: Upon receiving $(\mathsf{re}-\mathsf{configure}, \mathsf{pid}, \mathsf{sid}, \Gamma')$ from a subset $S \in \Gamma$ delete $(\mathsf{re}-\mathsf{configure}, \mathsf{sid}, \Gamma)$ and store $(\mathsf{re}-\mathsf{configure}, \mathsf{sid}, \Gamma')$ instead. Further more delete any message titled $\mathsf{decrypt}$ or $\mathsf{decrypted}$ along with the sets $S_{\mathsf{decrypt,sid,ct}}$.

– *Distribution:* $(\{\mathsf{ct}_i\}_{i\in[n]}, \pi_{\mathsf{Share}}) \leftarrow \mathsf{Dist}(\mathsf{pp}, \{\mathsf{pk}\}_{i\in[n]}, s)$*, where* $s \in \mathcal{S}$ *is a valid secret, outputs "encrypted shares"* ct_i *and a proof that they are correct* π_{Share}.
– *Distribution Verification:* $0/1 \leftarrow \mathsf{VerifySharing}(\mathsf{pp}, \{\mathsf{pk}_i, \mathsf{ct}_i\}_{i\in[n], \pi_{\mathsf{Share}}}$*, a verification algorithm for* π_{Share}.
– *Reconstruction:*
  - $([s]_i, \pi^i_{\mathsf{Dec}}) \leftarrow \mathsf{DecShare}(\mathsf{pp}, \mathsf{pk}_i, \mathsf{sk}_i, \mathsf{ct}_i)$*, outputs a decrypted share* $[s]_i$ *and a proof* π^i_{Dec} *of correct decryption.*
  - $(s') \leftarrow \mathsf{Rec}(\mathsf{pp}, \{[s]_i\}_{i\in S})$ *either returns* abort *if* $S \notin \Gamma$ *or an element of the secret space* $s' \in \mathcal{S}$ *if* $S \in \Gamma$.
– *Reconstruction Verification:* $0/1 \leftarrow \mathsf{VerifyDec}(\mathsf{pp}, \mathsf{pk}_i, \mathsf{ct}_i, [s]_i, \pi^i_{\mathsf{Dec}})$*, A verification algorith for the proof* π^i_{Dec}.

We refer the reader to [Sch99, CD24a, GHL22] for further details about the game based security definitions for a PVSS scheme.

B Languages used in our protocols.

The languages in our protocols are listed below:

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

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

- *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.*

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

- *Vector Commitment of discrete log.*

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

- *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.*

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

- *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 A.6) 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.

C Full Detailed Protocols

C.1 ECDSA Protocols

Next, we provide the full details of the key generation (Protocol C.1), presign (Protocol C.2), and sign (Protocol C.3) protocols for ECDSA and prove their correctness.

Correctness of the key generation protocol: The only correctness demand on the key generation is that the parties agree on X and that $\mathsf{ct}_{\mathsf{key}}$ is an encryption of the discrete log of X. From the consistency of the broadcast A_{pid_A} receives $(\mathsf{pid}_A, X_{0,B}, X_{1,B})$ which is the same as received by each party in B. From the uniqueness of pid_A and from the consistency of the broadcast functionality we get that the parties in B gets the same X_A. Thus the parties agree on the inputs to the random oracle which derive $\mu_x^0, \mu_x^1, \mu_x^G$ which in turn uniquely determine $X = X_A + X_B$. The correctness of $\mathsf{ct}_{\mathsf{key}}$ is inherited from the correctness of the aggregation protocol.

Correctness of the presign protocol: In this protocol correctness is defined by getting an encryption of a value γ along with an encryption of $\gamma \cdot x$ and values $R_{B,0}, R_{B,1}$ along with an encryption of their discrete logs. The correctness follows simply from the correctness of the aggregation protocol and from the consistency of the broadcast functionality.

Correctness of the sign protocol: The correctness of the sign protocol is more complex so we are going to go through it in a bit more details. Let's see that the proof π_k, π_α will pass. We have $C_k = k_A \cdot G + \rho_0 H$ thus $G = k_A^{-1} C_k - k_A^{-1} \rho_0 H$ which is exactly of the needed form. Similarly for π_α. From the correctness of the pre-sign we have that $\mathsf{ct}_{\gamma \cdot k} = \mathsf{Enc}_{\mathsf{pk}}(\gamma(\mu_k^0 \cdot k_0 + \mu_k^1 \cdot k_1 + \mu_k^G)))$ and that $R'_B = (\mu_k^0 \cdot k_0 + \mu_k^1 \cdot k_1 + \mu_k^G) \cdot G$. We denote by $k_B = \alpha(\mu_k^0 \cdot k_0 + \mu_k^1 \cdot k_1 + \mu_k^G) + \beta, k = k_A^{-1} \cdot k_B$ then $R'_B = k_B \cdot G, R = k \cdot G$. From this we have that

1. $\mathsf{ct}_{\alpha,\beta} = \mathsf{Enc}_{\mathsf{pk}}(\gamma \cdot k_B)$
2. $\mathsf{ct}_A = \mathsf{Enc}_{\mathsf{pk}}(\gamma(rk_A x_A + mk_A + x_B r k_A)) = \mathsf{Enc}_{\mathsf{pk}}(\gamma k_A(m + rx))$.
3. $C_1 = \mathsf{Com}_{\mathsf{pp}}(a_1)$
4. $C_2 = \mathsf{Com}_{\mathsf{pp}}(a_2)$

The above observation helps us to show that the following proofs are correct. Since the random oracle is consistent and receives the same values it returns μ_k to B and the calculation of the values in commands $i. - iv.$ is the same as was done by A_{pid_A}. Then after decryption we get $s' = \gamma^{-1} \cdot k_B^{-1} k_A \gamma(r + mx) = k^{-1}(r + mx)$ which is exactly the form of a valid signature with $R = k \cdot G$.

C.2 Schnorr-Based Protocols

Next, we provide the full details and prove the correctness of the presign (Protocol C.4) and sign (Protocol C.5) protocols for Schnorr-based signatures. The correctness of the key generation and pre-sign protocols is similar to the ECDSA case.

Correctness of the sign protocol. We denote $k_B = k_{B,0} + \mu_k \cdot k_{B,1}$ and $k = k_A + k_B$ then $K = kG$. We have that $z_A G = k_A G + ek_A G = K_A + eK_A$. $\mathsf{ct}_B = \mathsf{Enc}_{\mathsf{pk}}(k_B + ex_B + k_A + ex_A) = \mathsf{Enc}_{\mathsf{pk}}(k + ex)$. Thus $\mathsf{pt}_B = k + ex$ which is a valid signature with respect to K.

PROTOCOL C.1 (*Key-Generation* Π_{keygen}: $\mathsf{KeyGen}(\mathbb{G}, G, q, \Gamma_B, \mathit{sid}, \mathit{pk})$)

The protocol is parameterized with the ECDSA group description $(\mathbb{G}, G, q)$ along with an access structure Γ_B, a unique session id sid, and a 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 $\mathcal{F}_{\mathsf{global\text{-}broadcast}}$. The parties output the public verification key X and record the following:

- X_A - The centralized party's share of a public verification key for an ECDSA signature (implicitly gives the distributed party share).
- The distributed party B also records $\mathsf{ct}_{\mathsf{key}}$ an encryption of the secret share held by the distributed party B
- The centralized party A_{pid_A} also records x_A - its secret share of an ECDSA signature key.

The parties do as follows:

1. **Each B_i does as follows**:
   - **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 $X_0^i = x_0^i \cdot G, X_1^i = x_1^i \cdot G$ 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 on $(X_1^i, \mathsf{ct}_1^i; x_1^i, \eta_1^i)$ receiving $(X_{0,B}, X_{1,B}, \mathsf{ct}_{0,\mathsf{key}}, \mathsf{ct}_{1,\mathsf{key}})$.
   - **Broadcast Round**:
   (a) Send $(\mathsf{global\text{-}broadcast}, \mathsf{sid}, i, (\mathsf{pid}_A, X_{0,B}, X_{1,B}))$ to $\mathcal{F}_{\mathsf{global\text{-}broadcast}}^{\Gamma_B}$.
2. **A_{pid_A} does as follows**:
   (a) Receive $(\mathsf{global\text{-}broadcast}, \mathsf{sid}, (\mathsf{pid}_A, X_{0,B}, X_{1,B}))$ from $\mathcal{F}_{\mathsf{global\text{-}broadcast}}^{\Gamma_B}$.
   (b) Verify that $X_{0,B}, X_{1,B} \in \mathbb{G}$ else consider B malicious and abort.
   (c) Sample $x_A \leftarrow \mathbb{Z}_q$ and sets $X_A = x_A \cdot G$.
   (d) Run $\Pi_{\mathsf{zk\text{-}uc}}^{L_{\mathsf{DL}}[(\mathbb{G},G,q)]}(X_A; x_A)$ generating a proof π_{DL}.
   (e) Send $(\mathsf{broadcast}, \mathsf{sid}, \mathsf{pid}_A, X_A, \pi_{\mathsf{DL}})$ to $\mathcal{F}_{\mathsf{broadcast}}$.
3. **Output.**
   - Each B_i does:
     (a) Receive $(\mathsf{sid}, \mathsf{pid}_A, X_A, \pi_{\mathsf{DL}})$
     (b) Verify π_{DL}, if fails consider A_{pid_A} malicious and abort, else continue.
     (c) Call the Random Oracle $\mathcal{H}(\mathsf{sid}, \mathbb{G}, G, q, X_A, X_{0,B}, X_{1,B}, \pi_{\mathsf{DL}})$ and receive $\mu_x^0, \mu_x^1, \mu_x^G$
     (d) Set $X = X_A + \mu_x^0 \cdot X_{0,B} + \mu_x^1 \cdot X_{1,B} + \mu_x^G \cdot G$.
     (e) Set $\mathsf{ct}_{\mathsf{key}} = \mu_x^0 \odot \mathsf{ct}_{0,\mathsf{key}} \oplus \mu_x^1 \odot \mathsf{ct}_{1,\mathsf{key}} \oplus \mu_x^G$.
     (f) Output X and records $(\mathsf{keygen}, \mathsf{pid}_A, X, X_A, \mathsf{ct}_{\mathsf{key}})$.
   - Party A_{pid_A} does:
     (a) Call the Random Oracle $\mathcal{H}(\mathsf{sid}, \mathbb{G}, G, q, X_A, X_{0,B}, X_{1,B}, \pi_{\mathsf{DL}})$ receiving $\mu_x^0, \mu_x^1, \mu_x^G$
     (b) Set $X = X_A + \mu_x^0 \cdot X_{0,B} + \mu_x^1 \cdot X_{1,B} + \mu_x^G \cdot G$.
     (c) Output X and records $(\mathsf{keygen}, \mathsf{pid}_A, X, X_A; x_A)$.

PROTOCOL C.2 (*Pre-sign* $\Pi^{ECDSA}_{\mathsf{Presign}}$: $\mathsf{Presign}(\mathbb{G}, G, q, sid, pk, X, X_A, ct_{key})$)

The protocol is parameterized with the ECDSA group description $(\mathbb{G}, G, q)$ along with an access structure Γ_B, a unique session id sid, a encryption key to a TAHE scheme pk, an ECDSA verification key X, the centralized party share of the verification key X_A and an encryption of the share of the distributed party of the signing key $\mathsf{ct_{key}}$. 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:

- ct_γ - an encryption of a randomizer γ.
- $\mathsf{ct}_{\gamma\cdot\mathsf{key}}$ - an encryption of the randomizer γ multiplied by the signing key share of the blockchain x_B.
- $\mathsf{ct}_{\gamma\cdot k_0}, \mathsf{ct}_{\gamma\cdot k_1}$ - an encryption of a randomized nonces $k_0 \cdot \gamma$ and $k_1 \cdot \gamma$ respectively.
- $R_{B,0}, R_{B,1}$ - points on the curve equal to $k_0 \cdot G$ and $k_1 \cdot G$ respectively.

1. **Each B_i does as follows**:
   - **Round 1**:
   (a) Sample $\gamma_i \leftarrow \mathbb{Z}_q$ and $\eta^i_1,\ \eta^i_2 \leftarrow \mathcal{R}_{pk}$, and compute:
      i. $\mathsf{ct}^i_\gamma = \mathsf{AHE.Enc}(\mathsf{pk}, \gamma_i; \eta^i_1)$
      ii. $\mathsf{ct}^i_{\gamma\cdot\mathsf{key}} = \mathsf{AHE.Scale}(\mathsf{pk}, \mathsf{ct_{key}}, \gamma_i; \eta^i_2)$
   (b) Run $\Pi^{\Gamma_B, L_{\mathsf{EncDH}}[\mathsf{pk},\mathsf{ct_{key}}]}_{\mathsf{agg}}$ on inputs $(\mathsf{ct}^i_\gamma, \mathsf{ct}^i_{\gamma\cdot\mathsf{key}}; \gamma_i, \eta^i_1, \eta^i_2)$ receiving $(\mathsf{ct}_\gamma, \mathsf{ct}_{\gamma\cdot\mathsf{key}})$.
   - **Round 2**:
   (a) Sample $k^i_0, k^i_1 \leftarrow \mathbb{Z}_q$ and $\eta^i_{3,0}, \eta^i_{3,1} \leftarrow \mathcal{R}_{pk}$ and compute:
      i. $\mathsf{ct}^i_{\gamma\cdot k_0} = \mathsf{AHE.Scale}(\mathsf{pk}, \mathsf{ct}_\gamma, k^i_0; \eta^i_{3,0})$
      ii. $\mathsf{ct}^i_{\gamma\cdot k_1} = \mathsf{AHE.Scale}(\mathsf{pk}, \mathsf{ct}_\gamma, k^i_1; \eta^i_{3,1})$
      iii. $R^i_{B,0} = k^i_0 \cdot G$
      iv. $R^i_{B,1} = k^i_1 \cdot G$
   (b) Run $\Pi^{\Gamma_B, L_{\mathsf{ScaleDL}}[\mathsf{pk},(\mathbb{G},G,q),\mathsf{ct}_\gamma]}_{\mathsf{agg}}$ on inputs $(\mathsf{ct}^i_{\gamma\cdot k_0}, R^i_{B,0}; k^i_0, \eta^i_{3,0})$ and $(\mathsf{ct}^i_{\gamma\cdot k_1}, R^i_{B,1}; k^i_1, \eta^i_{3,1})$ receiving $(\mathsf{ct}_{\gamma\cdot k_0}, R_{B,0}), (\mathsf{ct}_{\gamma\cdot k_1}, R_{B,1})$.
   - **Broadcast Round**:
   (a) Send $(\mathsf{global\text{-}broadcast}, \mathsf{sid}, i, (\mathsf{pid}_A, \mathsf{ct}_\gamma, \mathsf{ct}_{\gamma\cdot\mathsf{key}}, \mathsf{ct}_{\gamma\cdot k_0}, \mathsf{ct}_{\gamma\cdot k_1}, R_{B,0}, R_{B,1}))$ to $\mathcal{F}^{\Gamma_B}_{\mathsf{global\text{-}broadcast}}$.
2. **Output.**
   - A_{pid_A} receives $(\mathsf{global\text{-}broadcast}, \mathsf{sid}, (\mathsf{pid}_A, \mathsf{ct}_\gamma, \mathsf{ct}_{\gamma\cdot\mathsf{key}}, \mathsf{ct}_{\gamma\cdot k_0}, \mathsf{ct}_{\gamma\cdot k_1}, R_{B,0}, R_{B,1}))$ from $\mathcal{F}^{\Gamma_B}_{\mathsf{global\text{-}broadcast}}$.
   - A_{pid_A} verify that $R_{B,0}, R_{B,1} \in \mathbb{G} \setminus \{0\}$ and $\mathsf{ct}_\gamma, \mathsf{ct}_{\gamma\cdot\mathsf{key}}, \mathsf{ct}_{\gamma\cdot k_0}, \mathsf{ct}_{\gamma\cdot k_1} \in \mathcal{C}_{pk}$ else it aborts and considers B malicious.
   - Each party records $(\mathsf{sid}, X, \mathsf{ct}_\gamma, \mathsf{ct}_{\gamma\cdot\mathsf{key}}, \mathsf{ct}_{\gamma\cdot k_0}, \mathsf{ct}_{\gamma\cdot k_1}, R_{B,0}, R_{B,1})$

C.3 Reconfiguration Protocols

First let us introduce the following notations . For an element $z \in \mathbb{Z}$ we denote $[[z]]_b := (z_\ell)_{\ell\in[w]}$ as the b-ary decomposition of z i.e. $z = \sum_{\ell\in[w]} z_\ell b^\ell$ where $w := \lfloor \log_b(z) \rfloor$. For a sequence $(z_\ell)_{\ell\in[w]}$ we define the packing of the sequence in base b to be $\mathsf{Pack}_b((z_\ell)_{\ell\in[w]}) = \sum_{\ell\in[w]} z_\ell b^\ell$. Note that the input sequence may have elements larger then the base. In addition we define the following operations for brevity, they represent limb-wise operations:

1. $\mathbf{ct} \leftarrow \mathsf{Enc}^b_{\mathsf{pk}}(z, \vec{\eta}) = \mathsf{Enc}_{\mathsf{pk}}([[z]]_b) = \{\mathsf{Enc}_{\mathsf{pk}}(z_\ell, \eta_\ell)\}_{\ell \in [w]}$ where $\eta_\ell \leftarrow \mathcal{R}_{pk}$.
2. $\mathbf{ds}_i \leftarrow \mathsf{TDec}_{\mathsf{pk}}(\mathbf{ct}; [\mathsf{sk}]_i) = \{\mathsf{TDec}_{\mathsf{pk}}(\mathsf{ct}_\ell, [\mathsf{sk}]_i)\}_{\ell \in [w]}$.
3. $z \leftarrow \mathsf{Rec}^b_{\mathsf{pk}}(\{\mathbf{ds}_i\}_{i \in S}) = \mathsf{Pack}_b(\mathsf{Rec}_{\mathsf{pk}}(\{\mathsf{ds}^i_\ell\}_{i \in S}))$.
4. $\mathbf{ct}_3 \leftarrow \mathsf{Add}_{\mathsf{pk}}(\mathbf{ct}_1, \mathbf{ct}_2) = \{\mathsf{Add}_{\mathsf{pk}}(\mathbf{ct}^1_\ell, \mathbf{ct}^2_\ell)\}_{\ell \in [w]}$.

Here we provide the full protocols for reconfiguration.

The Varying Threshold Protocol C.6

The Constant Threshold Protocol C.7

D Omitted Security Proofs and Definitions

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

D.1 Proofs of Security for ECDSA Based Protocols

Proof of UC realization of the ECDSA protocol (Theorem 8.3) Next, we provide the full proof of Theorem 8.3.

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}.

Malicious B Simulation:

We assume, without loss of generality (WLOG), that all parties in B are corrupted. Thus, their part does not need simulation. For honest parties in B, the simulator follows the honest protocol, and the simulation remains unchanged.

Key Generation Simulation:

1. Upon receiving $(\mathsf{global\text{-}broadcast}, \mathsf{sid}, A, (\mathsf{pid}_A, X_{0,B}, X_{1,B}))$, the simulator verifies that $X_{0,B}, X_{1,B} \in \mathbb{G}$, otherwise, it aborts.
2. The simulator $\mathcal{S}$ sends to $\mathcal{F}^{\mathcal{G}^*_{\mathsf{SE\text{-}ECDSA}}}_{\mathsf{tsig}}$ the message $(\mathsf{keygen}, \mathsf{sid}, i)$ for $i \in U$. It then receives $(\mathsf{keygen}, \mathsf{sid}, X)$.
3. The simulator sends $(\mathsf{biaskey}, \mathsf{sid}, 1, 0, 1, 0)$ to $\mathcal{F}^{\mathcal{G}^*_{\mathsf{SE\text{-}ECDSA}}}_{\mathsf{tsig}}$ and receives (sid, X).
4. The simulator samples μ^0_x, μ^1_x, and μ^G_x and sets $X_B = \mu^0_x \cdot X_{0,B} + \mu^1_x \cdot X_{1,B} + \mu^G_x \cdot G$.
5. The simulator sets $X_A = X$ and sets the random oracle to return $(\mu^0_x, \mu^1_x, \mu^G_x)$ on the transcript. It then invokes the zero-knowledge (zk) property of $\Pi^{L_{\mathsf{DL}}}_{\mathsf{uc\text{-}zk}}$ to generate a proof π_{DL} for the statement X_A, ensuring it passes verification.
6. The simulator sends $(\mathsf{broadcast}, \mathsf{sid}, \mathsf{pid}_A, X_A, \pi_{\mathsf{DL}})$ to the adversary.

Pre-Sign and Sign Simulation:

1. The simulator first performs the following tasks:
   (a) The simulator extracts the private key sk correspondent to the published public key pk from a zk proof of the private key.
   (b) $\mathcal{S}$ sends $(\mathsf{pres}, \mathsf{sid}, \mathsf{ssid}, i)$ to $\mathcal{F}^{\mathcal{G}^*_{\mathsf{SE\text{-}ECDSA}}}_{\mathsf{tsig}}$ for $i \in U$, and receives $(\mathsf{pres}, \mathsf{sid}, \mathsf{ssid}, K_0, K_1)$ in response.
   (c) Upon receiving $(\mathsf{global\text{-}broadcast}, \mathsf{sid}, (\mathsf{pid}_A, \mathsf{ct}_\gamma, \mathsf{ct}_{\gamma\cdot\mathsf{key}}, \mathsf{ct}_{\gamma\cdot k_0}, \mathsf{ct}_{\gamma\cdot k_1}, R_{B,0}, R_{B,1}))$ 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) The simulator 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. The simulator also obtains $(\mu_k^0, \mu_k^1, \mu_k^G)$ from the random oracle.
   (e) The simulator then attempts to 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) The simulator 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) The simulator generates π_{kx}, π_R as in the protocol.
   (c) It then 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) The simulator sends everything to the adversary.
   (e) The simulator aborts if it receives a non-valid signature from the adversary.

3. Otherwise, the simulator does as follows:
   (a) The simulator 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 $\beta_{\mathsf{key}} = x'_B$.
   (b) The simulator 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) The simulator sets $R = K$ and r, and compute C_1, C_2, and a_2 according to r.
   (d) The simulator sets $\mathsf{ct}_A = \mathsf{AHE.Scale}(\mathsf{pk}, \mathsf{ct}_{\alpha,\beta}, s; *)$.
   (e) The simulator 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.
   (f) The simulator aborts if the signature it receives from the adversary is incorrect.

Malicious B 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 ZkP.

Pre-Sign and Sign: In the pre-sign 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 $X_A + x'_B \cdot G$. 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:

Key Generation Simulation:

1. The simulator holds sk as it simulated $\mathcal{F}_{\mathsf{DAHE}}$.
2. The simulator receives $(\mathsf{sid}, \tilde{X}_0, \tilde{X}_1)$ from $\mathcal{F}_{\mathsf{tsig}}^{\mathcal{G}^*_{\mathsf{SE\text{-}ECDSA}}}$.
3. For every $i \in B \setminus U$, the simulator samples $x_0^i, x_1^i \leftarrow \mathbb{Z}_q$ and $\eta_0^i, \eta_1^i \leftarrow \mathcal{R}_{pk}$. It sets $X_0^i = \tilde{X}_0 + x_0^i \cdot G$, $X_1^i = \tilde{X}_1 + x_1^i \cdot G$, $\mathsf{ct}_0^i = \mathsf{Enc}_{\mathsf{pk}}(x_0^i, \eta_0^i)$, and $\mathsf{ct}_1^i = \mathsf{Enc}_{\mathsf{pk}}(x_1^i, \eta_1^i)$. The simulator generates fake proofs using the zk property for the aggregation protocol and uses 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, the simulator flags i' as malicious. Otherwise, it decrypts $\mathsf{ct}_0^{i'}$ and $\mathsf{ct}_1^{i'}$ to obtain $x_0^{i'}$ and $x_1^{i'}$.
   (b) The simulator then emulates $\mathcal{F}_{\mathsf{ACS}}$ and receives an authorized subset S_B from the adversary. This subset must be one where the zk proofs have been validated. The simulator then computes:

– $X_{0,B} = \sum_{i \in S_B} X_0^i$
– $X_{1,B} \sum_{i \in S_B} X_1^i$
– $\mathsf{ct}_0 = \sum_{i \in S_B} \mathsf{ct}_0^i$
– $\mathsf{ct}_1 = \sum_{i \in S_B} \mathsf{ct}_1^i$

(c) The simulator receives from the adversary $(\mathsf{global\text{-}broadcast}, \mathsf{sid}, i, \mathsf{pid}_A, X_{0,B}, X_{1,B}$ for every $i \in S_B \setminus U$ and sends $(\mathsf{global\text{-}broadcast}, \mathsf{sid}, \mathsf{pid}_A, X_{0,B}, X_{1,B}$ to the adversary.

(d) Upon receiving $(\mathsf{broadcast}, \mathsf{sid}, \mathsf{pid}_A, X_A, \pi_{\mathsf{DL}})$, the simulator verifies π_{DL}. If the verification fails, it aborts; otherwise, it extracts the witness x_A (this can be done without rewinding since it's 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 not, 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) The simulator sets
  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) The simulator sends $(\mathsf{bias}, \mathsf{sid}, \alpha_x^{(0)}, \beta_x^{(0)}, \alpha_x^{(1)}, \beta_x^{(1)})$ to $\mathcal{F}_{\mathsf{tsig}}^{\mathcal{G}^*_{\mathsf{SE\text{-}ECDSA}}}$.

(h) The simulator computes $X_B = \overline{\mu_x^0} \cdot X_{0,B} + \overline{\mu_x^1} \cdot X_{1,B} + \overline{\mu_x^G} \cdot G$ and $\mathsf{ct}_{\mathsf{key}}$ respectively.

Pre-Sign 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_0^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: $(\mathsf{sid}, \mathsf{msg}, \mathbb{G}, G, q, H, X, (\mathsf{ct}_\gamma, \mathsf{ct}_{\gamma\cdot\mathsf{key}}, \mathsf{ct}_{\gamma\cdot k_0}, \mathsf{ct}_{\gamma\cdot k_1}, R_{B,0}, R_{B,1}), C_k, C_{kx}, X_A, C_\alpha, C_\beta, \pi_k, \pi_\alpha, \pi_\beta)$, 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 extract 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 $(\mathsf{sign}, \mathsf{msg}, \mathsf{sid}, \mathsf{ssid}, 0, |S_{1,B} \setminus U| \cdot \rho, \rho \cdot k_{B,0}, |S_{1,B} \setminus U|, k_{B,1})$ 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_0^i and ct_1^i. 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 $\mathcal{F}_{\mathsf{tsig}}^{\mathcal{G}^*_{\mathsf{SE\text{-}ECDSA}}}$ as:

$$\begin{aligned}
X &= \mu_x^0 \cdot X_0 + \mu_x^1 \cdot X_1 + \mu_x^G \cdot G \\
&= \mu_x^0 \cdot (\alpha_x^0 \cdot \tilde{X}_0 + \beta_x^0 \cdot G) + \mu_x^1 \cdot (\alpha_x^1 \cdot \tilde{X}_1 + \beta_x^1 \cdot G) + \mu_x^G \cdot G \\
&= \mu_x^0 \cdot (|S_B \setminus U| \cdot \tilde{X}_0 + \rho \cdot x_A \cdot G + \sum_{i \in S_B} x_0^i \cdot G) + \\
&\qquad\qquad + \mu_x^1 \cdot (|S_B \setminus U| \cdot \tilde{X}_1 + \sum_{i \in S_B} x_1^i \cdot G) + \mu_x^G \cdot G \\
&= \mu_x^0 \cdot (\rho \cdot x_A \cdot G + \sum_{i \in U} (x_0^i \cdot G) + \sum_{i \in S_B \setminus U} (\tilde{X}_0 + x_0^i \cdot G)) + \\
&\qquad\qquad + \mu_x^1 \cdot (\sum_{i \in U} (x_1^i \cdot G) + \sum_{i \in S_B \setminus U} (\tilde{X}_1 + x_1^i \cdot G)) + \mu_x^G \cdot G \\
&= \mu_x^0 \cdot (\rho \cdot X_A + \sum_{i \in U} X_0^i + \sum_{i \in S_B \setminus U} X_0^i) + \\
&\qquad\qquad + \mu_x^1 \cdot (\sum_{i \in U} X_1^i + \sum_{i \in S_B \setminus U} X_1^i) + \mu_x^G \cdot G \\
&= \mu_x^0 \cdot (\rho \cdot X_A + \sum_{i \in S_B} X_0^i) + \mu_x^1 \cdot \sum_{i \in S_B} X_1^i + \mu_x^G \cdot G \\
&= \mu_x^0 \cdot (\rho \cdot X_A + X_{0,B}) + \mu_x^1 \cdot X_{1,B} + \mu_x^G \cdot G \\
&= \overline{\mu_x^0} \cdot (\rho \cdot X_A + X_{0,B}) + \overline{\mu_x^1} \cdot X_{1,B} + (\overline{\mu_x^G} + (1 - \rho \cdot \overline{\mu_x^0}) x_A) \cdot G \\
&= X_A + \overline{\mu_x^0} \cdot X_{0,B} + \overline{\mu_x^1} \cdot X_{1,B} + \overline{\mu_x^G} \cdot G
\end{aligned}$$

Pre-sign and sign: In the pre-sign 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 pt_4. 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:

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

Proof of the Security of Slightly Enhanced ECDSA Our security analysis for Slightly Enhanced ECDSA is based on the approach introduced in [GS22]. 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 [GS22]) with the notation in the rest of this paper.

A simulation of Functionality 8.1 in the EC-GGM is presented in Figure D.1. 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.

The design of the lazy simulation D.1 and the reduction to the symbolic simulation D.2 are essentially the same as the security proof of ECDSA with presigns in [GS22, 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 D.3: 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

Object	Notation: [GS22]	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, [GS22] vs this work.

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 [GS22], 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 enable 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.

Similarly to [GS22], we begin with presenting a corresponding symbolic simulation:

Lemma D.1 *A PPT adversary forging signatures for the lazy simulation (Simulation D.1) with success rate ϵ, can forge signatures for the symbolic simulation (Simulation D.2) with success rate $\epsilon + \mathcal{O}(\frac{N^2}{q})$.*

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 $i \neq j \in \mathbb{Z}_q[\mathbf{d}, \tilde{\mathbf{r}}_1^{(0)}, \tilde{\mathbf{r}}_1^{(1)}, \tilde{\mathbf{r}}_2^{(0)}, \tilde{\mathbf{r}}_2^{(1)} \ldots]$ where $(i, \mathcal{P}_i)$ and $(j, \mathcal{P}_j)$ are recorded, yet $i(d, \tilde{r}_1^{(1)}, \tilde{r}_2^{(1)}, \ldots) = j(d, \tilde{r}_1^{(1)}, \tilde{r}_2^{(1)}, \ldots)$. 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 [GS22, 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 D.2 *A PPT adversary forging signatures for the symbolic simulation (Simulation D.2) with success rate ϵ, can forge signatures for the modified symbolic simulation (Simulation D.3) with success rate $\epsilon + \mathcal{O}(\frac{N^2}{q})$.*

Proof. There are two differences between the simulations — in the bias key and the sign queries. We begin with the bias key query: In the symbolic simulation, the adversary sends coefficients $\alpha_x^{(0)}, \beta_x^{(0)}, \alpha_x^{(1)}, \beta_x^{(1)}$, 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 then 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 $\mu_x^0\tilde{\mathbf{d}}_0 + \mu_x^1\tilde{\mathbf{d}}_1 + \mu_x^G$.

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 $\alpha_x^{(0)}$ and $\beta_x^{(0)}$; otherwise, stop and return an arbitrary output. Similarly, extract $\alpha_x^{(1)}$ and $\beta_x^{(1)}$ 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
$$(\mu_x'^0, \mu_x'^1, \mu_x'^G) = \left(\frac{\mu_x^0}{\alpha_x^{(0)}}, \frac{\mu_x^1}{\alpha_x^{(1)}}, \mu_x^G - \frac{\mu_x^0\beta_x^{(0)}}{\alpha_x^{(0)}} - \frac{\mu_x^1\beta_x^{(1)}}{\alpha_x^{(1)}}\right).$$

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 D.2 and Simulation D.3 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 to cancel 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 D.3 *Let $\mathcal{A}$ be an adversary to the existential unforgeability game D.6 with respect to the ECDSA oracle $\mathcal{G}^*_{\mathsf{SE\text{-}ECDSA}}$ (Functionality 8.1) in the EC-GGM model (LazySimulation D.1), that makes at most N presignature, signing, hash, or group queries, when $\mathcal{H}_{\mathsf{key}}, \mathcal{H}_{\mathcal{M}}$ and all $\mathcal{H}_k$ are modeled as independent Random Oracles. Then*

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

Where $C = |\mathcal{C}|$ is the size of the offset space allowed to the adversary.

Remark D.1. We note that the option of adding an offset $e \in \mathcal{C}$ to the secret key, independently for each signature request, is enabled for compatibility with the Additive Key Derivation supported by BIP32. Usecases that are not interested in Additive Key Derivation may set $\mathcal{C} = \{0\}$ and $C = 1$, and obtain the bound $\mathsf{Adv}(\mathcal{A}, \mathsf{Exp}_{\mathsf{EU}}^{\mathcal{G}^*_{\mathsf{SE\text{-}ECDSA}}}, \text{lazy-sim}) \leq \mathcal{O}(\frac{N^2}{q})$.

Proof. By Lemma D.1 and Lemma D.2, it is sufficient to bound the advantage in the modified symbolic simulation (Simulation D.3). Following the proof of [GS22, Theorem 1], we denote the forgery by $(m^*, R^*, s^*, t^*, e^*)$. The forgery is successful if the equation

$$\pi^{-1}(R^*) = s^{*-1}(h^* + t^*(\mu_x^0 \tilde{\mathbf{d}}_0 + \mu_x^1 \tilde{\mathbf{d}}_1 + \mu_x^G + e^*)) \tag{15}$$

holds. Without loss of generality, we assume that the adversary evaluates D before making the key bias request, evaluates R_k before requesting the corresponding signature, and verifies its own forgery before outputting it. Consequently, every initial application of a map query originates either from $\mathcal{A}$'s group query, the key generation process, or a presign process. We split the forgery into several types and analyze the probability of each separately.

Type I forger: Suppose R^* first appeared prior to the key bias request. Consequently, it must have appeared before any presign or sign request. At that point, the only formal variables present were $\tilde{\mathbf{d}}_0$ and $\tilde{\mathbf{d}}_1$. Therefore, the query that generated R^* must have been of the form $(\mathsf{map}, c_0\tilde{\mathbf{d}}_0 + c_1\tilde{\mathbf{d}}_1 + c_2)$, implying that $R^* = \pm\pi(c_0\tilde{\mathbf{d}}_0 + c_1\tilde{\mathbf{d}}_1 + c_2)$, where the equality holds up to sign, since each group query maps an element along with its negation. By (15), we obtain that

$$c_0\tilde{\mathbf{d}}_0 + c_1\tilde{\mathbf{d}}_1 + c_2 = \eta s^{*-1}(h^* + t^*(\mu_x^0 \tilde{\mathbf{d}}_0 + \mu_x^1 \tilde{\mathbf{d}}_1 + \mu_x^G + e^*))$$

for some $\eta \in \{\pm 1\}$, implying that

$$c_0 = \eta s^{*-1} t^* \mu_x^0, \qquad c_1 = \eta s^{*-1} t^* \mu_x^1, \qquad \text{and} \quad c_2 = \eta s^{*-1}(h^* + t^*(\mu_x^G + e^*)). \tag{16}$$

Since η, s^*, t^*, and μ_x^1 are all required to be nonzero, we obtain that $c_1 \neq 0$. Moreover, we obtain

$$\frac{c_0}{c_1} = \frac{\mu_x^0}{\mu_x^1} \qquad \text{and} \qquad \frac{c_2}{c_1} = \frac{h^* + t^*(\mu_x^G + e^*)}{t^*\mu_x^1} \Longrightarrow h^* = t^*(\frac{c_2\mu_x^1}{c_1} - \mu_x^G - e^*). \tag{17}$$

Now let us bound the probability of a Type I forgery. First we denote by Z_{I} the event that at any point during $\mathcal{A}$'s execution, one of the following two events occurs: (1) $\mathcal{H}_{\mathsf{key}}$ produces a pair of triplets $(\mu_x^0, \mu_x^1, \mu_x^G)$ with the same ratio μ_x^0/μ_x^1, or (2) $\mathcal{H}_{\mathcal{M}}$ outputs 0. Clearly, $\Pr[Z_{\mathrm{I}}] = \mathcal{O}(\frac{N^2}{q})$. Next, we fix an index i of a group query of the form $R^* \leftarrow (\mathsf{map}, c_0^{(i)}\tilde{\mathbf{d}}_0 + c_1^{(i)}\tilde{\mathbf{d}}_1 + c_2^{(i)})$, and fix values h^* and e^*, and denote by $B^{\mathrm{I}}_{i,h^*,e^*}$ the event that Z_{I} does not occur and $\mathcal{A}$ forges a signature using h^*, e^*, and R^*. We can easily bound the probability of $B^{\mathrm{I}}_{i,h^*,e^*}$: Since μ_x^0/μ_x^1 is assumed to be unique and by (17), after picking $(c_0^{(i)}, c_1^{(i)}, c_2^{(i)})$ there is at most one suitable choice for $(\mu_x^0, \mu_x^1, \mu_x^G)$. Therefore, and again by (17), there is at most one suitable value for t^* (recall that assuming Z_{I} does not occur, we have $h^* \neq 0$). Since t^* is sampled uniformly after picking $(c_0^{(i)}, c_1^{(i)}, c_2^{(i)})$, we obtain that $\Pr[B^{\mathrm{I}}_{i,h^*,e^*}] \leq \mathcal{O}(\frac{1}{q})$. Combining all the events, we obtain that the probability of a Type I forgery is at most

$$\Pr[Z_{\mathrm{I}}] + \sum_{i\in[N], h^*\in\mathsf{Range}(\mathcal{H}_{\mathcal{M}}), e\in\mathcal{C}} \Pr[B^{\mathrm{I}}_{i,h^*,e^*}] \leq \mathcal{O}(\frac{N^2}{q}) + CN^2 \cdot \mathcal{O}(\frac{1}{q}) = \mathcal{O}(\frac{CN^2}{q}).$$

Type II forger: $R^* = \eta R_k$ for some index k and $\eta \in \{\pm 1\}$. Following the description of the sign oracle, we obtain that

$$\pi^{-1}(R_k) = s_k^{-1}(h_k + t_k(\mu_x^0\tilde{\mathbf{d}}_0 + \mu_x^1\tilde{\mathbf{d}}_1 + \mu_x^G + e_k)).$$

Additionally, from the forgery we obtain the equation

$$\pi^{-1}(R^*) = s^{*-1}(h^* + t^*(\mu_x^0\tilde{\mathbf{d}}_0 + \mu_x^1\tilde{\mathbf{d}}_1 + \mu_x^G + e^*)),$$

where $h^* = \mathcal{H}_{\mathcal{M}}(m^*)$. Recall that $t_k = \bar{C}(R_k)$ and $t^* = \bar{C}(R^*)$, and that $\bar{C}(-R) = \bar{C}(R)$. Therefore, from $R^* = \eta R_k$ we obtain $t_k = t^*$. Therefore, the equation

$$\eta s_k^{-1}(h_k + t_k(\mu_x^0\tilde{\mathbf{d}}_0 + \mu_x^1\tilde{\mathbf{d}}_1 + \mu_x^G + e_k)) = s^{*-1}(h^* + t_k(\mu_x^0\tilde{\mathbf{d}}_0 + \mu_x^1\tilde{\mathbf{d}}_1 + \mu_x^G + e^*))$$

holds as a symbolic equation, implying that

$$\eta s_k^{-1} t_k \mu_x^0 = s^{*-1} t_k \mu_x^0 \quad \text{and} \quad \eta s_k^{-1}(h_k + t_k(\mu_x^G + e_k)) = s^{*-1}(h^* + t_k(\mu_x^G + e^*)). \tag{18}$$

Recall that both t_k and μ_x^0 are nonzero. Therefore, (18) implies

$$h_k + t_k e_k = h^* + t_k e^*. \tag{19}$$

Notably, the signing process implies that the initial query generating R_k is of the form $R_k \leftarrow (\mathsf{map}, \mu_k^0\tilde{\mathbf{r}}_k^{(0)} + \mu_k^1\tilde{\mathbf{r}}_k^{(1)} + \mu_k^G)$.

Now let us bound the probability of a Type II forgery. First, we denote by Z_{II} the event that at any point during $\mathcal{A}$'s execution, one of the following two events occurs: (1) for some index k, $\mathcal{H}_k$ produces the same triplet $(\mu_k^0, \mu_k^1, \mu_k^G)$ (up to the sign) for two different inputs, or (2) $\mathcal{H}_{\mathcal{M}}$ produces the same output for two different messages. Clearly, $\Pr[Z_{\mathrm{II}}] = \mathcal{O}(\frac{N^2}{q})$. Next, we fix an index i of a group query of the form $R_k \leftarrow (\mathsf{map}, (\mu_k^0)_i\tilde{\mathbf{r}}_k^{(0)} + (\mu_k^1)_i\tilde{\mathbf{r}}_k^{(1)} + (\mu_k^G)_i)$, fix values h^* and e^*, and denote by $B^{\mathrm{II}}_{i,h^*,e^*}$ the event that Z_{II} does not occur and $\mathcal{A}$ forges a signature using h^*, e^*, and $R^* = \pm R_k$. We can bound the probability of $B^{\mathrm{II}}_{i,h^*,e^*}$: The equation $R^* = \pm R_k$ implies that $((\mu_k^0)_i, (\mu_k^1)_i, (\mu_k^G)_i) = \pm\mathcal{H}_k(D, e_k, \tilde{R}_k^{(0)}, \tilde{R}_k^{(1)}, m_k, \mathsf{id}_k)$. Since the outputs of $\mathcal{H}_k$ are assumed to be unique, after picking $((\mu_k^0)_i, (\mu_k^1)_i, (\mu_k^G)_i)$ there is at most one suitable choice for m_k and e_k. Moreover, (19) implies $t_k = \frac{h_k - h^*}{e^* - e_k}$. Notably, $e^* \neq e_k$, since an equality would imply $h^* = h_k$ by (19) and contradict the assumption that $\mathcal{H}_{\mathcal{M}}$ produces unique outputs. Therefore, there is at most one suitable value for t_k that is counted by $B^{\mathrm{II}}_{i,h^*,e^*}$. Since t_k is sampled uniformly after picking $((\mu_k^0)_i, (\mu_k^1)_i, (\mu_k^G)_i)$, we obtain that $\Pr[B^{\mathrm{II}}_{i,h^*,e^*}] \le \mathcal{O}(\frac{1}{q})$. Combining all the events, we obtain that the probability of a Type II forgery is at most

$$\Pr[Z_{\mathrm{II}}] + \sum_{i\in[N],h^*\in\mathsf{Range}(\mathcal{H}_{\mathcal{M}}),e\in\mathcal{C}} \Pr[B^{\mathrm{II}}_{i,h^*,e^*}] \le \mathcal{O}(\frac{N^2}{q}) + CN^2\cdot\mathcal{O}(\frac{1}{q}) = \mathcal{O}(\frac{CN^2}{q}).$$

Type III forger: A forger which is neither Type I nor Type II. Aside from the key bias process (which is covered by Type I) and the signing process (covered by Type II), there are two additional ways of generating group elements: group queries and presign queries. Importantly, a forgery cannot involve a group element R^* that was generated by a presign query. The reason is as follows: Suppose that $R^* \in \{\pm\tilde{R}^{(0)}_k, \pm\tilde{R}^{(1)}_k\}$. If $R^* = \pm\tilde{R}^{(1)}_k$, then $\pi^{-1}(R^*) = \pm\tilde{\mathbf{r}}^{(1)}_k$, and no future substitution will change it. On the other hand, if $R^* = \pm\tilde{R}^{(0)}_k$, then $\pi^{-1}(R^*) = \pm\tilde{\mathbf{r}}^{(0)}_k$, and after the substitution we will obtain

$$\pi^{-1}(R^*) = \pm\left(\mu^0_k\right)^{-1}\left(-\mu^1_k\tilde{\mathbf{r}}^{(1)}_k + s_k^{-1}t_k(\mu^0_x\tilde{\mathbf{d}}_0 + \mu^1_x\tilde{\mathbf{d}}_1 + \mu^G_x + e_k) + s_k^{-1}h_k - \mu^G_k\right).$$

Since both μ^0_k and μ^1_k are nonzero, we obtain that in either case, the final value of $\pi^{-1}(R^*)$ will involve $\tilde{\mathbf{r}}^{(1)}_k$. However, a successful forgery requires $\pi^{-1}(R^*) = s^{*-1}(h^* + t^*(\mu^0_x\tilde{\mathbf{d}}_0 + \mu^1_x\tilde{\mathbf{d}}_1 + \mu^G_x + e^*))$, which does not involve $\tilde{\mathbf{r}}^{(1)}_k$.

We may conclude that in every Type III forgery, the initial query generating R^* is a group query. We denote its initial preimage by

$$\pi^{-1}(R^*) = a + b_0\tilde{\mathbf{d}}_0 + b_1\tilde{\mathbf{d}}_1 + \sum_{k\in K_{\mathrm{III}}}\left(c_k\tilde{\mathbf{r}}^{(0)}_k + c'_k\tilde{\mathbf{r}}^{(1)}_k\right), \tag{20}$$

where for all $k \in K_{\mathrm{III}}$ we have either $c_k \neq 0$ or $c'_k \neq 0$. Note that we may have $K_{\mathrm{III}} = \emptyset$. By the verification equation we have

$$\pi^{-1}(R^*) = s^{*-1}(h^* + t^*(\mu^0_x\tilde{\mathbf{d}}_0 + \mu^1_x\tilde{\mathbf{d}}_1 + \mu^G_x + e^*)). \tag{21}$$

Therefore, the substitutions must eliminate all variables in $\pi^{-1}(R^*)$ except for $\tilde{\mathbf{d}}_0$ and $\tilde{\mathbf{d}}_1$. First, we observe that all presigns $k \in K_{\mathrm{III}}$ must be used before the forgery. Indeed, if the k'th presign is never used, then the final value of $\pi^{-1}(R^*)$ will involve either $\tilde{\mathbf{r}}^{(0)}_k$ or $\tilde{\mathbf{r}}^{(1)}_k$, contradicting (21). Moreover, we observe that after the k'th presign is used, the coefficient of $\tilde{\mathbf{r}}^{(1)}_k$ in (20) becomes $c'_k - \frac{c_k\mu^1_k}{\mu^0_k}$. Again from (21), we obtain that $\tilde{\mathbf{r}}^{(1)}_k$ must be eliminated in $\pi^{-1}(R^*)$, so $c'_k = \frac{c_k\mu^1_k}{\mu^0_k}$. If $c'_k = 0$ then this equation implies $c_k = 0$, contradicting the assumption, so we obtain that $c'_k \neq 0$ and therefore

$$\forall k \in K_{\mathrm{III}} : \frac{c_k}{c'_k} = \frac{\mu^0_k}{\mu^1_k}. \tag{22}$$

Let us bound the probability of a Type III forgery. First we denote by Z_{III} the event that at any point during $\mathcal{A}$'s execution, one of the following two events occurs: (1) for some index k, $\mathcal{H}_k$ produces a pair of triplets $(\mu^0_k, \mu^1_k, \mu^G_k)$ with the same ratio μ^0_k/μ^1_k, (2) $\mathcal{H}_{\mathcal{M}}$ outputs 0, or (3) a map query of the form of (20) is applied, and *afterwards* $\mathcal{H}_k$ produces a triplet $(\mu^0_k, \mu^1_k, \mu^G_k)$ with $\mu^0_k/\mu^1_k = c_k/c'_k$ for some k. Clearly, $\Pr[Z_{\mathrm{III}}] = \mathcal{O}(\frac{N^2}{q})$. Next, we fix an index i of a group query of the form

$$R^* \leftarrow (\mathsf{map}, a^{(i)} + b^{(i)}_0\tilde{\mathbf{d}}_0 + b^{(i)}_1\tilde{\mathbf{d}}_1 + \sum_{k\in K_{\mathrm{III}}}(c^{(i)}_k\tilde{\mathbf{r}}^{(0)}_k + c'^{(i)}_k\tilde{\mathbf{r}}^{(1)}_k)),$$

fix values h^* and e^*, and denote by $B^{\mathrm{III}}_{i,h^*,e^*}$ the event that Z_{III} does not occur and $\mathcal{A}$ forges a signature using h^*, e^*, and R^*. We can bound the probability of $B^{\mathrm{III}}_{i,h^*,e^*}$: For a successful forgery using the ith group query, we obtain by (22) that $\mathcal{A}$ must have $\frac{c_k^{(i)}}{c_k'^{(i)}} = \frac{\mu_k^0}{\mu_k^1}$ for all $k \in K_{\mathrm{III}}$, where $(\mu_k^0, \mu_k^1, \mu_k^G) = \mathcal{H}_k(D, e_k, \tilde{R}_k^{(0)}, \tilde{R}_k^{(1)}, m_k, \mathsf{id}_k)$ is the triplet that will be used in the kth signing process. Since the ratios μ_k^0/μ_k^1 are assumed to be unique, we obtain that for every $k \in K_{\mathrm{III}}$, after picking $c_k^{(i)}$ and $c_k'^{(i)}$ there is at most one suitable choice for $(\mu_k^0, \mu_k^1, \mu_k^G)$, m_k, and e_k. By fixing the values sampled during the kth signing process for all $k \in K_{\mathrm{III}}$, we obtain that after picking the ith query, there is at most one option for the final expression representing $\pi^{-1}(R^*)$ for which a forgery is possible, and this expression is of the form

$$\pi^{-1}(R^*) = A^{(i)} + B_0^{(i)}\tilde{\mathbf{d}}_0 + B_1^{(i)}\tilde{\mathbf{d}}_1. \tag{23}$$

Intuitively, the reason is that $\mathcal{A}$ needs the coefficients in (20) to satisfy the equation (22) in order to eliminate the variables $\tilde{\mathbf{r}}_k^{(0)}$ and $\tilde{\mathbf{r}}_k^{(1)}$, and there is at most one possible strategy to satisfy all these equations. Since the coefficients $A^{(i)}, B_0^{(i)}$, and $B_1^{(i)}$ are fixed, we may observe that (23) is equivalent to Type I forgeries and deduce that $\Pr[B^{\mathrm{III}}_{i,h^*,e^*}]$ is also bounded by $\mathcal{O}(\frac{1}{q})$. Combining all the events, we obtain that the probability of a Type III forgery is at most

$$\Pr[Z_{\mathrm{III}}] + \sum_{i\in[N],h^*\in\mathsf{Range}(\mathcal{H}_{\mathcal{M}}),e\in\mathcal{C}} \Pr[B^{\mathrm{III}}_{i,h^*,e^*}] \le \mathcal{O}(\frac{N^2}{q}) + CN^2 \cdot \mathcal{O}(\frac{1}{q}) = \mathcal{O}(\frac{CN^2}{q}).$$

D.2 Proofs of Security for Schnorr Based Protocols

Proof of UC realization of the Schnorr-based protocol (Theorem 8.4)

Next, we provide the full proof of Theorem 8.4.

Proof. The simulator works as follows:

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

Malicious B: We assume, without loss of generality (WLOG), that all parties in B are corrupted. Therefore, we only simulate 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.

Malicious A_{pid_A}:

1. For each $i \in B \setminus U$, the simulator randomizes $k_0^i, k_1^i \leftarrow \mathbb{Z}_q$ and $\eta_0^i, \eta_1^i \leftarrow \mathcal{R}_{pk}$. It sets $K_{B,0}^i = K_0 + k_0^i \cdot G$ and $K_{B,1}^i = K_1 + k_1^i \cdot G$. The simulator then invokes the zk property to generate 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 $i \in U$ along with zk 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 set either contains malicious parties with validated proofs or honest parties. The simulator sets $k_0' = \sum_{i\in S_B} k_0^i$ and $k_1' = \sum_{i\in S_B} k_1^i$.
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:

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 signature $\sigma = (r, s)$ with public nonce K.
2. The simulator randomizes values $\mu_k, e, z_A \leftarrow \mathbb{Z}_q$, sets $K_A = z_A \cdot G - e \cdot X_A$, and computes $K = K_{B,0} + \mu_k \cdot K_{B,1} + K_A$. 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 signature σ, the simulator verifies its validity. If the signature fails verification, it aborts.

Indistinguishability: The adversary's view in this simulation consists of $(X, \pi_{\mathsf{DL}}, z_A, K_A)$ along with e and μ_k from the random oracle. All these values are uniformly distributed and satisfy the checks imposed by the adversary, particularly that the zk-proof verifies and that $z_A \cdot G = K + e \cdot X_A$. Hence, the only way for the adversary to distinguish between the real and ideal worlds is through the distribution of aborts. The only difference in aborts between the real and ideal executions could occur during the verification of the signature, i.e., when the adversary forges a signature. Assuming such an adversary exists, we can construct one that breaks the unforgeability assumption. Since the messages in the protocol are indistinguishable from an interaction with a signing oracle, if the adversary produces a signature that verifies on a message $\mathsf{msg}' \neq \mathsf{msg}$ that was not signed, the simulator can use this signature to send to the signing oracle and break the unforgeability assumption.

Malicious A_{pid_A}:

1. Upon receiving (z_A, K_A), the simulator verifies that $z_A \cdot G = K_A + e \cdot X_A$. If the check fails, it aborts.
2. The simulator calculates $k_A = z_A - e x_A$ (where x_A was extracted during the key generation simulation) and computes the following values:
   (a) $\alpha_0 = n_H$
   (b) $\alpha_1 = n_H$
   (c) $\beta_0 = k'_0$
   (d) $\beta_1 = k'_1$
3. The simulator sends $(\mathsf{sign}, \mathsf{msg}, \mathsf{sid}, \mathsf{ssid}, (\alpha_0, \beta_0, \alpha_1, \beta_1))$ to $\mathcal{F}_{\mathsf{tsig}}^{\mathcal{G}^*_{\mathsf{SE\text{-}Sch}}}$ and receives a valid signature (z, e).
4. The simulator emulates $\mathcal{F}_{\mathsf{DAHE}}$ to return z as the plaintext.
5. The simulator emulates $\mathcal{F}_{\mathsf{global\text{-}broadcast}}$ to send $(\mathsf{global\text{-}broadcast}, \mathsf{sid}, (z, e))$ to the adversary.

Proof of the Security of Slightly Enhanced Schnorr-based Oracle (Theorem 8.2) We prove Theorem 8.2 first by defining the Generalized Forking Lemma (Lemma D.4) as in [BN06, Lemma 1]. Then, we define Non-Biased Slightly Enhanced Schnorr-based oracle (Functionality D.4), existential unforgeability experiment (Experiment D.6), and Algebraic One More Discrete Log experiment (Experiment D.7). We also provide a simulation of a Slightly Enhanced Schnorr Signing oracle using a Non-Biased Slightly Enhanced Schnorr Signing oracle (Simulation D.10), thus proving that they are equivalent.

Lemma D.4 (Generalized Forking Lemma [BN06, Lemma 1]) *Let $n_r \geq 1$ be an integer. Let $\mathcal{A}_{n_r}(inp, h_1, \dots h_{n_r}; \rho)$ 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 a random oracle uniformly samples from H. Let $\mathsf{acc}(\mathcal{A}_{n_r})$ be the probability, over $inp \leftarrow \mathsf{InpGen}$, $h_1, \dots h_{n_r} \leftarrow H$ and $\rho \leftarrow R$ for some given sets H and R, that $\mathcal{A}_{n_r}(inp, h_1, \dots h_{n_r}; \rho)$ does not return $\perp$. Consider $\mathsf{Fork}_{\mathcal{A}_{n_r}}(h_1, h_1', \dots, h_{n_r}, h_{n_r}')$ as in Simulation D.5. The probability that $\mathsf{Fork}_{\mathcal{A}_{n_r}}(h_1, h_1', \dots, h_{n_r}, h_{n_r}')$ does not return $\perp$, where $h_1, \dots, h_{n_r}$ and $h_1', \dots, h_{n_r}'$ are uniformly sampled 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 (Proof of Theorem 8.2). Let $\mathcal{A}^{n_\mathsf{s}, n_\mathsf{kg}, n_\mathsf{r}}_{\mathsf{SE-Sch-Forger}}$ be a PPT algorithm that satisfies $\mathsf{Exp}^{\mathcal{G}^*_{\text{Non-Biased-SE-Sch}}}_{\mathsf{EU}}(\mathcal{A}, 1^\kappa, n_\mathsf{s}, n_\mathsf{kg})$ with probability at least ϵ, and uses at most n_r calls to the random oracle $\mathcal{H}$. Then, $\mathcal{S}^{\mathsf{Exp}_\mathsf{AOMDL}}_{\mathsf{Exp}_\mathsf{EU}(\mathcal{A}^{n_\mathsf{s}, n_\mathsf{kg}, n_\mathsf{r}}_{\mathsf{SE-Sch-Forger}})}$ and $\mathcal{A}^{\mathcal{S}_{\mathsf{Exp}_\mathsf{EU}}}_{\mathsf{AOMDL-Sch}}$ are well defined. The algorithm $\mathcal{A}^{\mathcal{S}_{\mathsf{Exp}_\mathsf{EU}}}_{\mathsf{AOMDL-Sch}}$ until command line 8 is equivalent to $\mathsf{Fork}_{\mathcal{S}^{\mathsf{Exp}_\mathsf{AOMDL}}_{\mathsf{Exp}_\mathsf{EU}(\mathcal{A}^{n_\mathsf{s}, n_\mathsf{kg}, n_\mathsf{r}}_{\mathsf{SE-Sch-Forger}})}}$. Thus, according to the Generalized Forking Lemma (Lemma D.4), the success probability of $\mathcal{A}^{\mathcal{S}_{\mathsf{Exp}_\mathsf{EU}}}_{\mathsf{AOMDL-Sch}}$ until command line 8 is at least

$$\epsilon \left(\frac{\epsilon}{n_\mathsf{r}} - \frac{1}{|H|} \right)$$

Therefore, the success probability of $\mathcal{A}^{\mathcal{S}_{\mathsf{Exp}_\mathsf{EU}}}_{\mathsf{AOMDL-Sch}}$ is at least $\epsilon \left(\frac{\epsilon}{n_\mathsf{r}} - \frac{1}{|H|} \right)$ times the chance that the vectors would be linearly dependent. The sampled vectors are linearly dependent if and only if a presign has the same value of μ for the two signatures. The probability of a presign with the same μ for two signatures is at most $\frac{n_\mathsf{s}}{|H|}$. Thus, the success probability of $\mathcal{A}^{\mathcal{S}_{\mathsf{Exp}_\mathsf{EU}}}_{\mathsf{AOMDL-Sch}}$ is at least

$$\epsilon \left(\frac{\epsilon}{n_\mathsf{r}} - \frac{1}{|H|} \right) \left(1 - \frac{n_\mathsf{s}}{|H|} \right)$$

The expected running time of $\mathcal{A}^{\mathcal{S}_{\mathsf{Exp}_\mathsf{EU}}}_{\mathsf{AOMDL-Sch}}$ is bounded by $2\text{TIME}(\mathcal{A}^{n_\mathsf{s}, n_\mathsf{kg}, n_r}_{\mathsf{SE-Sch-Forger}}) + \mathcal{O}(n_\mathsf{r} + n_\mathsf{s} + n_\mathsf{kg})$.

D.3 Proof of Realization of the Reconfiguration (Theorem 8.5)

Next, we provide the full proof of Theorem 8.5.

Proof. We start with the varying threshold protocol: First let us describe the correctness of the protocol:

Correctness: We want to see that one can decrypt as typical: $\sum_{j_R \in S_{B_R}} \Delta \lambda^{S_{B_R}}_{0, j_R} [\mathsf{sk}]_{j_R} = \Delta^2 \mathsf{sk}$. First let us make a few notations we denote $r = \sum_{j_T \in S^1_{B_T}} \sum_{\ell \in [w]} [[r_{j_T}]]_{b \cdot 2^\sigma} b^\ell$. Then note that $\mathsf{sk}_\mathsf{masked} = \sum_{\ell \in [w]} ([[\mathsf{sk}]]_b + \sum_{j_T \in S^1_{B_T}} [[r_{j_T}]]_{b \cdot 2^\sigma}) b^\ell$. We have that $[[\mathsf{sk}]]_b + [[\mathsf{sk}]]_b + \sum_{j_T \in S^1_{B_T}} [[r_{j_T}]]_{b \cdot 2^\sigma} \leq b + (t_R + 1) b \cdot 2^\sigma \leq_{\text{by assumption}} |\mathcal{P}_{pk}|$ thus $[[\mathsf{sk}]]_b + [[\mathsf{sk}]]_b + \sum_{j_T \in S^1_{B_T}} [[r_{j_T}]]_{b \cdot 2^\sigma}$

mod $|\mathcal{P}_{pk}| = [[\mathsf{sk}]]_b + [[\mathsf{sk}]]_b + \sum_{j_T \in S^1_{B_T}} [[r_{j_T}]]_{b \cdot 2^\sigma}$ over $\mathbb{Z}$. From this we can conclude that $\mathsf{sk}_{\mathsf{masked}} = \mathsf{sk} + r$ (This entire argument assumes that the plaintext space is a $\mathbb{Z}$-module, see definition A.1). Returning to our goal we get that

$$\begin{aligned}\sum_{j_R \in S_{B_R}} \Delta\lambda^{S_{B_R}}_{0,j_R} [\mathsf{sk}]_{j_R} &= \sum_{j_R \in S_{B_R}} \Delta\lambda^{S_{B_R}}_{0,j_R} \left(\Delta \mathsf{sk}_{\mathsf{masked}} - \sum_{j_T \in S^1_{B_T}} [r_{j_T}]_{j_R} \right) \\ &= \Delta^2 \mathsf{sk} + \Delta^2 r - \sum_{j_T \in S^1_{B_T}} \sum_{\ell \in [w]} \Delta^2 [[r_{j_T}]]_{b \cdot 2^\sigma} b^\ell \\ &= \Delta^2 \mathsf{sk} + \Delta^2 r - \Delta^2 r = \Delta^2 \mathsf{sk}\end{aligned}$$

Here the second equality comes from the correctness of the PVSS scheme. The computation of the verification keys is essentially the same but happens under the homomorphism of the commitment scheme.

Varying Threshold Simulation: Assume the existence of two straight line simulators $\mathcal{S}_{\mathsf{keygen}}$ and $\mathcal{S}_{\mathsf{TDec}}$ satisfying the following properties:

1. $\mathcal{S}_{\mathsf{keygen}}(\mathsf{pk})$ emulates a key generation protocol ending with a public key pk and verification keys vk_{B_T}. The verification keys are commitments of points on a t-degree polynomial where the free coefficient is sk, and sk corresponds to pk. In addition, malicious parties receive shares $[\mathsf{sk}]_{i_T}$ (indistinguishable from the shares in the real execution) corresponding to the verification keys. The simulator can extract the shares.
2. $\mathcal{S}_{\mathsf{TDec}}(\mathsf{pk}, \{\mathsf{vk}_{B_T}\}, \mathsf{ct}, \{[\mathsf{sk}]_{j_T}\}_{j_T \in U})$ that given a cipher text ct emulates a threshold decryption protocol ending with the correct decryption pt.

The simulation will thus work as follows:

1. Upon receiving a request for key generation, the simulator sends $(\mathsf{keygen}, \mathsf{pid}, \mathsf{sid})$ on behalf of the malicious parties to $\mathcal{F}_{\mathsf{DAHE}}$ and receives pk. It then uses $\mathcal{S}_{\mathsf{keygen}}(\mathsf{pk})$ to land on the correct key pk along with verification keys vk_{B_T} for $i_T \in B_T$. If the adversary aborts during the execution on behalf of a subset $U_{B_T,\mathsf{keygen}}$, the simulator sends $(\mathsf{continue}, U_{B_T,\mathsf{keygen}})$ to $\mathcal{F}_{DAHE}$.
2. Upon receiving a valid decryption request from $\mathcal{A}$ the simulator sends $(\mathsf{decrypt}, \mathsf{pid}, \mathsf{sid}, \mathsf{ct})$ to $\mathcal{F}_{\mathsf{DAHE}}$. Upon receiving the plaintext pt the simulator uses $\mathcal{S}_{\mathsf{TDec}}(\mathsf{pk}, \{\mathsf{vk}_{B_T}\}, \mathsf{ct}, \{[\mathsf{sk}]_{j_T}\}_{j_T \in U})$ to emulate the decryption protocol landing on pt as the decryption.
3. Upon receiving a reconfiguration request the simulator sends $\mathcal{F}_{\mathsf{DAHE}}$ $(\mathsf{reconfigure}, \mathsf{pid}, \mathsf{sid}, \Gamma)$ for the correspondent access structure. It then starts the emulation of the reconfiguration protocol as follows:
   (a) Randomize an honest party j^*_T pick random values for the shares for the malicious parties and this honest party (denoted by $[r_{i_T}]_{j_T}$). For the malicious parties compute their commitments, encryption and proofs honestly and set $C^{i_T,j^*_T}_{\mathsf{Share},R} = \mathsf{Com}_{\mathsf{pp}}([r_{i_T}]_{j^*_T}) + \mathsf{vk}$. Then generate the commitments on the coefficients $C^{i^*_T,\ell}_{\mathsf{Share},T}$ using interpolation in the exponent. In addition generate a fake proof for this party.
   (b) Act according to the real execution emulating $\mathcal{F}_{\mathsf{ACS}}$. Upon receiving a subset S_B, check if $j^* \in S^1_{B_T}$. If not, rewind.

(c) Use the secret keys sk_{j_T} to reconstruct $[r]_{i_T}$ and predict the value of the plaintext of $\mathsf{ct}_{\mathsf{masked\text{-}key}}$ $(= \sum_{i \in S^1_{B_T}} r_{i_T})$.

(d) Use $\mathcal{S}_{\mathsf{TDec}}$ to simulate threshold decryption landing on pt.

Indistinguishability: We argue that the distribution is indistinguishable via a sequence of hybrids.

- H_0 - the real execution.
- H_1 - same as H_0 but messages during the key generation are replaced by those generated by $\mathcal{S}_{\mathsf{keygen}}$.
- H_2 - same as H_1 but messages during threshold decryption are replaced by those generated by $\mathcal{S}_{\mathsf{TDec}}$.
- H_3 - same as H_2 but encryptions are replaced by encryptions of zeros.
- H_4 - same as H_3 but encryptions are replaced by the encryptions generated by $\mathcal{S}$.
- H_5 - the ideal world execution.

Hybrid H_0, H_1 and H_2 are equivalent from the indistinguishability of $\mathcal{S}_{\mathsf{keygen}}, \mathcal{S}_{\mathsf{TDec}}$. Hybrids H_2, H_3, and H_4 are indistinguishable from each other due to the ind-cpa and circular security assumption of the encryption scheme. Thus the remaining differences between H_4 to H_5 are i) the zk-proof, ii) the commitment of the distinguished party, and iii) the output distribution. Thus i) H_4, H_5 are indistinguishable by the zk-property of the zk proof, ii) the hiding property of the commitment scheme, and iii) the output distribution of the verification keys which is randomized the same way as in the real execution and satisfies the property that vk_{B_R} lies on the same t_T degree polynomial with the free coefficient sk.

We are now ready to move to the Constant Threshold Protocol, let us again start by describing the correctness of the protocol:

Correctness: We argue that $[\mathsf{sk}]_{i_R} = f(i_R) + g(i_R)$ where f is the original polynomial used for the sharing of sk used by the quorum B_T and g is a polynomial representing a sharing of 0.

$$
\begin{aligned}
\Delta^3 f(i_R) &= \Delta^3 \sum_{j_T \in S^5_{B_T}} \lambda^{S^5_{B_T}}_{i_R, j_T} [\mathsf{sk}]_{j_T} \\
&= \Delta^3 \left(\sum_{j_T \in S^5_{B_T}} \lambda^{S^5_{B_T}}_{i_R, j_T} ([\mathsf{sk}]_{j_T} + \sum_{j^*_T \in S^1_{B_T}} [r_{j^*_T}]_{j_T}) \right) - \Delta^3 \sum_{j_T \in S^5_{B_T}} \lambda^{S^5_{B_T}}_{i_R, j_T} \sum_{j^*_T \in S^1_{B_T}} [r_{j^*_T}]_{j_T} \\
&= \Delta^2 \left(\sum_{j_T \in S^5_{B_T}} \Delta \lambda^{S^5_{B_T}}_{i_R, j_T} ([\bar{\mathsf{sk}}]_{j_T}) \right) - \Delta^3 \sum_{j_T \in S^5_{B_T}} \lambda^{S^5_{B_T}}_{i_R, j_T} \sum_{j^*_T \in S^1_{B_T}} [r_{j^*_T}]_{j_T} \\
&= \Delta^2 \left(\sum_{j_T \in S^5_{B_T}} \lambda^{S^5_{B_T}}_{i_R, j_T} (\sum_{j^*_T \in S^3_{B_T}} \lambda^{S^3_{B_T}}_{0, j^*_T} [[\bar{\mathsf{sk}}]_{j_T}]_{j^*_T}) \right) - \Delta^3 \sum_{j_T \in S^3_{B_T}} \lambda^{S^3_{B_T}}_{i_R, j_T} \sum_{j^*_T \in S^1_{B_T}} [r_{j^*_T}]_{j_T} \\
&= \Delta^2 \left((\sum_{j^*_T \in S^3_{B_T}} \lambda^{S^3_{B_T}}_{0, j^*_T} \sum_{j_T \in S^5_{B_T}} \lambda^{S^5_{B_T}}_{i_R, j_T} [[\bar{\mathsf{sk}}]_{j_T}]_{j^*_T}) \right) - \Delta^3 \sum_{j^*_T \in S^1_{B_T}} \sum_{j_T \in S^3_{B_T}} \lambda^{S^3_{B_T}}_{i_R, j_T} [r_{j^*_T}]_{j_T} \\
&= \Delta^3 [\mathsf{sk}]_{i_R} + \sum_{j^*_T \in S^1_{B_T}} [r_{j^*_T}]_{i_R} - \Delta^3 \sum_{j^*_T \in S^1_{B_T}} \sum_{j_T \in S^3_{B_T}} \lambda^{S^3_{B_T}}_{i_R, j_T} [r_{j^*_T}]_{j_T}
\end{aligned}
$$

This indeed gives $f(i_R) = [\mathsf{sk}]_{i_R} + g'(i_R) - g(i_R)$ where $g = \sum_{j^* \in S^1_{B_T}} g_{j^*_T}, g' = \sum_{j^*_T \in S^1_{B_T}} g'_{j^*}$ and $g_{j^*_T}(j) = [r_{j^*_T}]_{j_T}, g'_{j^*_T}(j_T) = [r_{j^*_T}]_{j_T}$. In particular $g_{j^*_T}(0) = g'_{j^*_T}(0) = r_{j^*_T}$ which gives that $g(0) = g'(0) = \sum_{j^*_T \in S^1_{j_T}} r_{j^*_T} = r$. Thus interpolation over any subset will give $\mathsf{sk} + r - r = \mathsf{sk}$ as needed. In terms of the size of the shares it is at most tripled from the size of the previous share.

Constant Threshold Simulation: The simulator is similar to the simulator for the varying threshold case. Steps (1) and (2) of the simulators are identical (i.e. the simulation of the Reconfiguration itself). The simulator then:

3. Set $[\bar{\mathsf{sk}}]_{i_T} = \sum_{j_T \in S^1_{B_T}} [r_{j_T}]_{i_T}$ and as the public parameters it sets $\mathsf{vk}_{i_T} + \sum_{j_T \in S^1_{B_T}} C^{j_T, i_T}_{\mathsf{Share}, T}$. It then cheats in the zk proof $\pi^{i_T}_{\mathsf{Share}, T}$.
4. It continues the rest of the protocol similarly to honest parties.

Indistinguishability: The indistinguishability argument is similar to the varying threshold case up to the transition between H_4 and H_5. In this case, the differences are: i) the ZK proofs, ii) the commitments of the free coefficient of the polynomial, and iii) the distribution of the output. The zk-proofs and commitments are indistinguishable as before. In the real execution the parties received $f(j_R) + g(j_R)$ where $f(0) = \Delta \mathsf{sk}, g(0) = 0$ while in the simulated execution they get $f_{\mathcal{S}}(j_R) + g_{\mathcal{S}}(j_R)$. The adversary may now hold $\{f(j_T)\}_{j_T \in U \cap B_T}$ and $\{f(j_R) + g(j_R)\}_{j_R \in U \cap B_R}$ both are indistinguishable from the values in the ideal execution based on the statistical security of Shamir Secret Sharing Over the Integers. Thus the output is indistinguishable.

E Performance Full Data

We thereby give the full experimental data for DKG, presign and sign in Table 2, Table 3 and Table 4 respectively.

Encryption Scheme	Number of Tangible Parties	Nodes Running Time (ms)	Client Running Time (ms)
Class Groups	3	309	6
	5	368	4
	10	443	7
	30	720	7
	50	1154	5
	100	1727	5
	150	2357	4
Paillier	3	233	7
	5	202	6
	10	253	5
	30	592	4
	50	987	8
	100	1600	5
	150	2368	3

Table 2: Key Generation Running Time by Encryption Scheme and Number of Tangible Parties

F Instantiating TAHE

In this section we discuss some of the approaches for instantiating the functionality $\mathcal{F}_{\mathsf{TAHE}}$ A.5. In particular we discuss the currently most well known options for TAHE based on Paillier, Class Groups and lattices. In addition we discuss some important optimizations in the case of decryption of signatures.

F.1 The Scheme

Using [FMM$^+$23] for threshold Paillier one can see that the threshold decryption is naturally asynchronous. Unfortunately the known methods for generation RSA modulus in MPC utilize additive secret sharing and thus are synchronous [CHI$^+$21]. This means that during key generation one must give up on asynchronicity. This can be mitigated by having a one time generation of the RSA modulus and then for future key generation use an ala-ElGamal variant [BG10]. For the classical Paillier scheme [FMM$^+$24] show one can achieve secure function evaluation for valid ciphertext. As shown in the original paper [Pai99] every element is a valid ciphertext and thus it is enough to satisfy our stronger definition.

For Class Groups based encryption as presented in [BDO23] the threshold decryption phase is still asynchronous. As for the key generation while it is presented in an synchronous manner it can be easily transformed to the asynchronous setting. We

Encryption Scheme	**Number of Tangible Parties**	**Nodes Running Time (ms)**
Class Groups	3	1480
	5	1585
	10	1949
	30	3070
	50	4775
	100	6558
	150	8501
Paillier	3	652
	5	729
	10	1086
	30	2629
	50	3830
	100	6899
	150	10578

Table 3: Pre-Sign Running Time by Encryption Scheme and Number of Tangible Parties

Encryption Scheme	**Total Voting Power**	**Nodes Running Time (ms)**	**Client Running Time (ms)**
Class Groups	3	1350	500
	9	2132	491
	30	3418	492
	90	4362	491
	150	15994	501
	300	18649	504
Paillier	3	292	649
	9	515	649
	30	536	652
	90	493	654
	150	454	659
	300	1063	670

Table 4: Sign Protocol Running Time by Encryption Scheme and Total Voting Power

present such transformation in protocol F.1 which also works for the ala ElGamal Pailiier variant. The simulation utilizes that in such encryption schemes linear key bias is proven to not hurt security [BCD+24] similarly to our technique for ECDSA. Another issue that arises is that the scheme demands honest majority, while this may look like a moot point as our protocol assumes $t < n/3$. But practically speaking one may still want to take the threshold of for the encryption scheme to be larger as the attacks based on breaking the threshold scheme are much simpler (just collecting the share), while typical attacks on the asynchronous protocol which cause us to choose $t < n/3$ demand "splitting" of the network which may be practically a much harder attack to preform. That being said following the proofs in [BDO23] the honest majority is needed for the complaint mechanism of the VSS and the guaranteed output delivery and not for the basic security of the encryption scheme. Thus using a PVSS instead of a VSS and giving up on guaranteed output delivery may allow one to use the scheme with any threshold. Secure function evaluation works as well although as shown in Lemma F.1.

Lattice based encryption is much more complex as it it's threshold decryption is typically synchronous [MBH23], while asynchronous solutions exists [BGG+18] they suffer from poor performance especially in regards to communication complexity. As for key generation we are unaware of any works providing an asynchronous solution. In addition it does not naturally admit secure linear evaluation although recent work shows that at least for BGV it is achievable [CNS23]. Otherwise one have to use multiple bootstrapping [Klu22] or resort to the classical approach of noise flooding. Note that these works assume proofs of correct evaluation are given which we would like to avoid in our framework.

The main advantage of using lattice based encryption is that you may also allow low level multiplication depth which will allow the presigns in the ECDSA protocol to be used for every user multiplying to get an encryption of kx.

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 evalution:

Lemma F.1 *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 F.1 .

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 request 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 [FMM+24] 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 preforms 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 [BBHP22] 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 effect performance significantly. On the contrast in system will small number of participants but high network latency it may be better for each party to locally recombine the decryption shares. We remark that this optimization does not work if Thus the distributed decryption looks as follows:

1. Generate the decryptions shares ds_i and proofs π^i_{ds} 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 it's validity, if it is not valid the consider the leader malicious. Upon receiving a message cheaters they verifies the proofs for the decryption shares and consider any party for which the proofs failed malicious.

PROTOCOL C.3 (*Sign* $\Pi^{ECDSA}_{\mathsf{Sign}}$: $\mathsf{Sign}(\mathbb{G}, G, H, q, sid, \Gamma_B, pk, X, X_A, ct_{key}, pres_X, msg)$)

The protocol is parameterized with an ECDSA group description $(\mathbb{G}, G, q)$ along with another generator H for Pedersen commitments, an access structure Γ_B, a unique session identifier sid, a TAHE public key pk, the DKG protocol output $(X, X_A, \mathsf{ct}_{\mathsf{key}})$, the pre-sign protocol output with corresponding sid, $\mathsf{pres}_{X,\mathsf{sid}} = (\mathsf{ct}_\gamma, \mathsf{ct}_{\gamma\cdot\mathsf{key}}, \mathsf{ct}_{\gamma\cdot k_0}, \mathsf{ct}_{\gamma\cdot k_1}, R_{B,0}, R_{B,1})$, and the message to be signed msg. The protocol interacts with a centralized party A_{pid_A} and a set of parties $B = \{B_i\}_{i\in[n]}$. The parties output a valid ECDSA signature $\sigma = (r, s)$ with verification key X.

1. A_{pid_A} **does as follows**:
   (a) Call the random oracle $\mathcal{H}$ on msg and receive m.
   (b) 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. $C_{kx} = \mathsf{Pedersen.Com}_{X_A,H}(k_A; \rho_3)$
   (c) Run the protocols $\Pi^{L_{\mathsf{Dcom}}[C_k,H]}_{\mathsf{zk}}(G; k_A^{-1}, -k_A^{-1}\rho_0)$, $\Pi^{L_{\mathsf{Dcom}}[C_\alpha,H]}_{\mathsf{zk}}(G; (\alpha^{-1}, -\alpha^{-1}\rho_1)$ and $\Pi^{L_{\mathsf{Dcom}}[G,H]}_{\mathsf{zk\text{-}uc}}(C_\beta; \beta, \rho_2)$ generating proofs π_k and π_α, π_β.
   (d) Call the random oracle $\mathcal{H}(\mathsf{sid}, \mathsf{msg}, \mathbb{G}, G, q, H, X, \mathsf{pres}_{X,\mathsf{sid}}, C_k, C_{kx}, X_A, C_\alpha, C_\beta, \pi_k, \pi_\alpha, \pi_\beta)$, and receives $\mu_k^0, \mu_k^1, \mu_k^G$. It then computes:
      – $\mathsf{ct}_{\gamma\cdot k} = (\mu_k^0 \odot \mathsf{ct}_{\gamma\cdot k_0}) \oplus (\mu_k^1 \odot \mathsf{ct}_{\gamma\cdot k_1}) \oplus \mu_k^G$
      – $R'_B = (\mu_k^0 R_{B,0}) + (\mu_k^1 \cdot R_{B,1}) + \mu_k^G \cdot G$
   (e) 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 $\gamma \cdot k_A(m + rx)$
   (f) Generate the following proofs:
      i. $\pi_{kx} \leftarrow \Pi^{L_{\mathsf{DcomEq}}[(G,H),(X_A,H)]}_{\mathsf{zk}}(C_k, C_{kx}; k_A, \rho_0, \rho_3)$
      ii. $\pi_R \leftarrow \Pi^{L_{\mathsf{DComDL}}[G,H,(\mathbb{G},R,q)]}_{\mathsf{zk}}(C_k, R_B; k_A, \rho_0)$
      iii. $\pi_{R_B} \leftarrow \Pi^{L_{\mathsf{VecDComDL}}[(G,H),(\mathbb{G},(R'_B,G),q)]}_{\mathsf{zk}}(C_\alpha, C_\beta), R_B; (\alpha, \beta), \rho_1, \rho_2)$
      iv. $\pi_{\mathsf{ct}_{\alpha,\beta}} \leftarrow \Pi^{L_{\mathsf{DComEval}}[G,H,\mathsf{pk},(\mathsf{ct}_\gamma,\mathsf{ct}_{\gamma\cdot k},(\mathbb{G},G,q))]}_{\mathsf{zk}}(\mathsf{ct}_{\alpha,\beta}, (C_\beta, C_\alpha); (\beta, \alpha), \rho_0, \rho_1, \eta_0)$
      v. $\pi_{\mathsf{ct}_A} \leftarrow \Pi^{L_{\mathsf{DComEval}}[G,H,\mathsf{pk},(\mathsf{ct}_\gamma,\mathsf{ct}_{\gamma\cdot\mathsf{key}}),(\mathbb{G},G,q)]}_{\mathsf{zk}}(\mathsf{ct}_A, (C_1, C_2); (a_1, a_2), \rho_3 r + \rho_0 m, r\rho_0, \eta_1)$
   (g) Send $(\mathsf{broadcast}, \mathsf{sid}, (R, R_B, C_k, C_\alpha, C_\beta, C_{kx}, \mathsf{ct}_A, \mathsf{ct}_{\alpha,\beta}, \pi_k, \pi_\alpha, \pi_\beta, \pi_{kx}, \pi_R, \pi_{R_B}, \pi_{\mathsf{ct}_{\alpha,\beta}}, \pi_{\mathsf{ct}_A}))$ to $\mathcal{F}_{\mathsf{broadcast}}$.
2. **Each** B_i **does as follows**:
   - **Round 1**:
   (a) Call the random oracle $\mathcal{H}(\mathsf{msg})$ and receives m.
   (b) 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 $\mathcal{F}_{\mathsf{broadcast}}$.
   (c) Call the random oracle $\mathcal{H}(\mathsf{sid}, \mathsf{msg}, \mathbb{G}, G, q, H, X, \mathsf{pres}_{X,\mathsf{sid}}, C_k, C_{kx}, X_A, C_\alpha, C_\beta, \pi_k, \pi_\alpha, \pi_\beta)$ and compute:
      i. $\mathsf{ct}_{\gamma\cdot k} = (\mu_k^0 \odot \mathsf{ct}_{\gamma\cdot k_0}) \oplus (\mu_k^1 \odot \mathsf{ct}_{\gamma\cdot k_1}) \oplus \mu_k^G$
      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. $C_2 = r \odot C_k$
   (d) Verify the proofs. If fail abort and consider A_{pid_A} malicious.
   (e) 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 $\mathcal{F}^{\Gamma_B}_{\mathsf{global\text{-}broadcast}}$
3. **Output**:
   (a) A_{pid_A} receives $(\mathsf{global\text{-}broadcast}, \mathsf{sid}, (r, s))$ from $\mathcal{F}^{\Gamma_B}_{\mathsf{global\text{-}broadcast}}$.
   (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).

PROTOCOL C.4 (*Schnorr Based Pre-sign* $\Pi_{\mathsf{pres}}^{\mathsf{Schnorr}}$: $\mathsf{SchPresign}(\mathbb{G}, G, q, \Gamma_B, \mathit{sid}, \mathit{pk})$)

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, a 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 $k_0^i, k_1^i \leftarrow \mathbb{Z}_q$ and $\eta_0^i, \eta_1^i \leftarrow \mathcal{R}_{pk}$ and computes:
      i. $K_{B,0}^i = k_0^i \cdot G$
      ii. $K_{B,1}^i = k_1^i \cdot G$
      iii. $\mathsf{ct}_{k_0}^i = \mathsf{AHE.Enc}(\mathsf{pk}, k_0^i; \eta_0^i)$
      iv. $\mathsf{ct}_{k_1}^i = \mathsf{AHE.Enc}(\mathsf{pk}, k_1^i; \eta_0^i)$
   (b) Run $\Pi_{\mathsf{agg}}^{L_{\mathsf{EncDL}}[\mathsf{pk},(\mathbb{G},G,q)]}$ on inputs $(\mathsf{ct}_{k_1}^i, K_{B,0}^i; k_0^i, \eta_0^i)$ and $(\mathsf{ct}_{k_1}^i, K_{B,1}^i; k_1^i, \eta_1^i)$ 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 C.5 (*Schnorr Based Sign* $\Pi_{\mathsf{sign}}^{\mathsf{Schnorr}}$: $\mathsf{SchSign}(\mathbb{G}, G, q, \mathit{sid}, \mathit{pk}, X, \mathit{ct}_{\mathit{key}}, \mathit{pres}_X)$)

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, a encryption key to a TAHE scheme pk, a verification key X, an encryption of the share of the distributed party of the signing key $\mathsf{ct}_{\mathsf{key}}$ and the output of a unique pre-sign $\mathsf{pres}_X = (K_{B,0}, K_{B,1})$. 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 $\mathsf{ct}_B = \mathsf{ct}_k \oplus (e \cdot \mathsf{ct}_{\mathsf{key}}) \oplus z_A$.
   (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.

PROTOCOL C.6 (*Reconfiguration with varying threshold* $\Pi^{\textit{var-th}}_{\textit{ReConfigure}}$: $\textit{ReConfigure}(\textit{TAHE}, \textit{PVSS}, 1^\kappa, 1^\sigma, t_T, t_R, n_T, n_R)$)

The protocol interacts with a transferring quorum $B_T = \{B_T^{i_T}\}_{i_T \in [n_T]}$ and a receiving quorum $B_R = \{B_R^{i_R}\}_{i_R \in [n_R]}$. B_T holds a t_T-out-of-n_T secret sharing $[\mathsf{sk}]^i_{B_T}$ of the secret decryption key. Both quorums hold the corresponding public key pk, the verification keys of B_T denoted $\mathsf{vk}_{B_T} = \{\mathsf{vk}^{i_T}_{B_T}\}_{i_T \in [n_T]}$ and public keys of an AHE scheme used for the PVSS $\mathsf{pk}_{B_T} = \{\mathsf{pk}^{i_T}_{B_T}\}_{i_T \in [n_T]}, \mathsf{pk}_{B_R} = \{\mathsf{pk}^{i_R}_{B_R}\}_{i_R \in [n_R]}$ along with an encryption of the secret key $\mathbf{ct}_{\mathsf{sk}} = \mathsf{Enc}^b_{\mathsf{pk}}(\mathsf{sk}; \vec{\eta})$. The protocol outputs a fresh t_R-out-of-n_R secret shares $[\mathsf{sk}]^{i_R}_{B_T}$ on the secret key sk held by B_R and public verifications keys $\mathsf{vk}_{B_R} = \{\mathsf{vk}^{i_R}_{B_R}\}_{i_R \in [n_R]}$ corresponding to the shares.

1. $B_{i_T} \in B_T$ **does as follows:**
 - **Round 1**:
   (a) Sample a mask $r_{i_T} \leftarrow [-D(\mathcal{K}_\kappa) \cdot 2^{\sigma \cdot w}, D(\mathcal{K}_\kappa) \cdot 2^{\sigma \cdot w}]$.
   (b) Calculate $\mathbf{ct}^{i_T}_{\mathsf{mask}} = \mathsf{Enc}^{b \cdot 2^\sigma}_{\mathsf{pk}}(r_{i_T}; \vec{\eta}_{i_T})$ along with a commitment $\mathbf{C}^{i_T}_{\mathsf{mask}} = \mathsf{Com}([[r_{i_T}]]_{b \cdot 2^\sigma})$. In addition generate proof $\pi^{i_T}_{\mathsf{mask},T} \leftarrow \Pi_{\mathsf{zk}}^{L_{\mathsf{EncDL}}[\mathsf{pk}, b \cdot 2^\sigma]}(\mathbf{ct}^{i_T}_{\mathsf{mask}}, \mathbf{C}^{i_T}_{\mathsf{mask}}; r_{i_T}, \vec{\eta}_{i_T})$.
   (c) $(\{\mathbf{ct}^{i_T \to j_R}_{\mathsf{Share},R}\}_{j_R \in [n_R]}, \{C^{i_T,\ell}_{\mathsf{Share},R}\}_{\ell \in [t_R]}, \pi^{i_T}_{\mathsf{Share},R}) \leftarrow \mathsf{Dist}(\mathsf{pk}_{B_R}, \mathsf{Pack}_b([[r_{i_T}]]_{b \cdot 2^\sigma}))$.
   (d) Send $(\mathsf{broadcast}, \mathsf{sid}, i_T, (\mathbf{ct}^{i_T}_{\mathsf{mask}}, \mathbf{C}^{i_T}_{\mathsf{mask}}, \pi^{i}_{\mathsf{mask}}, \{\mathbf{ct}^{i_T \to j_R}_{\mathsf{Share},R}\}_{j_R \in [n_R]}, \{C^{i_T,\ell}_{\mathsf{Share},R}\}_{\ell \in [1,t_R]}\}_{\ell \in [t_R]}, \pi^{i_T}_{\mathsf{Share},R}))$ to $\mathcal{F}_{\mathsf{broadcast}}$.
 - **Round 2**:
   (a) Receive $(\mathsf{broadcast}, \mathsf{sid}, j, (\mathbf{ct}^j_{\mathsf{mask}}, \pi^j_{\mathsf{mask}}, \{\mathbf{ct}^{j \to j_R}_{\mathsf{Share},R}\}_{j_R \in [n_R]}, \{C^{j,\ell}_{\mathsf{Share},R}\}_{\ell \in [t_R]}\}_{\ell \in [1,t_R]}, \pi^j_{\mathsf{Share},R})$ from $\mathcal{F}_{\mathsf{broadcast}}$.
   (b) Set $C^{j,0}_{\mathsf{Share},R} = \mathsf{Pack}_{b \cdot 2^\sigma}(\mathbf{C}^j_{\mathsf{mask}})$. If any of them fail consider j malicious. Else send $(\mathsf{validate}, \mathsf{sid}, i_T, j)$ to $\mathcal{F}^{\binom{[n]}{t+1}_B}_{\mathsf{ACS}}$.
 - **Round 3**:
   (a) Receive $S^1_{B_T}$ from $\mathcal{F}_{\mathsf{ACS}}$ and set $\mathbf{ct}_{\mathsf{masked\text{-}key}} = \mathbf{ct}_{\mathsf{sk}} \oplus \sum_{j_T \in S^1_{B_T}} \mathbf{ct}^{j_T}_{\mathsf{mask}}$.
   (b) Set $(\mathbf{ds}_{i_T}, \pi^{i_T}_{\mathsf{ds}}) \leftarrow \mathsf{TAHE.TDec}_{\mathsf{pk}}(\mathsf{ct}_{\mathsf{masked\text{-}key}}, \mathsf{vk}^{i_T}_{B_T}; [\mathsf{sk}]^{i_T}_{B_T})$ and send $(\mathsf{broadcast}, \mathsf{sid}, i_T, (\mathbf{ds}_{i_T}, \pi^{i_T}_{\mathsf{ds}}))$ to $\mathcal{F}_{\mathsf{broadcast}}$.
 - **Broadcast Round**:
   (a) Upon receiving $(\mathsf{broadcast}, \mathsf{sid}, j, (\mathsf{ds}_j, \pi^j_{\mathsf{ds}}))$ verify that the proof π^j_{ds} is valid, if it fails consider j malicious.
   (b) Collect a subset $S^{3,i_T}_{B_T} \in \binom{[n]}{t+1}$ with validated proofs and calculate $\mathsf{sk}_{\mathsf{masked}} = \mathsf{TAHE.Rec}^b_{\mathsf{pk}}(\{\mathbf{ds}_{j_T}\}_{j_T \in S^{3,i_T}_{B_T}})$.
   (c) Calculate $\mathsf{vk}^{j_R}_{B_R} = \mathsf{Com}_(\Delta \mathsf{sk}_{\mathsf{masked}}) - \sum_{i_T \in S^1_{B_T}} \sum_{\ell \in [t_R]} C^{j,\ell}_{\mathsf{Share},R} \odot j^\ell_T$.
   (d) Send $(\mathsf{global\text{-}broadcast}, \mathsf{sid}, i, (\mathsf{sk}_{\mathsf{masked}}, \{\{\mathbf{ct}^{j_T \to j_R}_{\mathsf{Share},R}\}_{j_T \in S^1_{B_T}}, \mathsf{vk}^{j_R}_{B_R}\}_{j \in S_B}\}_{j_R \in [n_R]})$ to $\mathcal{F}_{\mathsf{global\text{-}broadcast}}$.
2. $B_{i_R} \in B_R$ **does as follows:**
   (a) Upon receiving $(\mathsf{sk}_{\mathsf{masked}}, \{\{\mathbf{ct}^{j_T \to j_R}_{\mathsf{Share},R}\}_{j_T \in S^1_{B_T}}, \mathsf{vk}^{j_R}_{B_R}\}_{j \in S_B}\}_{j_R \in [n_R]})$ from $\mathcal{F}_{\mathsf{global\text{-}broadcast}}$ set $[r_{j_T}]_{i_R} \leftarrow \mathsf{Dec}(\mathsf{sk}_{i_T}, \mathbf{ct}^{j_T \to i_R}_{\mathsf{Share},R})$.
   (b) Calculate $[\mathsf{sk}]^{i_R}_R = \Delta \mathsf{sk}_{\mathsf{masked}} - \sum_{j_T \in S^1_{B_T}} [r_{j_T}]_{i_R}$
   (c) Record $([\mathsf{sk}]^{i_T}_{B_R}, \{\mathsf{vk}^{j_R}_{B_R}\}_{j_R \in [n_R]})$.

PROTOCOL C.7 (*Reconfiguration with Constant Threshold-* $\Pi^{\mathit{const\text{-}th}}_{\mathit{reconfigure}}$: *reconfigure*($\mathit{TAHE}, \mathit{PVSS}, 1^\kappa, t, n_T, n_R$))

The protocol interacts with a transferring quorum $B_T = \{B_T^{i_T}\}_{i_T \in [n_T]}$ and a receiving quorum $B_R = \{B_R^{i_R}\}_{i_R \in [n_R]}$. B_T holds a t-out-of-n_T secret sharing $[\mathsf{sk}]^i_{B_T}$ of the secret decryption key. Both quorums hold the corresponding public key pk, the verification keys of B_T denoted $\mathsf{vk}_{B_T} = \{\mathsf{vk}^{i_T}_{B_T}\}_{i_T \in [n_T]}$ and public keys of an AHE scheme used for the PVSS $\mathsf{pk}_{B_T} = \{\mathsf{pk}^{i_T}_{B_T}\}_{i_T \in [n_T]}, \mathsf{pk}_{B_R} = \{\mathsf{pk}^{i_R}_{B_R}\}_{i_R \in [n_R]}$. The protocol outputs a fresh t-out-of-n_R secret sharing $[\mathsf{sk}]^{i_T}_{B_R}$ of sk, with corresponding verification keys $\mathsf{vk}_{B_R} = \{\mathsf{vk}^{i_T}_{B_R}\}_{i_T \in [n_T]}$.

1. $B_T^{i_T} \in B_T$ **does as follows**:
- **Round 1**:
  (a) Sample $r_{i_T} \leftarrow \mathcal{K}_\kappa$.
  (b) $(\{\mathbf{ct}^{i_T \to j_T}_{\mathsf{Share},T}\}_{j_T \in [n_T]}, \{C^{i_T,\ell}_{\mathsf{Share},T}\}_{\ell \in [t]}, \pi^{i_T}_{\mathsf{Share},T}) \leftarrow \mathsf{Dist}(\mathsf{pk}_{B_T}; r_{i_T})$.
  (c) $(\{\mathbf{ct}^{i_T \to j_R}_{\mathsf{Share},R}\}_{j_R \in [n_R]}, \{C^{i_T,\ell}_{\mathsf{Share},R}\}_{\ell \in [t]}, \pi^{i_T}_{\mathsf{Share},R}) \leftarrow \mathsf{Dist}(\mathsf{pk}_{B_R}; r_{i_T})$.
  (d) Send $(\mathsf{broadcast}, \mathsf{sid}, i_T, ((\{\mathbf{ct}^{i_T \to j_T}_{\mathsf{Share},T}\}_{j_T \in [n_T]}, \{C^{i_T,\ell}_{\mathsf{Share},T}\}_{\ell \in [t]}, \pi^{i_T}_{\mathsf{Share},T},$ $\{\mathbf{ct}^{i_T \to j_R}_{\mathsf{Share},T}\}_{j_R \in [n_R]}, \{C^{i_T,\ell}_{\mathsf{Share},R}\}_{\ell \in [1,t]}, \pi^{i_T}_{\mathsf{Share},R})$ to $\mathcal{F}_{\mathsf{broadcast}}$.
- **Round 2**: Receive $(\mathsf{broadcast}, \mathsf{sid}, j, ((\{\mathbf{ct}^{j \to j_T}_{\mathsf{Share},T}\}_{j_T \in [n_T]}, \{C^{j,\ell}_{\mathsf{Share},T}\}_{\ell \in [t]}, \pi^{j}_{\mathsf{Share},T},$ $\{\mathbf{ct}^{j \to j_R}_{\mathsf{Share},T}\}_{j_R \in [n_R]}, \{C^{j,\ell}_{\mathsf{Share},R}\}_{\ell \in [1,t]}, \pi^{j}_{\mathsf{Share},R}))$
  (a) Set $C^{j,0}_{\mathsf{Share},R} = C^{j,0}_{\mathsf{Share},T}$ and verify the proofs. If any of the checks fail consider j malicious, Else send $(\mathsf{validate}, \mathsf{sid}, \mathsf{pid}_{i_T})$ to $\mathcal{F}_{\mathsf{ACS}}^{\binom{[n]}{t+1}_{B_T}}$.
- **Round 3**: Receive $(\mathsf{authorized\text{-}subset}, \mathsf{sid}, S^1_{B_T})$
  (a) For every $j_T \in S^1_{B_T}$ set $[r_{j_T}]_{i_T} = \mathsf{Dec}(\mathsf{sk}_{i_T}, \mathsf{ct}^{j_T \to i_T}_{\mathsf{Share},T})$. Calculate $[r_{S^1_{B_T}}]_{i_T} = \sum_{j_T \in S^1_{B_T}} [r_{j_T}]_{i_T}$ and $[\bar{\mathsf{sk}}]_{i_T} = [\mathsf{sk}]_{i_T} + [r_{S^1_{B_T}}]_{i_T}$.
  (b) $(\{\bar{\mathbf{ct}}^{i_T \to j_T}_{\mathsf{Share},T}\}_{j_T \in [n_T]}, \{\bar{C}^{i_T,\ell}_{\mathsf{Share},T}\}_{\ell \in [t]}, \bar{\pi}^{i_T}_{\mathsf{Share},T}) \leftarrow \mathsf{Dist}(\mathsf{pk}_{B_T}; [\bar{\mathsf{sk}}]_{i_T})$
  (c) Send $(\mathsf{broadcast}, \mathsf{sid}, i_T, (\{\bar{\mathbf{ct}}^{i_T \to j_T}_{\mathsf{Share},T}\}_{j_T \in [n_T]}, \{\bar{C}^{i_T,\ell}_{\mathsf{Share},T}\}_{\ell \in [1,t]}, \bar{\pi}^{i_T}_{\mathsf{Share},T}))$.
- **Round 4**: Receive $(\mathsf{broadcast}, \mathsf{sid}, j, (\{\bar{\mathbf{ct}}^{j \to j_T}_{\mathsf{Share},T}\}_{j_T \in [n_T]}, \{\bar{C}^{j,\ell}_{\mathsf{Share},T}\}_{\ell \in [1,t]}, \bar{\pi}^{j}_{\mathsf{Share},T}))$
  (a) Set $\bar{C}^{j,0}_{\mathsf{Share},T} = \mathsf{vk}_j + \sum_{j_T \in S^1_{B_T}} \sum_{\ell \in [t]} C^{j,\ell}_{\mathsf{Share},T} \odot j_T^\ell$. Verify the proofs $\bar{\pi}^{j}_{\mathsf{Share},T}$.
  If the check fails consider j malicious. Else send $(\mathsf{validate}, \mathsf{sid}, i_T)$ to $\mathcal{F}_{\mathsf{ACS}}^{\binom{[n]}{t+1}_{B_T}}$.
- **Round 5**: Receive $(\mathsf{authorized\text{-}subset}, \mathsf{sid}, S^3_{B_T})$ from $\mathcal{F}_{\mathsf{ACS}}$.
  (a) Set $[[\bar{\mathsf{sk}}]_{j_T}]_{i_T} \leftarrow \mathsf{Dec}(\mathsf{sk}_{i_T}, \bar{ct}^{j_T \to i_T}_{\mathsf{Share},T})$ for $j_T \in S^3_{B_T}$.
  (b) Set $[[\bar{\mathsf{sk}}]_{j_R}]_{i_T} = \sum_{j_T \in S^3_{B_T}} \Delta\lambda^{S^3_{B_T}}_{j_R,j_T} [[\bar{\mathsf{sk}}]_{j_T}]_{i_T}$ for $j_R \in [n_R]$.
  (c) Set $\bar{\mathsf{ct}}^{i_T \to j_R}_{\mathsf{Share},R} = \mathsf{Enc}_{\mathsf{pk}_{j_T}}([[\bar{\mathsf{sk}}]_{j_R}]_{i_T}, \eta_{i_T,j_R})$ along with a zk proof $\bar{\pi}^{i_T \to j_R}_{\mathsf{Share},R}$ of encryption of decommitment regarding $\bar{\mathsf{ct}}^{i_T \to j_R}_{\mathsf{Share},R}, \bar{C}^{j_R,i_T}_{\mathsf{Share},R}, [[\bar{\mathsf{sk}}]_{j_R}]_{i_T}, \eta_{i_T}, j_R$ where $\bar{C}^{j_R,i_R}_{\mathsf{Share},R} = \mathsf{Com}([[\bar{\mathsf{sk}}]_{j_R}]_{i_T})$
  (d) Send $(\mathsf{broadcast}, \mathsf{sid}, i_T, \{\bar{\mathsf{ct}}^{i_T \to j_R}_{\mathsf{Share},R}, \bar{\pi}^{i_T \to j_R}_{\mathsf{Share},R})\}_{j_R \in [n_R]}$ to $\mathcal{F}_{\mathsf{broadcast}}$.
- **Round 6**: Upon receiving a message from party j calculate $\bar{C}^{j_T,j}_{\mathsf{Share},T} = \sum_{\ell \in [t]} \bar{C}^{j_T,\ell}_{\mathsf{Share},T} \odot j^\ell$ and $\bar{C}^{j_R,j}_{\mathsf{Share},R} = \sum_{j_T \in S^3_{B_T}} \Delta\lambda^{S^3_{B_T}}_{j_R,j_T} \bar{C}^{j_T,j}_{\mathsf{Share},T}$ for $j_R \in [n_R]$ use $\bar{C}^{j_R,j}_{\mathsf{Share},R}$ as the commitment part of the zk statements. Then work similarly to round (2) to agree on a valid subset. $S^5_{B_T}$.
- **Broadcast Round**: Receive $(\mathsf{authorized\text{-}subset}, \mathsf{sid}, S^5_{B_T})$ from $\mathcal{F}_{\mathsf{ACS}}$.
  (a) Calculate $\mathsf{vk}^{j_R}_{B_R} = \sum_{j_T \in S^5_{B_T}} \Delta\lambda^{S^5_{B_T}}_{0,j_T} \bar{C}^{j_R,j_T}_{\mathsf{Share},R} - \Delta^3 \cdot (\sum_{j_T \in S^1_{B_T}} C^{j_T,j_R}_{\mathsf{Share},R})$.
  (b) Send $(\mathsf{global\text{-}broadcast}, \mathsf{sid}, (\{\mathsf{ct}^{j_T \to j_R}_{\mathsf{Share},R}\}_{i_T \in S^1_{B_T}} \{\bar{\mathsf{ct}}^{j_T \to j_R}_{\mathsf{Share}}\}_{j_T \in S^5_{B_T}}, \mathsf{vk}^{j_R}_{B_R}\}_{j_R \in [n_R]}))$ to $\mathcal{F}_{\mathsf{global\text{-}broadcast}}$.
2. $B_T^{i_T} \in B_R$ **Does as follows**: Receive $(\mathsf{global\text{-}broadcast},$ $(\{\mathsf{ct}^{j_T \to j_R}_{\mathsf{Share},R}\}_{i_T \in S^1_{B_T}} \{\bar{ct}^{j_T \to j_R}_{\mathsf{Share}}\}_{j_T \in S^5_{B_T}}, \mathsf{vk}^{j_R}_{B_R}\}_{j_R \in [n_R]}))$.
  (a) For every $j_T \in S^1_{B_T}$ set $[r_{j_T}]_{i_R} = \mathsf{Dec}(\mathsf{sk}_{i_R}, \mathsf{ct}^{j_T \to i_R}_{\mathsf{Share},R})$.
  (b) For every $j_T \in S^5_{B_T}$ set $[[\bar{\mathsf{sk}}]_{i_R}]_{j_T} = \mathsf{Dec}(\mathsf{sk}_{i_R}, \bar{\mathsf{ct}}^{j_T \to i_R}_{\mathsf{Share},R})$.
  (c) Set $[\mathsf{sk}]^{i_R}_{B_R} = \frac{1}{\Delta^3} \sum_{j_T \in S^5_{B_T}} \Delta\lambda^{S^5_{B_T}}_{0,j_T} [[\bar{\mathsf{sk}}]_{i_R}]_{j_T} - \sum_{j_T * \in S^1_{B_T}} [r_{j_T}]_{i_R}$.

SIMULATION D.1 (*Slightly Enhanced ECDSA Lazy Simulation* $\mathcal{S}_{\mathsf{ECDSA}}$)

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

```
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) 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 $(\mathsf{map}, d_0)$ and $(\mathsf{map}, d_1)$ 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 $(\mathsf{map}, d)$ to obtain $D$
   (f) $k \leftarrow 0; K \leftarrow \emptyset$
   (g) Return $D$
3. To process a group oracle query $(\mathsf{map}, i)$:
   (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 $\pi$
   (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 $\pi$
   (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 $(\mathsf{sign}, k, m_k, e_k, \alpha_k^{(0)}, \beta_k^{(0)}, \alpha_k^{(1)}, \beta_k^{(1)})$:
   (a) If $k \notin K$ or $e_k \notin \mathcal{C}$ then abort
   (b) $K \leftarrow K \setminus \{k\}$
   (c) $h_k \leftarrow \mathcal{H}_{\mathcal{M}}(m_k)$
   (d) Set $r_k^{(0)} \leftarrow \alpha_k^{(0)} \cdot \tilde{r}_k^{(0)} + \beta_k^{(0)}$ and $r_k^{(1)} \leftarrow \alpha_k^{(1)} \cdot \tilde{r}_k^{(1)} + \beta_k^{(1)}$
   (e) $R_k^{(0)} \leftarrow (\mathsf{map}, r_k^{(0)}),\ R_k^{(1)} \leftarrow (\mathsf{map}, r_k^{(1)})$
   (f) $(\mu_k^0, \mu_k^1, \mu_k^G) \leftarrow \mathcal{H}_k(D, e_k, R_k^{(0)}, R_k^{(1)}, m_k)$
   (g) $r_k \leftarrow \mu_k^0 r_k^{(0)} + \mu_k^1 r_k^{(1)} + \mu_k^G$
   (h) $R_k \leftarrow (\mathsf{map}, r_k)$
   (i) $t_k \leftarrow \bar{C}(R_k) \in \mathbb{Z}_q$; if $t_k = 0$ or $h_k + t_k d = 0$ then return fail.
   (j) $s_k \leftarrow r_k^{-1}(h_k + t_k(d + e_k))$
   (k) Return $(R_k, s_k, t_k)$
```

```
**SIMULATION D.2** ( *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) 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) $d(\tilde{\mathbf{d}}_0, \tilde{\mathbf{d}}_1) \leftarrow \mu_x^0 \cdot d_0(\tilde{\mathbf{d}}_0) + \mu_x^1 \cdot d_1(\tilde{\mathbf{d}}_1) + \mu_x^G$
       // From now on, $d(\tilde{\mathbf{d}}_0, \tilde{\mathbf{d}}_1)$ is thought of as the private key
   (e) Invoke $(\mathsf{map}, d(\tilde{\mathbf{d}}_0, \tilde{\mathbf{d}}_1))$ to obtain $D$
   (f) Set $k \leftarrow 0$ and $K \leftarrow \emptyset$, and return $D$
3. To process a group oracle query $(\mathsf{map}, i)$:
   (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 $\pi$
   (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 $\pi$
   (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 $(\mathsf{sign}, k, m_k, e_k, \alpha_k^{(0)}, \beta_k^{(0)}, \alpha_k^{(1)}, \beta_k^{(1)})$:
   (a) If $k \notin K$ or $e_k \notin \mathcal{C}$ then abort
   (b) $K \leftarrow K \setminus \{k\}$
   (c) $h_k \leftarrow \mathcal{H}_{\mathcal{M}}(m_k)$
   (d) 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
   (e) $R_k^{(0)} \leftarrow (\mathsf{map}, r_k^{(0)}(\tilde{\mathbf{r}}_k^{(0)}))$, $R_k^{(1)} \leftarrow (\mathsf{map}, r_k^{(1)}(\tilde{\mathbf{r}}_k^{(1)}))$
   (f) $(\mu_k^0, \mu_k^1, \mu_k^G) \leftarrow \mathcal{H}_k(D, e_k, R_k^{(0)}, R_k^{(1)}, m_k)$
   (g) $r_k(\tilde{\mathbf{r}}_k^{(0)}, \tilde{\mathbf{r}}_k^{(1)}) \leftarrow \mu_k^0 r_k^{(0)}(\tilde{\mathbf{r}}_k^{(0)}) + \mu_k^1 r_k^{(1)}(\tilde{\mathbf{r}}_k^{(1)}) + \mu_k^G$
   (h) $R_k \leftarrow (\mathsf{map}, r_k(\tilde{\mathbf{r}}_k^{(0)}, \tilde{\mathbf{r}}_k^{(1)}))$
   (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
       $\left(\alpha_k^{(0)}\right)^{-1} \left( \left(\mu_k^0\right)^{-1} \left( -\mu_k^1 r_k^{(1)}(\tilde{\mathbf{r}}_k^{(1)}) + s_k^{-1} t_k (d(\tilde{\mathbf{d}}_0, \tilde{\mathbf{d}}_1) + e_k) + s_k^{-1} h_k - \mu_k^G \right) - \beta_k^{(0)} \right)$
       for the variable $\tilde{\mathbf{r}}_k^{(0)}$ throughout $\mathsf{Domain}(\pi)$, to obtain $r_k(\tilde{\mathbf{r}}_k^{(0)}, \tilde{\mathbf{r}}_k^{(1)}) =$
       $s_k^{-1}(h_k + t_k(d(\tilde{\mathbf{d}}_0, \tilde{\mathbf{d}}_1) + e_k))$, and abort if any two polynomials collapse
   (l) Return $(R_k, s_k, t_k)$
```

SIMULATION D.3 (*Slightly Enhanced ECDSA Modified Symbolic Simulation*)

Key generation is applied immediately at the beginning. A 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) Return $(G, \tilde{D}_0, \tilde{D}_1)$
2. To process a key bias request $(\mathsf{biaskey}, \mathsf{id}_x)$:
   (a) $(\mu_x^0, \mu_x^1, \mu_x^G) \leftarrow \mathcal{H}_{\mathsf{key}}(\tilde{D}_0, \tilde{D}_1, \mathsf{id}_x)$
       // From now on, $\mu_x^0 \tilde{\mathbf{d}}_0 + \mu_x^1 \tilde{\mathbf{d}}_1 + \mu_x^G$ is thought of as the private key.
   (b) Invoke $(\mathsf{map}, \mu_x^0 \tilde{\mathbf{d}}_0 + \mu_x^1 \tilde{\mathbf{d}}_1 + \mu_x^G)$ to obtain $D$
   (c) $k \leftarrow 0; K \leftarrow \emptyset$
   (d) Return $D$
3. To process a group oracle query $(\mathsf{map}, i)$:
   (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 $\pi$
   (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 $\pi$
   (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 $(\mathsf{sign}, k, m_k, \mathsf{id}_k, e_k)$:
   (a) If $k \notin K$ or $e_k \notin \mathcal{C}$ then abort
   (b) $K \leftarrow K \setminus \{k\}$
   (c) $h_k \leftarrow \mathcal{H}_{\mathcal{M}}(m_k)$
   (d) $(\mu_k^0, \mu_k^1, \mu_k^G) \leftarrow \mathcal{H}_k(D, e_k, \tilde{R}_k^{(0)}, \tilde{R}_k^{(1)}, m_k, \mathsf{id}_k)$
   (e) $R_k \leftarrow (\mathsf{map}, \mu_k^0 \tilde{\mathbf{r}}_k^{(0)} + \mu_k^1 \tilde{\mathbf{r}}_k^{(1)} + \mu_k^G)$
   (f) $t_k \leftarrow \bar{C}(R_k) \in \mathbb{Z}_q$; if $t_k = 0$ then abort
   (g) $s_k \leftarrow \mathbb{Z}_q^*$
   (h) Substitute $(\mu_k^0)^{-1} \left( -\mu_k^1 \tilde{\mathbf{r}}_k^{(1)} + s_k^{-1} t_k (\mu_x^0 \tilde{\mathbf{d}}_0 + \mu_x^1 \tilde{\mathbf{d}}_1 + \mu_x^G + e_k) + s_k^{-1} h_k - \mu_k^G \right)$
       for $\tilde{\mathbf{r}}_k^{(0)}$ throughout $\mathsf{Domain}(\pi)$, and abort if any two polynomials collapse
   (i) Return $(R_k, s_k, t_k)$
```

FUNCTIONALITY D.4 (*Slightly Enhanced Schnorr Signing Oracle without biases : $\mathcal{G}^*_{\textsf{Non-Biased-SE-Sch}}$*)

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 D.5 (*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 D.6 (*$\mathcal{G}^*$-Existential Unforgeability Experiment $\mathsf{Exp}^{\mathcal{G}^*}_{\mathsf{EU}}(\mathcal{A}, 1^\kappa, n_s, n_{kg})$ following [BMP22]*)

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 $\mathsf{Verfiy}(\mathsf{sid}, \mathsf{msg}, \sigma) = 1$ and m was not queried to $\mathcal{G}^*$, otherwise output 0.

EXPERIMENT D.7 (*Algebraic One More Discrete Log* $\mathsf{Exp}^{n_{DL}}_{\mathsf{AOMDL}}(\mathcal{A}, 1^{\kappa}, (\mathbb{G}, G, q))$ *[Bol02]*)

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 D.8 (*Hybrid Simulation of Experiment D.6 with predetermined keys and presigns* $\mathcal{S}^{\mathsf{Exp}_{AOMDL}}_{\mathsf{Exp}_{EU}(\mathcal{A}^{n_s,n_{kg},n_r}_{\mathsf{SE-Sch-Forger}})}(1^\kappa, (\mathbb{G}, G, q), h_1, \ldots, h_{n_r+2n_s}; \rho)$)

This hybrid simulates the challenger in Experiment D.6 facing $\mathcal{A}^{n_s,n_{kg},n_r}_{\mathsf{SE-Sch-Forger}}$ and simulates $\mathcal{A}$ in $\mathsf{Exp}^{2n_s+n_{kg}}_{\mathsf{AOMDL}}(\mathcal{A}, 1^\kappa, (\mathbb{G}, G, q))$ facing the challenger from Experiment D.7. The hybrid uses predetermined values for: $\mathcal{A}^{n_s,n_{kg},n_r}_{\mathsf{SE-Sch-Forger}}$'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 D.7.

1. Receive $K_1, \ldots K_{2n_s+n_{kg}}$ from the challenger.
2. Replace the notation $X_\ell = K_{2n_s+\ell}$ for $\ell \le n_{kg}$.
3. Run $\mathcal{A}^{n_s,n_{kg},n_r}_{\mathsf{SE-Sch-Forger}}$ with randomness ρ as follows:
   - Upon receiving ($\mathsf{keygen}, \mathsf{sid}$ from $\mathcal{A}^{n_s,n_{kg},n_r}_{\mathsf{SE-Sch-Forger}}$:
     (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 $\mathcal{A}^{n_s,n_{kg},n_r}_{\mathsf{SE-Sch-Forger}}$.
   - Upon receiving $(\mathsf{pres}, \mathsf{ssid})$ from $\mathcal{A}^{n_s,n_{kg},n_r}_{\mathsf{SE-Sch-Forger}}$:
     (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 $\mathcal{A}^{n_s,n_{kg},n_r}_{\mathsf{SE-Sch-Forger}}$.
   - 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 $\mathcal{A}^{n_s,n_{kg},n_r}_{\mathsf{SE-Sch-Forger}}$ 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 $\mathcal{A}^{n_s,n_{kg},n_r}_{\mathsf{SE-Sch-Forger}}$.
   - 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 $\mathcal{A}^{n_s,n_{kg},n_r}_{\mathsf{SE-Sch-Forger}}$.
     - 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 $\mathcal{A}^{n_s,n_{kg},n_r}_{\mathsf{SE-Sch-Forger}}$.
       (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 $\mathcal{A}^{n_s,n_{kg},n_r}_{\mathsf{SE-Sch-Forger}}$.
   - Upon receiving $\perp$ from $\mathcal{A}^{n_s,n_{kg},n_r}_{\mathsf{SE-Sch-Forger}}$, abort.
   - Upon receiving $(\mathsf{sid}, \mathsf{msg}, K, z)$ from $\mathcal{A}^{n_s,n_{kg},n_r}_{\mathsf{SE-Sch-Forger}}$ 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 D.9 (*AOMDL adversary* $\mathcal{A}^{\mathcal{S}_{\mathsf{Exp}_{EU}}}_{\mathsf{AOMDL-Sch}}(1^\kappa, (\mathbb{G}, G, q))$)

This algorithm is a PPT adversary $\mathcal{A}$ for $\mathsf{Exp}^{2n_\mathsf{s}+n_\mathsf{kg}}_{\mathsf{AOMDL}}$ (Experiment D.7) that reduces AOMDL to forging sigantures. It uses $\mathcal{S}_{\mathsf{Exp}_\mathsf{EU}}$ to simulate the challenger in Experiment D.6 and retrieve a forged signature from $\mathcal{A}^{n_s,n_\mathsf{kg},n_r}_{\mathsf{SE-Sch-Forger}}$. It uses a fork to forge two signatures with the same nonce, in order to break AOMDL.

```
1. Sample $h_1, h'_1, \ldots, h_{n_\mathsf{r}+2n_\mathsf{s}}, h'_{n_\mathsf{r}+2n_\mathsf{s}}$ from $H$.
2. Receive $K_1, \ldots, K_{2n_\mathsf{s}+n_\mathsf{kg}}$ from the challenger.
3. Sample $\rho \leftarrow \mathcal{R}$. // the random space of $\mathcal{A}^{n_s,n_\mathsf{kg},n_r}_{\mathsf{SE-Sch-Forger}}$
4. Simulate $\mathcal{S}_{\mathsf{Exp}_\mathsf{EU}}(1^\kappa, (\mathbb{G}, G, q), h_1, \ldots, h_{n_\mathsf{r}+2n_\mathsf{s}}; \rho)$ as follows:
   – Send $K_1, \ldots, K_{2n_\mathsf{s}+n_\mathsf{kg}}$ to $\mathcal{S}_{\mathsf{Exp}_\mathsf{EU}}$.
   – Upon receiving $\alpha_1, \ldots, \alpha_{2n_\mathsf{s}+n_\mathsf{kg}}$ from $\mathcal{S}_{\mathsf{Exp}_\mathsf{EU}}$:
      (a) If there is $z$ such that $((\alpha_1, \ldots, \alpha_{2n_\mathsf{s}+n_\mathsf{kg}}), z)$ is in memory, send $z$ to $\mathcal{S}_{\mathsf{Exp}_\mathsf{EU}}$
          and break.
      (b) Otherwise, send $\alpha_1, \ldots, \alpha_{2n_\mathsf{s}+n_\mathsf{kg}}$ to the challenger.
      (c) Receive $z$ from the challenger.
      (d) Keep $((\alpha_1, \ldots, \alpha_{2n_\mathsf{s}+n_\mathsf{kg}}), z)$ 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 $\mathcal{S}_{\mathsf{Exp}_\mathsf{EU}}(1^\kappa, (\mathbb{G}, G, q), h_1, \ldots, h_{\ell_0-1}, h'_{\ell_0}, \ldots, h'_{n_\mathsf{r}+2n_\mathsf{s}}; \rho)$:
   – Send $K_1, \ldots, K_{2n_\mathsf{s}+n_\mathsf{kg}}$ to $\mathcal{S}_{\mathsf{Exp}_\mathsf{EU}}$.
   – Upon receiving $\alpha_1, \ldots, \alpha_{2n_\mathsf{s}+n_\mathsf{kg}}$ from $\mathcal{S}_{\mathsf{Exp}_\mathsf{EU}}$:
      (a) If there is $z$ such that $((\alpha_1, \ldots, \alpha_{2n_\mathsf{s}+n_\mathsf{kg}}), z)$ is in memory, send $z$ to $\mathcal{S}_{\mathsf{Exp}_\mathsf{EU}}$
          and break.
      (b) Otherwise, send $\alpha_1, \ldots, \alpha_{2n_\mathsf{s}+n_\mathsf{kg}}$ to the challenger.
      (c) Receive $z$ from the challenger.
      (d) Keep $((\alpha_1, \ldots, \alpha_{2n_\mathsf{s}+n_\mathsf{kg}}), z)$ 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 $\ell > 2n_\mathsf{s}$ 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 $\ell' \leq 2n_\mathsf{s} + n_\mathsf{kg}$, set $\alpha_{\ell'}$ to 1 if $\ell' = 2n_\mathsf{s} + \ell$, and to 0 otherwise.
12. Keep $((\alpha_1, \ldots, \alpha_{2n_\mathsf{s}+n_\mathsf{kg}}), \frac{z_0 - z_1}{h_{\ell_0} - h'_{\ell_0}})$ in memory.
13. Let $\{v_{\ell'}\}_{\ell'=1}^{2n_\mathsf{s}+n_\mathsf{kg}} \subset H^{2n_\mathsf{s}+n_\mathsf{kg}}$ be a set of vectors such that:
   – For every $((\alpha_1, \ldots, \alpha_{2n_\mathsf{s}+n_\mathsf{kg}}), z)$ in memory, there is $\ell'$ such that $v_{\ell'} =$
     $(\alpha_1, \ldots, \alpha_{2n_\mathsf{s}+n_\mathsf{kg}})$.
   – The set $\{v_{\ell'}\}_{\ell'=1}^{2n_\mathsf{s}+n_\mathsf{kg}}$ 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 $\log_G K_1, \ldots, \log_G K_{2n_\mathsf{s}+n_\mathsf{kg}}$ using the linear combinations $\{v_{\ell'}\}_{\ell'=1}^{2n_\mathsf{s}+n_\mathsf{kg}}$ as
    the result of each linear combination of them is already known.
17. Send $\log_G K_1, \ldots, \log_G K_{2n_\mathsf{s}+n_\mathsf{kg}}$ to the challenger.
```

SIMULATION D.10 (*Simulation of Slightly Enhanced Schnorr Signing Oracle (with multiplication):* $\mathcal{S}_{\textsf{SE-Sch}}$)

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

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

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 $\mathcal{F}_{\mathsf{ACS}}^{\binom{[n]}{t+1}B}$. Else consider B_j malicious.
3. **Output**:
   (a) Receive S_B from $\mathcal{F}_{\mathsf{ACS}}$ and set $\mathsf{pk} = \mathit{\Pi}_{j\in S_B} C_{j,0}$.
   (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 $\mathsf{vk}_j = \mathit{\Pi}_{j'\in S_B} C_{j',j}$.
   (d) record $(\mathsf{pk}, \{\mathsf{vk}_j\}_{j\in[n]}; [\mathsf{sk}]_i)$.