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

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

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

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

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

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

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

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

1 Introduction

In recent years, there has been growing interest in thresholdizing digital signatures [4, 11, 33], distributing the signing process among multiple parties. This is primarily because digital signatures serve as an authentication mechanism in Bitcoin [60], Ethereum [14], and similar platforms where the digital signature on a transaction constitutes a proof of ownership of the funds. Specifically, much effort has been made for thresholdizing ECDSA [1, 6, 7, 15–17, 23, 27, 28, 30, 31, 41–43, 54, 56, 73, 75, 76] due to its widespread use in cryptocurrency.

Threshold signature based solutions have been widely deployed in the cryptocurrency industry and focus mainly on two use cases: decentralized bridges (e.g. THORChain [69], Keep Network [57], Near Network [48] and Internet Computer [68]) and distributed custody (e.g. Coinbase Wallet [55], Qredo [51], Zengo [50], Fireblocks [36], Copper [24], Web3 Auth [52] and Lit Protocol [21]).

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

Additionally, to the best of our knowledge, due to inherent limitations of existing threshold ECDSA signing protocols, all real-world networks for both use-cases are deployed with a rather small number of nodes (e.g. Internet Computer uses up to 30 [68]). 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 [34]). Recently, [39] has taken a pivotal step towards increasing the number of nodes, and we follow their approach in this work. Their main observation is that designing the protocol over a *broadcast channel* is more scalable with the number of parties. Indeed, prior to the work of [39], all practical threshold ECDSA schemes were designed to work over *unicast channels*, assuming secure point-to-point (P2P) channels between *every* pair of parties. While secure point-to-point channels can be constructed in theory (under an appropriate PKI setup), in practice, they inherently incur a high network load and a message complexity quadratic in the number of parties, as each party computes and sends a message to every other party. Instead, [39] opts to rely on a *broadcast channel*, and reduces the message complexity to linear. Moreover, by working over a broadcast channel, their protocol achieves *identifiable abort* (IA) and *public verifiability* (PV). When the broadcast channel is implemented over a blockchain, one also gets *immutability of messages*, which enables easy *recovery from failures*. It also provides *incentive mechanisms* for faithful participation. Blockchains also provide *censorship resistance* for the client, as opposed to trusted custodians.

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

parties do not have synchronized clocks, cannot agree on which parties are online and which parties are unavailable. Therefore, parties who participate in one communication round may not be available for the next one. Section 2 discusses these gaps and places our approach in the context of prior work on threshold signatures. Building on [39], we address these critical challenges and provide an affirmative answer to the question below:

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

Zero Trust. Our protocol demands participation of a client in addition to the distributed network and thus maintains the *Zero Trust* principle [67], which has been foundational in blockchain ecosystems. Under this principle, no one is trusted, not even miners or validators, and every client calculates and verifies the state itself.

Currently, isolated networks that offer *interoperability* are compromising on Zero Trust principles in order to enable cross-network operations. Instead, they take a *Castle-and-Moat* approach, aiming to create a strong, fortified perimeter around their network. Unfortunately, *Castle-and-Moat Protocols* (CMPs) manage an ever-increasing amount of digital assets in cross-chain interactions, creating the *Honeypot Problem*: as a CMP gains higher *Total Value Locked* (TVL), the protocol becomes a more lucrative target for malicious players. For many years, we have witnessed attacks targeting CMPs. For instance, Wormhole [45] was exploited for over 320 million USD [74].

In order to resolve those issues and comply with the Zero Trust architecture, [39] proposed the *2PC-MPC* framework, which we follow in this work and briefly cover below. In this framework, both the client and a threshold of a decentralized network must collaborate in order to produce a signature. This ensures that even if the network is compromised, it cannot sign any transaction without client consent. In particular, on top of hacking the network, the adversary has to access the secret key share of each client individually, thereby addressing the honeypot problem. The only threat of an adversary controlling the network is Denial of Service (DoS), namely, preventing a client from using its wallet. Nevertheless, the requirement for the client to collaborate with the network to produce a signature raises many practical difficulties. First, the access structure is inherently hierarchical. Prior solutions that used a threshold access structure had to compromise on the size of the network, as the user must have the same weight as the whole network. Second, the client is typically lightweight, and does not have comparable computational power to parties in the decentralized network. Optimally, the computation complexity of the client should be independent of the network size. Third, the client typically does not have direct access to internal entities in the decentralized network, and might not even be aware of their identities. Exposing such communication channels may pose additional cybersecurity threat for parties in the network, and would also significantly increase the communication overhead of the client.

With the above issues in mind, [39] added to the 2PC-MPC *hierarchical access structure* an additional requirement that when met, allows for the system

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

Another important aspect of a decentralized system is the number of clients it can concurrently service. Prior works on threshold signatures are typically focused on a single key, and do not consider the system as a whole. Therefore, each node in the decentralized network must hold a private share per each wallet of each client in the system. Instead, in [39], 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) [25]. Essentially, the network share of each signing key is stored in public, encrypted under the TAHE public key, and only the underlying AHE secret decryption key is secretly shared.[2] In this work, we take a different approach, enabling the network to utilize a single *global key share* which will be used across all clients. This change also significantly simplifies any resharing or reconfiguration of the secret. Specifically, one does not need to reshare the secret key of the TAHE scheme but instead may only reshare elements in $\mathbb{Z}_q$ and generate a fresh TAHE key.

1.1 Our Contribution

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

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

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

[2] The idea of using TAHE and storing only the secret-key shares to minimize private storage was proposed in the YOSO line of work [44].

Technical Contributions. On top of that, this work introduces several contributions that might have independent interest. As a side-effect of complying with asynchronous networks, we improve the round complexity, compared to the 2PC-MPC protocol in [39]. This is because we may not use commitment rounds, as parties may not be available in the next round to open them. As a result, both DKG and signing consist of a one-round trip between the blockchain and the client. Our sign phase can be broken into a *presign* phase, and an *online-signing phase*. The presign phase is executed without client involvement in an *offline phase*, potentially before the message to be signed is decided. This preprocessing can reduce computation and latency during high-demand periods. When instantiated in the standard threshold access structure, that is, by removing the client and retaining only the committee of parties, DKG completes in a single round, presigning requires two rounds; and the online signing step is a single round. In this sense, our round complexity matches the best known constructions that allow the presigning and signing phases to involve different subsets of parties [23]. If, instead, one enforces that the same set of parties participates in both presign and sign, then recent two-round protocols are also known [26, 58].

In addition, we alleviate the need for *zero-knowledge* (ZK) proofs towards the client, that were used in [39] to prove honest behavior. In order to admit the 2PC-MPC framework, [39] offered to aggregate the proofs from each party in the network. However, their aggregation approach is inherently synchronous and as such doesn't work for our purpose.

In terms of security, our protocol circumvents the *Enhanced-ECDSA unforgeability* assumption (proposed in [15]), even when allowing presigns. Enhanced ECDSA was proven secure in the EC-GGM model in [46], but the concrete parameters of the reduction are not sufficient when working over 256-bit elliptic curves commonly used nowadays (e.g., `secp256k1` [5] used in both Bitcoin and Ethereum). We propose the *Slightly-Enhanced ECDSA unforgeability* assumption, and provide a reduction (also in EC-GGM) with a satisfying complexity. We believe other threshold ECDSA protocols can be adjusted to rely on this assumption instead.

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

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

1.2 System Overview

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

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

The blockchain is based on *Proof of Stake* (PoS), and each *validator* has a *voting power* proportional to its stake [53]. In turn, during each reconfiguration, validators may also withdraw part of their stake, or increase their stake, which could change their voting power accordingly. The weighted access structure is implemented by giving each validator a number of secret shares of the TAHE secret key proportional to their voting power. This only affects performance during the sign phase of the protocol.

Communication. Clients can communicate with the blockchain by sending transactions and parsing blocks. The MPC party emulation sub-protocols can be implemented over the *consensus protocol* [9,10,37] used for producing blocks. The *block proposer* validator is responsible for aggregating the last MPC round, computing the message to send to the client, and including it in its proposed block. Oftentimes, such a communication channel is either asynchronous (as in Narwhal-Tusk [29]) or *partially synchronous* (as in Narwhal [66]-Bullshark).

Policies. Upon creation, wallets will specify policies, either by using *smart contracts* (see [59, 71] and [47, 70] for recent overviews and surveys) implemented on-chain, or using light-clients [19] in order to validate policies posted on another blockchain. Before validators participate in signing a transaction, they validate that it matches the policy of the corresponding wallet. Policies can specify, for instance, a *whitelist* or a *blacklist* of accounts, a *transaction limit*, a limit on the number of transactions per epoch, or anything else that can be specified in a smart contract. Compromising a client will not allow an adversary to bypass the policy of its wallet. Compromising the blockchain can only be exploited to bypass policies for wallets for which the client share was also compromised. This ensures security can only be enhanced by sharing the key with the blockchain. The only added risk for the client is DOS of the network.

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

1.3 Paper Organization

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

2 Related Work

This section discusses prior work, focusing on [39] whose 2PC-MPC framework we follow.

The work in [39] is described over an *ideal broadcast functionality*. In reality, consensus protocols implement a broadcast channel over P2P channels, e.g. via gossiping [2]. This distinction is crucial, since implementing an ideal broadcast channel over asynchronous networks is theoretically impossible [37]. This abstraction complicates using off-the-shelf broadcast implementations; we address these issues in this work.

In particular, they assume a *synchronous communication channel*, with a known bound on clock drift and message transmission delays, allowing parties to agree if a party does not broadcast a message. This also occurs in [63], an adaptation of [31] that achieves identifiable abort via a synchronous broadcast channel. Synchronous communication is a stringent requirement [72]; blockchains typically use partially-synchronous [32] or asynchronous channels. In this work, protocols are defined over an asynchronous channel, supporting existing broadcast implementations (e.g., Narwhal-Tusk [29], Narwhal-Bullshark [66], Mysticeti [3]) with significantly lower latency. In asynchronous communication, parties compute and broadcast messages for the next round as soon as a threshold from the previous round broadcasts, without coordinating on which parties failed to participate. While ROAST [63] adapts FROST1 [49] for asynchronous communication, it requires running multiple FROST sessions in parallel (up to the number of honest parties), making it asymptotically more expensive and consuming significant storage for managing parallel sessions.

In addition, prior works such as [39] often require the same subset of parties to participate in all rounds of a subprotocol (DKG, presign or sign). This is not on par with existing broadcast channel implementations [3, 29, 66], which only ensure that an arbitrary threshold of parties is online during each round. Our protocol allows participants to dynamically change between rounds, which aligns with existing broadcast implementations. While [63] also addresses this indirectly by ensuring one session where a fixed threshold participates in all rounds, our approach allows different subsets in each round, yielding better performance.

Finally, beyond practical challenges, we address theoretical concerns in securing ECDSA with presigns. Prior threshold ECDSA protocols enabling presigns (e.g., [15, 17, 27, 31, 39]) assumed the Enhanced ECDSA Unforgeability assumption [15], stating that forging signatures is hard even if the adversary queries nonce parts before deciding on messages. This was established in the EC-GGM model [46], but with cubic reduction complexity, insufficient for 256-bit curves. We propose Slightly-Enhanced (SE) ECDSA Unforgeability and achieve quadratic complexity. Our presign consists of a pair of nonces; at signing, they are combined based on the message, client input, presign, and public key, making SE-ECDSA compatible with 256-bit curves. Notably, in Appendix G.3, we prove security of an analogous SE-Schnorr oracle under the *Algebraic One-More Discrete Log* (AOMDL) assumption, which may simplify simulation of threshold Schnorr protocols with presigns such as FROST3 [22].

3 Preliminaries

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

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

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

The internal communication channel between parties in B is an asynchronous reliable broadcast (ARB) channel, which is captured in $\mathcal{F}_{\mathsf{broadcast}}$ (Functionality 1). This is an abstraction of the communication channel used in blockchains to achieve agreement on transactions. Some of the potential instantiations can be found in [3,29,66]. Following [65], we note that the functionality only ensures *consistency*, and not *completeness*. Completeness suggests that if a single honest party generates an output, then every honest party eventually generates an output. Instead, this is viewed as a property of the broadcast protocol itself. In our context, completeness ensures guaranteed output delivery, but for security and correctness of our protocol, consistent broadcast suffices.

Typically, a consensus protocol is implemented on top of a reliable broadcast channel, so as to agree on an *order* for the transactions. However, in our context, a weaker notion may suffice. Recall that our system produces signatures to be posted on other blockchains, who may be responsible for ordering these transactions. Therefore, we only require a mechanism to agree upon a subset of messages corresponding to an authorized subset of parties from the previous MPC round of a given session sid. This is captured by $\mathcal{F}_{\mathsf{ACS}}$ (Functionality 3). A potential instantiation can be found in [65]. 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 [65], while $\mathcal{F}_{\mathsf{ACS}}$ is a rephrasing from [64].

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

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

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

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

All of the ZKPs used are non-interactive, either by applying the Fiat-Shamir transform [35], or by applying Fischlin's transform [38] in case UC-extractable proofs are needed. A ZK protocol for the language L will be denoted as $\Pi_{\mathsf{zk}}^{L[\mathsf{pp}]}$ for the former case, and as $\Pi_{\mathsf{uc\text{-}zk}}^{L[\mathsf{pp}]}$ for the latter case. Formal definitions of the languages used in our protocol are provided in Appendix C.

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

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

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

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

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

Homomorphic Commitments. For a commitment scheme with public parameters pp we denote the message space of the commitment as $\mathcal{M}_{\mathsf{pp}}$ and randomness space as $\mathcal{R}_{\mathsf{pp}}$. Unless specified otherwise $\mathsf{Com}_{\mathsf{pp}}$ refers to Pedersen Commitments [62] on the appropriate elliptic curve. Homomorphic Addition is denoted by $\oplus$ and scalar multiplication by $\odot$.

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

We utilize an AHE scheme with $\mathcal{P}_{pk} = \mathbb{Z}_q$, where q is the prime order of the elliptic curve used for the digital signature scheme. This can be achieved natively using class group-based AHE [18] or by interpreting the plaintext as a $\mathbb{Z}$-module using the Paillier scheme [61], as done in [39]. We require the AHE scheme to satisfy *circuit privacy* with respect to linear transformations. Specifically, the scheme admits *Secure Linear Evaluation* of two ciphertexts, ct_0 and ct_1, with coefficients a_0 and a_1, denoted as $\mathsf{AHE.Eval}(\mathsf{pk}, (\mathsf{ct}_0, \mathsf{ct}_1), (a_0, a_1); \eta)$. This evaluation should not reveal any information to an adversary holding sk and the

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

In our protocols, we utilize Threshold Additively Homomorphic Encryption (TAHE), which allows any set of $t+1$ parties to decrypt while preventing any smaller subset from doing so. This is captured by an ideal functionality $\mathcal{F}_{\mathsf{TAHE}}$ (Functionality 5). We define an Additively Homomorphic Encryption (AHE) scheme and extend it to the threshold security model. This functionality can be emulated using class group-based TAHE [13] or Paillier-based TAHE with circuit privacy ([39], following [40]). Importantly, to obtain security against malicious adversaries, TAHE schemes often use *verification keys* which should be thought of as homomorphic commitments on the secret key shares. The verification keys are produced in the *Distributed Key Generation* (DKG) phase, and are then used during the threshold decryption phase in order to prove correctness of the decryption shares.

4 Protocols

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

4.1 Global Key Generation

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

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

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

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

4.2 Client Key Generation

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

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

4.3 Global Presign

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

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

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

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

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

4.4 Online Sign

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

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

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

The protocol outputs:

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

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

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

Nevertheless, the above protocol enables the following attack. Party B sends X instead of R_B, which becomes possible now that a ZKP for $R_B \leftarrow X$ is not required. It will get $k_A^{-1} \cdot X$ and (the encryption of) $\sigma_A = k_A(m + rx)$. It then multiplies $k_A^{-1} X$ by σ_A to get $(m + rx) \cdot X$, which gives away $x^2 \cdot G$. By repeating this, it can get $x^k \cdot G$, for any $k \in \mathsf{poly}(\kappa)$. This is an instance of the Strong Diffie-Hellman Problem which is known to be broken in certain settings [20].

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

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

5 Security

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

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

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

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

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

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

5.1 Analyzing Slightly Enhanced ECDSA

In this section, we provide a sketch of the security proof of the existential unforgeability game 1 with respect to the signing oracle 2 in the EC-GGM model [46]. In this model, a *group oracle* essentially samples a random mapping π between $\mathbb{Z}_q$ and the curve $\mathbb{G}$. The adversary interacts with the group oracle in order to perform computations over $\mathbb{G}$.

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

Proof (sketch). Our proof follows the steps taken in [46], and we refer to Appendix D.2 for a detailed proof.

As in [46], 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 [46], we show that by using two nonces in the presign, it is possible to keep one of the two corresponding variables symbolic throughout the entire simulation. This is the main difference which leads to a better security bound. Specifically, we prove that any adversary to our protocol that can forge a signature in the EC-GGM model with non-negligible probability must use $\Omega(\sqrt{q})$ overall oracle queries.

The security proof of the symbolic simulation is based on the symbolic verification equation of the forged signature and the equality of symbolic polynomials. Following [46], we then split the proof into several cases, depending on the creation of R^* of the forged signature. We show that the probability of each case is negligible.

5.2 Protocol Simulations.

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

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

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

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

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

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

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

6 Performance

We implemented our ECDSA protocol in Rust; the code will be made publicly available upon publication. We instantiate our TAHE with Class Groups based on [12]. We also build upon [8] and implement the first constant-time Class-Group library. All times are reported for $\sigma = 64, \kappa = 128, t = \frac{2}{3}n$ and the secp256k1 curve. We report the raw data for our experiments in Appendix E.

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

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

Real-World Deployment. We report performance data from real-world usage of our protocol over the Narwhal consensus layer with $n = 85$ validators in a permissionless setting where participants may dynamically enter and leave. Validator nodes are required to have minimum hardware specifications of 16 physical CPU cores, 128 GB RAM, 4 TB NVMe storage, and 1 Gbps network connectivity. That said, this requirement cannot be enforced or verified. Figure 2 shows the total protocol time distribution across validators (i.e. including both computation and communication), with complete data provided in Tables 5 and 6. The Sign protocol achieves a mean time of 1.05 s with a standard deviation of 0.47 s, while the Presign protocol requires 17.3 s on average with 9.2 s standard deviation. The significant variance (CV $\approx$ 45-53%) demonstrates that asynchronous execution provides performance benefits beyond being merely a consensus necessity; validators complete computations at heterogeneous speeds without blocking protocol progress.

We do not have direct access to a global breakdown of communication versus computation time across validators. That said an execution report was provided to us by a single participating validator. Based on this report, communication accounts for 88.3% of the Presign protocol execution time and 54.4% of the Sign protocol execution time, where communication includes all consensus layer overhead as well as data uploads and downloads. The full reported measurements are summarized in Table 8.

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

References

1. Abram, D., Nof, A., Orlandi, C., Scholl, P., Shlomovits, O.: Low-bandwidth threshold ECDSA via pseudorandom correlation generators. In: SP. pp. 2554–2572. IEEE (2022)
2. Aysal, T.C., Yildiz, M.E., Sarwate, A.D., Scaglione, A.: Broadcast gossip algorithms for consensus. IEEE Transactions on Signal processing **57**(7), 2748–2761 (2009)
3. Babel, K., Chursin, A., Danezis, G., Kokoris-Kogias, L., Sonnino, A.: Mysticeti: Low-latency dag consensus with fast commit path. arXiv preprint arXiv:2310.14821 (2023)
4. Bacho, R., Loss, J., Tessaro, S., Wagner, B., Zhu, C.: Twinkle: Threshold signatures from ddh with full adaptive security. In: Annual International Conference on the Theory and Applications of Cryptographic Techniques. pp. 429–459. Springer (2024)
5. Balasubramanian, K.: Security of the secp256k1 elliptic curve used in the bitcoin blockchain. Indian Journal of Cryptography and Network Security (IJCNS) **4**(1), 1–5 (2024)
6. Blokh, C., Makriyannis, N., Peled, U.: Efficient asymmetric threshold ecdsa for mpc-based cold storage. Cryptology ePrint Archive (2022)
7. Boneh, D., Gennaro, R., Goldfeder, S.: Using level-1 homomorphic encryption to improve threshold DSA signatures for bitcoin wallet security. In: LATINCRYPT. Lecture Notes in Computer Science, vol. 11368, pp. 352–377. Springer (2017)
8. Bouvier, C., Castagnos, G., Imbert, L., Laguillaumie, F.: I want to ride my bicycl: Bicycl implements cryptography in class groups. Journal of Cryptology **36**(3), 17 (2023)
9. Bracha, G., Toueg, S.: Resilient consensus protocols. In: Proceedings of the second annual ACM symposium on Principles of distributed computing. pp. 12–26 (1983)
10. Bracha, G., Toueg, S.: Asynchronous consensus and broadcast protocols. Journal of the ACM (JACM) **32**(4), 824–840 (1985)
11. Brandão, L., Davidson, M.: Notes on threshold eddsa/schnorr signatures (2023)
12. Braun, L., Castagnos, G., Damgård, I., Laguillaumie, F., Melissaris, K., Orlandi, C., Tucker, I.: An improved threshold homomorphic cryptosystem based on class groups. Cryptology ePrint Archive (2024)
13. Braun, L., Damgård, I., Orlandi, C.: Secure multiparty computation from threshold encryption based on class groups. In: Annual International Cryptology Conference. pp. 613–645. Springer (2023)
14. Buterin, V., et al.: Ethereum white paper. GitHub repository **1**, 22–23 (2013)
15. Canetti, R., Gennaro, R., Goldfeder, S., Makriyannis, N., Peled, U.: UC non-interactive, proactive, threshold ECDSA with identifiable aborts. In: CCS. pp. 1769–1787 (2020)
16. Castagnos, G., Catalano, D., Laguillaumie, F., Savasta, F., Tucker, I.: Bandwidth-efficient threshold EC-DSA. In: PKC. Lecture Notes in Computer Science, vol. 12111, pp. 266–296. Springer (2020)

17. Castagnos, G., Catalano, D., Laguillaumie, F., Savasta, F., Tucker, I.: Bandwidth-efficient threshold ec-dsa revisited: Online/offline extensions, identifiable aborts proactive and adaptive security. Theoretical Computer Science **939**, 78–104 (2023)
18. Castagnos, G., Laguillaumie, F.: Linearly homomorphic encryption from. In: Cryptographers' Track at the RSA Conference. pp. 487–505. Springer (2015)
19. Chatzigiannis, P., Baldimtsi, F., Chalkias, K.: Sok: Blockchain light clients. In: International Conference on Financial Cryptography and Data Security. pp. 615–641. Springer (2022)
20. Cheon, J.H.: Security analysis of the strong diffie-hellman problem. In: Annual International Conference on the Theory and Applications of Cryptographic Techniques. pp. 1–11. Springer (2006)
21. Chris, C., David, S.: Lit protocol whitepaper. Github Repository https://github.com/LIT-Protocol/whitepaper (2024)
22. Chu, H., Gerhart, P., Ruffing, T., Schröder, D.: Practical schnorr threshold signatures without the algebraic group model. In: Annual International Cryptology Conference. pp. 743–773. Springer (2023)
23. Cohen, R., Doerner, J., Kondi, Y., Shelat, A.: Secure multiparty computation with identifiable abort via vindicating release. In: Annual International Cryptology Conference. pp. 36–73. Springer (2024)
24. Copper: Wallets as a service. Whitepaper https://copper.co/en/products/wallets-as-a-service (2021)
25. Cramer, R., Damgård, I., Nielsen, J.B.: Multiparty computation from threshold homomorphic encryption. In: Advances in Cryptology—EUROCRYPT 2001: International Conference on the Theory and Application of Cryptographic Techniques Innsbruck, Austria, May 6–10, 2001 Proceedings 20. pp. 280–300. Springer (2001)
26. Dahari-Garbian, H., Nof, A., Parker, L.: Trout: Two-round threshold ecdsa from class groups. In: Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security. pp. 380–393 (2025)
27. Dalskov, A.P.K., Orlandi, C., Keller, M., Shrishak, K., Shulman, H.: Securing DNSSEC keys via threshold ECDSA from generic MPC. In: ESORICS. vol. 12309, pp. 654–673. Springer (2020)
28. Damgård, I., Jakobsen, T.P., Nielsen, J.B., Pagter, J.I., Østergård, M.B.: Fast threshold ECDSA with honest majority. In: SCN. LNCS, vol. 12238, pp. 382–400. Springer (2020)
29. Danezis, G., Kokoris-Kogias, L., Sonnino, A., Spiegelman, A.: Narwhal and tusk: a dag-based mempool and efficient bft consensus. In: Proceedings of the Seventeenth European Conference on Computer Systems. pp. 34–50 (2022)
30. Doerner, J., Kondi, Y., Lee, E., Shelat, A.: Threshold ECDSA from ECDSA assumptions: The multiparty case. In: SP 2019. pp. 1051–1066. IEEE (2019)
31. Doerner, J., Kondi, Y., Lee, E., Shelat, A.: Threshold ECDSA in three rounds. IACR Cryptol. ePrint Arch. p. 765 (2023)
32. Dwork, C., Lynch, N., Stockmeyer, L.: Consensus in the presence of partial synchrony. J. ACM **35**(2), 288–323 (apr 1988). https://doi.org/10.1145/42282.42283, https://doi.org/10.1145/42282.42283
33. Ergezer, S., Kinkelin, H., Rezabek, F.: A survey on threshold signature schemes. Network **49** (2020)
34. Etherscan: Ethereum node tracker. Etherscan https://etherscan.io/nodetracker (2024)
35. Fiat, A., Shamir, A.: How to prove yourself: Practical solutions to identification and signature problems. In: Conference on the theory and application of cryptographic techniques. pp. 186–194. Springer (1986)

36. 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)
37. Fischer, M.J., Lynch, N.A., Paterson, M.S.: Impossibility of distributed consensus with one faulty process. Journal of the ACM (JACM) **32**(2), 374–382 (1985)
38. Fischlin, M.: Communication-efficient non-interactive proofs of knowledge with online extractors. In: Annual International Cryptology Conference. pp. 152–168. Springer (2005)
39. Friedman, O., Marmor, A., Mutzari, D., Sadika, O., Scaly, Y.C., Spiizer, Y., Yanai, A.: 2pc-mpc: Emulating two party ecdsa in large-scale mpc. Cryptology ePrint Archive (2024)
40. Friedman, O., Marmor, A., Mutzari, D., Scaly, Y.C., Spiizer, Y., Yanai, A.: Tiresias: Large scale, maliciously secure threshold paillier. Cryptology ePrint Archive (2023)
41. Gagol, A., Kula, J., Straszak, D., Swietek, M.: Threshold ECDSA for decentralized asset custody. IACR Cryptol. ePrint Arch. p. 498 (2020)
42. Gennaro, R., Goldfeder, S.: Fast multiparty threshold ECDSA with fast trustless setup. IACR Cryptol. ePrint Arch. p. 114 (2019)
43. Gennaro, R., Goldfeder, S., Narayanan, A.: Threshold-optimal DSA/ECDSA signatures and an application to bitcoin wallet security. In: International Conference on Applied Cryptography and Network Security. pp. 156–174. Springer (2016)
44. Gentry, C., Halevi, S., Krawczyk, H., Magri, B., Nielsen, J.B., Rabin, T., Yakoubov, S.: Yoso: You only speak once: Secure mpc with stateless ephemeral roles. In: Annual International Cryptology Conference. pp. 64–93. Springer (2021)
45. Gray, E.: Governor. GitHub Repository https://github.com/wormhole-foundation/wormhole/blob/main/whitepapers/0007_governor.md (2022)
46. Groth, J., Shoup, V.: On the security of ecdsa with additive key derivation and presignatures. In: Annual International Conference on the Theory and Applications of Cryptographic Techniques. pp. 365–396. Springer (2022)
47. Hu, B., Zhang, Z., Liu, J., Liu, Y., Yin, J., Lu, R., Lin, X.: A comprehensive survey on smart contract construction and execution: paradigms, tools, and systems. Patterns **2**(2) (2021)
48. Illia, P., Alexander, S.: The near white paper. Whitepaper https://pages.near.org/papers/the-official-near-white-paper/ (2021)
49. Komlo, C., Goldberg, I.: Frost: flexible round-optimized schnorr threshold signatures. In: Selected Areas in Cryptography: 27th International Conference, Halifax, NS, Canada (Virtual Event), October 21-23, 2020, Revised Selected Papers 27. pp. 34–65. Springer (2021)
50. KZen, T.: Bitcoin wallet powered by two-party ecdsa - extended abstract. Whitepaper https://github.com/ZenGo-X/gotham-city/blob/master/white-paper/ (2019)
51. KZen, T.: Radical new infrastructure for digital asset ownership and blockchain interoperability. Whitepaper https://www.qredo.com/qredo-white-paper.pdf (2019)
52. Labs, T.: Web3 auth. Github Repository https://github.com/web3auth (2024)
53. Leonardos, S., Reijsbergen, D., Piliouras, G.: Weighted voting on the blockchain: Improving consensus in proof of stake protocols. International Journal of Network Management **30**(5), e2093 (2020)
54. Lindell, Y.: Fast secure two-party ECDSA signing. In: Annual International Cryptology Conference. pp. 613–644. Springer (2017)
55. Lindell, Y.: Digital asset management with mpc. Whitepaper https://www.coinbase.com/blog/digital-asset-management-with-mpc-whitepaper (2023)

56. Lindell, Y., Nof, A.: Fast secure multiparty ECDSA with practical distributed key generation and applications to cryptocurrency custody. In: CCS. pp. 1837–1854. ACM (2018)
57. Luongo, M., Pon, C.: The keep network: A privacy layer for public blockchains. Keep Netw., Rep (2018)
58. Lyu, Y., Li, Z., Zhou, H.S., Deng, X.: Threshold ecdsa in two rounds. In: Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security. pp. 2937–2950 (2025)
59. Mohanta, B.K., Panda, S.S., Jena, D.: An overview of smart contract and use cases in blockchain technology. In: 2018 9th international conference on computing, communication and networking technologies (ICCCNT). pp. 1–4. IEEE (2018)
60. Nakamoto, S.: Bitcoin whitepaper. URL: https://bitcoin.org/bitcoin.pdf-(:17.07.2019) **9**, 15 (2008)
61. Paillier, P.: Public-key cryptosystems based on composite degree residuosity classes. In: Advances in Cryptology-EUROCRYPT 99: International Conference on the Theory and Application of Cryptographic Techniques Prague, Czech Republic, May 2-6, 1999 Proceedings 18. pp. 223–238. Springer (1999)
62. Pedersen, T.P.: Non-interactive and information-theoretic secure verifiable secret sharing. In: Annual international cryptology conference. pp. 129–140. Springer (1991)
63. Ruffing, T., Ronge, V., Jin, E., Schneider-Bensch, J., Schröder, D.: Roast: robust asynchronous schnorr threshold signatures. In: Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security. pp. 2551–2564 (2022)
64. Shoup, V.: A theoretical take on a practical consensus protocol. Cryptology ePrint Archive (2024)
65. Shoup, V., Smart, N.P.: Lightweight asynchronous verifiable secret sharing with optimal resilience. Journal of Cryptology **37**(3), 27 (2024)
66. Spiegelman, A., Giridharan, N., Sonnino, A., Kokoris-Kogias, L.: Bullshark: Dag bft protocols made practical. In: Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security. pp. 2705–2718 (2022)
67. Stafford, V.: Zero trust architecture. NIST special publication **800**, 207 (2020)
68. Team, D., et al.: The internet computer for geeks. Cryptology ePrint Archive (2022)
69. Thorchain: Thorchain: A decentralised liquidity network. Whitepaper https://github.com/thorchain/Resources/blob/master/Whitepapers/THORChain-Whitepaper-May2020.pdf (2020)
70. Tolmach, P., Li, Y., Lin, S.W., Liu, Y., Li, Z.: A survey of smart contract formal specification and verification. ACM Computing Surveys (CSUR) **54**(7), 1–38 (2021)
71. Wang, S., Yuan, Y., Wang, X., Li, J., Qin, R., Wang, F.Y.: An overview of smart contract: architecture, applications, and future trends. In: 2018 IEEE Intelligent Vehicles Symposium (IV). pp. 108–113. IEEE (2018)
72. Wilhelmi, F., Giupponi, L., Dini, P.: Analysis and evaluation of synchronous and asynchronous flchain. Computer Networks **218**, 109390 (2022)
73. Wong, H.W.H., Ma, J.P.K., Yin, H.H.F., Chow, S.S.M.: Real threshold ECDSA. In: NDSS. The Internet Society (2023)
74. Wormhole: Wormhole incident report — 02/02/22. Wormholecrypto Medium https://wormholecrypto.medium.com/wormhole-incident-report-02-02-22-ad9b8f21eec6 (2022)
75. Xue, H., Au, M.H., Xie, X., Yuen, T.H., Cui, H.: Efficient online-friendly two-party ECDSA signature. IACR Cryptol. ePrint Arch. p. 318 (2022)

76. Zyskind, G., Yanai, A., Pentland, A.S.: Unstoppable wallets: Chain-assisted threshold ECDSA and its applications. IACR Cryptol. ePrint Arch. p. 832 (2023)

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

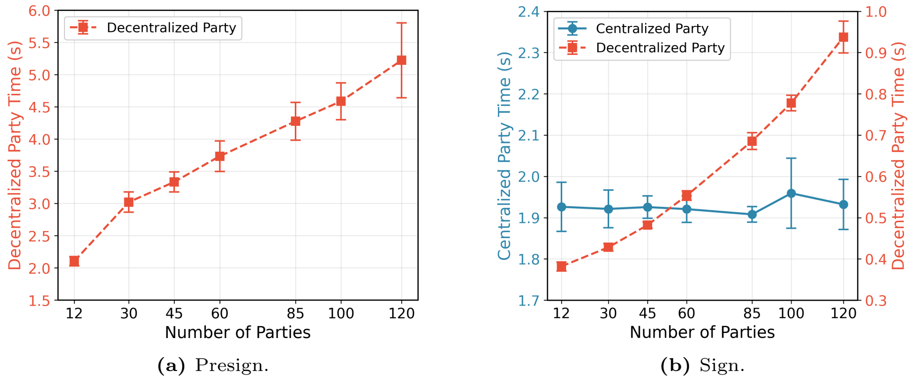
1. A_{pid_A} **does as follows**:
   (a) Calculate $\mathsf{ct}_{\gamma\cdot\mathsf{key}} = \mu^0_x \odot \mathsf{ct}_{\gamma_B\cdot\mathsf{key},0} \oplus \mu^1_x \odot \mathsf{ct}_{\gamma_B\cdot\mathsf{key},1} \oplus \mu^G_x \odot \mathsf{ct}_\gamma$ and $X = X_A + \mu^0_x X_{0,B} + \mu^1_x \cdot X_{1,B}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

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

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

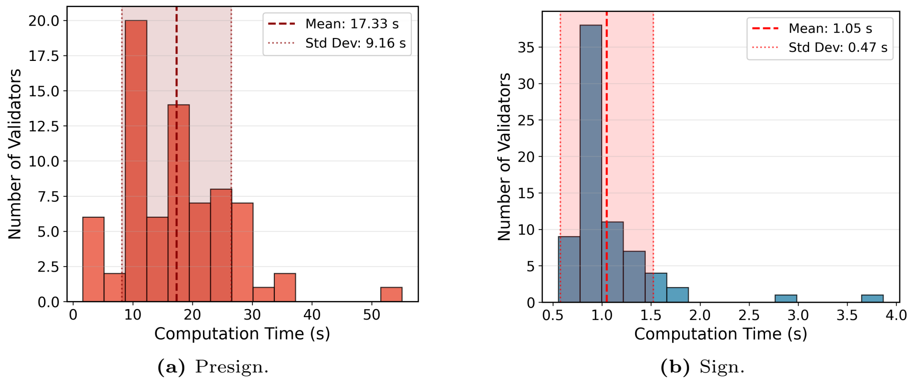
1. Call $\mathcal{G}^*$ with (setup) and hand $(\mathbb{G}, G, q)$ to $\mathcal{A}$.
2. The adversary $\mathcal{A}$ makes at most $n_\mathsf{kg} = \mathsf{poly}(\kappa)$ adaptive keygen calls to $\mathcal{G}^*$ and at most $n_\mathsf{s} = \mathsf{poly}(\kappa)$ adaptive pres or sign calls to $\mathcal{G}^*$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

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



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

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



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

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