Refresh When You Wake Up: Proactive Threshold Wallets with Offline Devices*

Yashvanth Kondi
ykondi@ccs.neu.edu
Northeastern University

Bernardo Magri
magri@cs.au.dk
Aarhus University

Claudio Orlandi
orlandi@cs.au.dk
Aarhus University

Omer Shlomovits
omer@ZenGo.com
KZen Research

January 22, 2021

Abstract

Proactive security is the notion of defending a distributed system against an attacker who compromises different devices through its lifetime, but no more than a threshold number of them at any given time. The emergence of threshold wallets for more secure cryptocurrency custody warrants an efficient proactivization protocol tailored to this setting. While many proactivization protocols have been devised and studied in the literature, none of them have communication patterns ideal for threshold wallets. In particular a (t, n) threshold wallet is designed to have t parties jointly sign a transaction (of which only one may be honest) whereas even the best current proactivization protocols require at least an additional $t-1$ *honest* parties to come online simultaneously to refresh the system.

In this work we formulate the notion of refresh with *offline devices*, where any t_ρ parties may proactivize the system at any time and the remaining $n - t_\rho$ offline parties can non-interactively "catch up" at their leisure. However, many subtle issues arise in realizing this pattern. We identify that this problem is divided into two settings: $(2, n)$ and (t, n) where $t > 2$. We develop novel techniques to address both settings as follows:

- We show that the $(2, n)$ setting permits a tight t_ρ for refresh. In particular we give a highly efficient $t_\rho = 2$ protocol to upgrade a number of standard $(2, n)$ threshold signature schemes to proactive security with offline refresh. This protocol can augment existing implementations of threshold wallets for immediate use– we show that proactivization does not have to interfere with their native mode of operation. This technique is compatible with Schnorr, EdDSA, and even sophisticated ECDSA protocols. By implementation we show that proactivizing two different recent $(2, n)$ ECDSA protocols incurs only 14% and 24% computational overhead respectively, less than 200 bytes, and no extra round of communication.
- For the general (t, n) setting we prove that it is impossible to construct an offline refresh protocol with $t_\rho < 2(t-1)$, i.e. tolerating a dishonest majority of online parties. Our techniques are novel in reasoning about the message complexity of proactive security, and may be of independent interest.

Our results are positive for small-scale decentralization (such as 2FA with threshold wallets), and negative for large-scale distributed systems with higher thresholds. We thus initiate the study of proactive security with offline refresh, with a comprehensive treatment of the dishonest majority case.

*Research supported by: the Concordium Blockhain Research Center, Aarhus University, Denmark; the Carlsberg Foundation under the Semper Ardens Research Project CF18-112 (BCM); the European Research Council (ERC) under the European Unions's Horizon 2020 research and innovation programme under grant agreement No 803096 (SPEC); the Danish Independent Research Council under Grant-ID DFF-6108-00169 (FoCC); the Office of the Director of National Intelligence (ODNI), Intelligence Advanced Research Project Activity (IARPA) under contract number 2019-19-020700009 (ACHILLES). This is the full version of the paper under the same title appearing in the 2021 IEEE S&P Conference [KMOS21].

1 Introduction

Threshold Signatures as conceived by Desmedt [Des87] allow the ability to sign messages under a public key to be delegated to a group of parties instead of a single one. In particular, a subset of these parties greater than a certain threshold must collaborate in order to sign a message. This primitive finds application in many scenarios, but more recently it has seen interest from the blockchain community as a method to manage private keys effectively. From multi-factor authentication to distribution of spending authority, threshold signature schemes allow cryptocurrency wallets to build resilience against compromise of up to a threshold number of devices. This is because threshold signature protocols never physically reconstruct the signing key at a single location, and so an attacker who compromises fewer devices than the signing threshold learns no useful information to forge signatures.

A long line of works has constructed threshold versions of common signature schemes [GJKR01, ADN06, Sho00]. Despite the non-linearity of the ECDSA signing equation making its thresholdization challenging, recent works have seen even threshold ECDSA schemes [GG18, LNR18, DKLs19, DOK$^+$20, CCL$^+$20] enter the realm of practicality. This has immediate implications for users of the many cryptocurrencies (Bitcoin, Ethereum, etc.) that have adopted ECDSA as their canonical signature algorithm. Besides ECDSA, Schnorr [Sch89] and other Schnorr-like signature schemes (eg. EdDSA [BDL$^+$12]) are seeing an increase in interest from the cryptocurrency community, of which many employ threshold-friendly signing equations.

However threshold signature schemes by themselves do not address a number of security concerns that arise in real-world deployment. Indeed, all privacy/unforgeability guarantees of such a system are completely and irreparably voided if an adversary breaks into even one device more than the threshold *throughout the lifetime of the system*. A natural question to ask is instead of assuming that an adversary is threshold-limited to the same devices essentially forever, whether it is meaningful to consider a threshold-limited adversary with mobility across devices in time. In more detail an attacker may break into different devices in the system (possibly *all* of them in its lifetime) however at any given point in time, not more than a threshold number of them are compromised. This question was first considered by Ostrovsky and Yung [OY91] who devised the notion of a *mobile adversary*, which may change which devices are compromised at marked epochs in time. They found that the trick to thwarting such an adversary is to have each party proactively re-randomize its secret state between epochs. This technique ensures that the views of different parties at different epochs in time are independent, and can not be combined to reveal any meaninningful information about shared secrets by a mobile attacker.

1.1 Proactivizing Threshold Signatures

Proactive Secret Sharing (PSS) as it has come to be known, has seen a number of realizations for different ranges of parameters since the introduction of the mobile adversary model [OY91]. In fact, even proactive signature schemes themselves have been studied directly [ADN06, FGMY97]. A naive adaptation of any off-the-shelf PSS scheme to the threshold signature setting would in many cases yield proactive threshold signature schemes immediately. However, heavy use of an honest majority by most PSS schemes would already rule out many practical applications of such an approach. Moreover all such solutions will have communication patterns that require every party in the system to be online at pre-defined times, at the close of *every* epoch, in order to keep the system proactivized and moving forward.

To see why requiring all parties to be online simultaneously is not reasonable especially for threshold wallets, consider the following scenarios:

- **Cold storage:** Alice splits her signing key between her smartphone and laptop and has them execute a threshold signing protocol when a message is to be signed. However if for any number of operational reasons one of the devices (say her smartphone) malfunctions, the secret key is lost forever and any funds associated with the corresponding public key are rendered inaccessible. In order to avoid this situation, Alice stores a third share of the signing key in a secure *cold storage* server. While this third share does not by itself leak the signing key, along with the laptop it can aid in the restoration of the smartphone's key share when required. In this scenario it would be quite inconvenient (and also defeat

the purpose of two-party signing) if the cold storage server has to participate in the proactivization every time the system needs to be re-randomized; it would be much more reasonable to have the smartphone and laptop proactivize when required, and send update packages to the server.

- **(2,3)-factor authentication:** Alice now splits her signing key across her smartphone, laptop, and tablet so that she must use any two of them to sign a message. Even in this simple use case, having all of her devices online and active simultaneously (possibly multiple times a day) just so that they can refresh would be cumbersome. Ideally every time she uses two of them to sign a message, they also refresh their key shares and leave an update package for the offline device to catch up at its leisure.
- **Concurrent use:** Alice, Bob, Carol, and Dave are executives at a corporation, and at least two of them must approve a purchase funded by the company account. This is enforced by giving each of them a share of the signing key, so that any two may collaborate to approve a transaction. Requiring them all to be online simultaneously is impractical given their schedules; it would be much more convenient to have any two of them refresh the system when they meet to sign, and send updates to the others.

Correlated Risks Beyond convenience, there are qualitative security implications for the de-facto standard pattern of proactivization. In particular, the validity of the assumption that an adversary controls only up to a threshold number of devices hinges on the risk of compromise of each device being independent. However having all devices in the system come online at frequent pre-specified points in time and connect to each other to refresh may significantly correlate their risk of compromise. Instead it would be preferable that only the minimal number of devices (i.e. the signing threshold) interact with each other in the regular mode of operation, and enable the system to non-interactively refresh itself.

The ideal communication pattern alluded to above is the following: in a (t, n) proactive threshold signature scheme, any t parties are able to jointly produce all the necessary components to refresh the system, and send the relevant information to offline parties. When an offline party wakes up, it processes the messages received and is able to "catch up" to the latest sharing of the secret.

1.2 Challenges in Realizing this Pattern

While this communication pattern sounds ideal, a whole host of subtle issues arise in potential realizations. For instance, in the Cold Storage case, how does the server know that the updates it receives are "legitimate"? An attacker controlling Alice's smartphone could spoof an update message and trick the server into deleting its key share and replacing it with junk.

Due to the inherent unfairness of two-party/dishonest majority MPC protocols, an adversary can obtain the output of the computation while depriving honest parties of it. In this spirit, the smartphone (acting for the attacker) could work with the laptop until it obtains the "update" message to send to the server, but abort the computation before the laptop gets it. Now the attacker has the ability to convince the server to delete its old share by using this message, whereas the laptop has no idea whether the attacker will actually do this (and therefore doesn't know whether to replace its own key share).

Implicit in these scenarios is the problem of *unanimous erasure*:

> How can we design a proactivization protocol in which the adversary can not convince an honest party to prematurely erase its secret key share?

In the $(2, 2)$ case even a network adversary (who does not control either party) can induce premature deletion by simply dropping a message in the protocol. Moreover is it possible to restrain such a proactivization procedure to be *minimally invasive* to the threshold wallet? i.e. native to usage patterns and protocol structures of threshold wallets.

1.3 Our Contributions

In this work we give a comprehensive treatment of the notion of proactive security with offline-refresh, with our study progressing in four phases:

1. **Defining Offline Refresh.** We formalize the notion of offline refresh for threshold protocols in the Universal Composability (UC) framework [Can01], and justify why our definition (unanimous erasure) is the correct one. Our starting point is the definition of Almansa et al. [ADN06] which we build on to capture that all parties need not be in agreement about which epoch they are in, and that an adversary can change corruptions while other parties are offline. Intuitively previous definitions have had an inherent synchrony in the progress of the system, which we remove in ours and show how to capture that parties may refresh at different rates.

2. **Upgrading $(2, n)$ Schemes.** We show how to upgrade $(2, n)$ threshold Schnorr-like signature schemes to proactive security tailored for use with a threshold wallet, in that it makes use of transactions posted to the blockchain for synchronization purposes. We make the case in Section 4.2 that the power of a ledger is necessary for this task. Our refresh protocol adds no extra assumptions, incurs very little overhead as compared to running the threshold signature itself, and exactly matches the *ideal* communication pattern outlined in the previous section.

3. **Proactive Multiplication.** We construct a mechanism to proactivize OT Extension state. This allows us to proactivize even threshold ECDSA protocols, which are sophisticated due to the non-linear signing equation. We prove the efficiency of our construction by means of an implementation, specifically the overhead incurred in computational time of our refresh procedure is roughly 24% for the ECDSA protocol of Doerner et al. [DKLs19] and 14% in the case of Gennaro and Goldfeder [GG18], while the communication round overhead is zero in both cases.

4. **Impossibility of Online Dishonest Majority for $(3, n)$ and Beyond.** Intuition would strongly suggest that any (t, n) threshold scheme could also be upgraded to proactive security with offline refresh in the presence of a dishonest majority online using sufficiently heavy cryptographic hammers. However, surprisingly we show this intuition to be false; i.e. even assuming arbitrary trusted setup/random oracle and an ideal ledger, *there must be an honest majority online* to refresh the system. We prove this result by developing new elegant techniques to reason about security in this setting.

We therefore formulate the problem of offline refresh and address the most pressing practical and theoretical questions: the honest majority online case is simple, the $(2, n)$ case permits a novel efficient protocol with a ledger which we implement, and the (t, n) case for $t > 2$ must necessarily have an honest majority of participants online.

Broader Implications Our results can be interpreted as positive for small-scale decentralization, eg. 2FA across personal devices. In particular the $(2, n)$ refresh protocol is readily compatible with existing implementations of threshold wallets, and essentially comes at only the cost of implementing forward-secure channels. However our impossibility result rules out this strong form of security for larger scale systems, where many servers hold shares of a secret with a high reconstruction threshold. In those cases system designers who desire proactive security must account for the cost of either bringing an honest majority online, or waiting to hear from all parties before progressing epochs.

1.4 Our Techniques

We first sketch the ideas behind our $(2, n)$ construction, and then discuss how to reason about the general (t, n) case and show impossibility.

1.4.1 $(2, n)$ Construction

Roughly, our approach is to use private channels to communicate candidate refresh packages, and the public ledger to achieve consensus on which one to use. We take advantage of the fact that threshold wallets already rely on posting signatures to a public ledger in order to coordinate these refreshes. Let each party P_i own point $f(i)$ on a shared polynomial f where $f(0) = \mathsf{sk}$ (i.e. standard Shamir sharing of the secret key sk). We

have parties generate a candidate refresh polynomial f' when they sign a message, associate each signature with f', and "apply" the refresh (i.e. replace $f(i)$ with $f'(i)$) when the corresponding signature appears on the blockchain. While this handles the coordination part, the major issue of verifiably communicating $f'(j)$ to offline party P_j remains a challenge. To solve this, we have the online refreshing parties jointly generate a *local* threshold signature authenticating f' when communicated to each offline party; such a signature can only be produced by two parties working together, so any candidate f' received when offline must have been created with the approval of an honest party.

Working Around Unfairness Note that this approach is still vulnerable to attacks where the adversary withholds the threshold signature from an honest party in the protocol; if an online signing protocol aborts, how does an honest party know if its (possibly malicious) signing counterparty sent f' and the corresponding signature to offline parties? This is an issue that stems from the inherent unfairness of two-party computation. While this is impossible to solve in general, we observe that most threshold ECDSA/Schnorr signature protocols are simulatable so the signing nonce R is leaked, but the signature itself stays hidden until the final round. We exploit this fact to bind each f' to R instead of the signature itself; so our proactive version of threshold ECDSA/Schnorr will proceed as follows:

1. Run the first half of threshold ECDSA/Schnorr to obtain R.
2. Sample candidate f', bind it to R, threshold-sign these values and send them to offline parties.
3. Continue with threshold ECDSA/Schnorr to produce the signature itself.

Correspondingly when *any* signature under R appears on the blockchain, each party searches for a bound f' that it can apply. With overwhelming probability there will never be two independently generated signatures that share the same R nonce throughout the lifetime of the system.

Threshold ECDSA and Multipliers Threshold ECDSA protocols require use of a secure two-party multiplication functionality $\mathcal{F}_{\mathsf{MUL}}$ (or equivalent protocol) due to its non-linear signing equation. Indeed, recent works [GG18, LNR18, DKLs19] have constructed practical threshold ECDSA protocols that make use of multipliers that can be instantiated with either Oblivious Transfer or Paillier encryption. Using these multipliers is significantly more efficient in the offline-online model where parties run some kind of preprocessing in parallel with key generation, and make use of this preprocessed state for efficient $\mathcal{F}_{\mathsf{MUL}}$ invocation when signing a message (this is done by all cited works). However as this preprocessed state is persistent across $\mathcal{F}_{\mathsf{MUL}}$ invocations, it becomes an additional target to defend from a mobile adversary. We show how to efficiently re-randomize this preprocessed state for OT-based instantiations of $\mathcal{F}_{\mathsf{MUL}}$, and therefore get offline-refresh proactive security for $(2, n)$ threshold ECDSA in its entirety. Our proactivization of $\mathcal{F}_{\mathsf{MUL}}$ makes novel use of the classic technique of Beaver [Bea95] to preprocess oblivious transfer, in combination with the mechanism we build to deliver updates securely.

1.4.2 General (t, n) Impossibility

We develop a novel technique to reason about the security of protocols that tolerate mobile corruptions. We first prove that any refresh protocol that tolerates an online dishonest majority must have the property that a minority of online parties holds enough information to allow any offline party to refresh. Subsequently we show that a mobile adversary can exploit this property to derive the refreshed private state of a previously corrupt offline party even after it is un-corrupted. The proof is built up from this underlying insight, discussed further in Section 10.

1.5 Related Work

The notion of mobile adversaries with a corresponding realization of proactive MPC was first introduced by Ostrovsky and Yung [OY91]. Herzberg et al. [HJKY95] devise techniques for proactive secret sharing, subsequently adapted for use in proactive signature schemes by Herzberg et al. [HJJ+97]. Cachin et

al. [CKLS02] show how to achieve proactive security for a shared secret over an asynchronous network. Maram et al. [MZW+19] construct a proactive secret sharing scheme that supports dynamic committees, with a portion of the communication done through a blockchain. For a more comprehensive survey, we refer the reader to the works of Maram et al. [MZW+19] and Nikov and Nikova [NN05].

Very recently Benhamouda et al. [BGG+20] and Goyal et al. [GKM+20] introduced a protocol in which a committee (elected from a larger set of parties) runs what is essentially a proactivization with offline-refresh. However they work in the setting of an honest majority, and their techniques are tailored as such.

The work of Canetti et. al. [CHH00] solves the problem of an offline node regaining the ability to authenticate its communication after having suffered a break-in. However the settings are incomparable; our network model is stronger in that we assume authenticated communication (details in Section 4), but weaker in another dimension as we do not rely on an honest majority among online parties. Our use of the ledger is merely as a passive public signalling mechanism, and not as interactive party-specific storage (eg. no issuing of certificates to individual parties).

As discussed earlier, *every* existing work (including those since the above mentioned surveys) assumes either that all parties come online [CGG+20], an honest majority of parties collaborate in order to proactivize the system [BGG+20, GKM+20], or that corruptions are passive [EOPY18]. Additionally they require this honest majority of parties to come online simultaneously at pre-specified points in time to run the refresh protocol. As the entire premise of the (t, n) threshold signature setting is that only t parties need be online simultaneously to use the system,

- For the $(2, n)$ case we impose as a strict requirement that only two parties be sufficient to proactivize the system. Consequently as it is meaningless to have an honest majority among two parties, we can not directly apply techniques from previous works to our setting. To our knowledge the conceptual core of our protocol– a threshold signature (internal to the system) interleaved with a threshold signature that appears on the blockchain, is novel.
- For the general $t > 2$ case, we prove that the weakest possible notion of dishonest majority for proactivization, i.e. refresh with $2t - 1$ online parties, is impossible to achieve.

Therefore we give a comprehensive treatment of proactivization with an online dishonest majority, which has not previously been studied in the literature.

1.6 Organization

We first present the definitions we use in Section 2. We then give our formalization of mobile adversaries and offline refresh in Section 3, following which we detail our threshold signature abstraction in Section 5. We begin by introducing the protocol to coordinate simple (2,2) key refresh in Section 6, and then give the extension to $(2, n)$ proactive threshold signatures in Section 7. Following this, we show how to proactivize every component of the more sophisticated recent ECDSA protocols in Section 8. We demonstrate the practicality of our protocols by implementation to augment two different ECDSA protocols, the results of which we present in Section 9. Finally we prove the impossibility of offline refresh with a dishonest online majority for larger thresholds in Section 10.

2 Preliminaries

Throughout this paper, we fix the corruption threshold as $t = 1$ and hence formulate all of our definitions assuming one malicious adversarial corruption.

Network Model We assume a synchronous network, as already required by recent threshold signature schemes [GG18, LNR18, DKLs19]. For the blockchain model, we follow the synchronous functionality of Kiayias et al. [KZZ16]. In this functionality, the blockchain only progresses after all parties finish their current round, therefore parties are always synchronized during the protocol run.

Additionally, we make the necessary assumption of proactive channels that support delivery to offline parties, discussed further in Section 4.1.

Protocol Input/Output Notation for $(2, n)$ setting The $(2, n)$ protocols in this paper are described for any pair of parties indexed by $i, j \in [n]$. In particular, any two parties P_i, P_j out of a group of n parties $\vec{\mathsf{P}}$ can run a protocol π with private inputs x_i, x_j to get their private outputs y_i, y_j respectively. For ease of notation since all of our protocols have the same instructions for each party, we choose to describe them as being run by P_b with P_{1-b} as the counterparty. The general format will be

$$y_b \leftarrow \pi(1 - b, x_b)$$

to denote that P_b gets output y_b by running protocol π with input x_b and counterparty P_{1-b}. For instance if π is run between P_2 and P_6, the protocol as described from the point of view of P_6 is interpreted with $b \equiv 6$ and $1 - b \equiv 2$.

Ideal Functionalities We assume access to a number of standard ideal functionalities: $\mathcal{F}_{\mathsf{Com}}$ (commitment), $\mathcal{F}_{\mathsf{Com\text{-}ZK}}^{R_{\mathsf{DL}}}$ (commited proof of knowledge of discrete logarithm), $\mathcal{F}_{\mathsf{Coin}}$ (coin tossing), $\mathcal{F}_{\mathsf{MUL}}$ (two-party multiplication) given formally in Appendix C.

Adversarial Model We prove our protocols secure in the Universal Composability (UC) framework of Canetti [Can01]. We give the specifics of our modelling in Section 3.

2.1 Blockchain Model

We detail here the relevant aspects of the underlying blockchain system that is required for our $(2, n)$ protocol.

A Transaction Ledger Functionality A transaction ledger can be seen as a public bulletin board where users can post and read transactions from the ledger. As it was shown in [GKL15], a ledger functionality must intuitively guarantee the properties of *persistence* and *liveness*, that we informally discuss next.

- *Persistence*: Once a honest user in the system announces a particular transaction as *final*, all of the remaining users when queried will either report the transaction in the same position in the ledger or will not report any other conflicting transaction as stable.
- *Liveness*: If any honest user in the system attempts to include a certain transaction into their ledger, then after the passing of some time, all honest users when queried will report the transaction as being stable.

We encapsulate the ledger in a functionality $\mathcal{G}_{\mathsf{Ledger}}$ inspired by the functionality of [KZZ16].

On the Supported Type of Ledgers. For simplicity, we present our results on a synchronous public transaction ledger (e.g., Bitcoin [Nak09] or Ethereum [Woo]) where there is a known delay for the delivery of messages. We note however that synchrony of the ledger is not a necessary assumption for our protocol. In fact, any ledger satisfying the standard properties of *persistence* and *liveness* as defined in [GKL15] can be employed by our protocol. As it was shown in [GKL15], Bitcoin satisfies both properties for an honest majority of mining power under the assumption of network *synchrony*. However, if one is willing to trade off the honest majority assumption for a partially synchronous network,[1] we point out that partially synchronous Byzantine Fault Tolerant (BFT) ledgers such as Algorand [CM19] can also be employed by our protocol due to how we define the corruption model, where the adversary "waits" for a full refresh before changing corruptions.

[1] It is a well known fact that it is impossible to achieve consensus on partially synchronous networks under honest majority [DLS88].

Without loss of generality we assume that every transaction that is included in the chain becomes final and will not be rolled-back. For a more detailed discussion we refer the reader to [BMTZ17]. We give a formal definition of the ledger functionality below.

Functionality 1: $\mathcal{G}_{\mathsf{Ledger}}$

The functionality $\mathcal{G}_{\mathsf{Ledger}}$ is globally available to all participants. The functionality is parameterized by a function $\mathsf{Blockify}$, a predicate $\mathsf{Validate}$, a constant T, and variables chain, slot, $\mathsf{clockTick}$ and buffer, and a set of parties $\mathcal{P}$. Initially set $\mathsf{chain} := \varepsilon$, $\mathsf{buffer} := \varepsilon$, $\mathsf{slot} := 0$ and $\mathsf{clockTick} := 0$.

- Upon receiving $(\texttt{Register}, \mathsf{sid})$ from a party P, set $\mathcal{P} := \{\mathcal{P}\} \cup P$ and if P was not registered before set $d_P := 0$. Send $(\texttt{Register}, \mathsf{sid}, P)$ to $\mathcal{A}$.
- Upon receiving $(\texttt{ClockUpdate}, \mathsf{sid})$ from some party $P_i \in \mathcal{P}$ set $d_i := 1$ and forward $(\texttt{ClockUpdate}, \mathsf{sid}, P_i)$ to $\mathcal{A}$. If $d_P = 1$ for all $P \in \mathcal{P}$ then set $\mathsf{clockTick} := \mathsf{clockTick} + 1$, reset $d_P := 0$ for all $P \in \mathcal{P}$ and execute *Chain extension*.
- Upon receiving $(\texttt{Submit}, \mathsf{sid}, \mathsf{tx})$ from a party P, If $\mathsf{Validate}(\mathsf{chain}, (\mathsf{buffer}, \mathsf{tx})) = 1$ then set $\mathsf{buffer} := \mathsf{buffer}||\mathsf{tx}$.
- Upon receiving $(\texttt{Read}, \mathsf{sid})$ from a party $P \in \{\mathcal{A} \cup \mathcal{P}\}$, If P is honest then set $b := \mathsf{chain}$ else set $b := (\mathsf{chain}, \mathsf{buffer})$. Then return the message $(\texttt{Read}, \mathsf{sid}, b)$ to party P.
- Upon receiving $(\texttt{Permute}, \mathsf{sid}, \pi)$ from $\mathcal{A}$ apply permutation π to the elements of buffer.

Chain extension: If $|\mathsf{clockTick} - (\mathsf{T} \cdot \mathsf{slot})| > \mathsf{T}$ then set $\mathsf{chain} := \mathsf{chain}||\mathsf{Blockify}(\mathsf{slot}, \mathsf{buffer})$ and $\mathsf{buffer} := \varepsilon$, and subsequently send $(\texttt{ChainExtended}, \mathsf{sid})$ to $\mathcal{A}$.

The functionality $\mathcal{G}_{\mathsf{Ledger}}$ is parameterised by a set $\mathcal{P}$ of participants P; for a new participant to join the protocol it must send a message `Register` to the $\mathcal{G}_{\mathsf{Ledger}}$ functionality. We parameterise $\mathcal{G}_{\mathsf{Ledger}}$ by a constant T that denotes the gap in clock tick units between two subsequent slots in the ledger. Without loss of generality, one could assume the existence of a function $\mathsf{Tick2Time}$ that maps clock ticks to physical time, in the same spirits of [KZZ16]. For concreteness, in such a case, the value of T would be 10 minutes in Bitcoin.

The functionality $\mathcal{G}_{\mathsf{Ledger}}$ is synchronous, and the $\mathsf{clockTick}$ variable is incremented only after all the parties send a message `ClockUpdate` to $\mathcal{G}_{\mathsf{Ledger}}$. A new block is created and appended to the chain only after T clock ticks have elapsed since the last block creation; in the meantime, parties can submit new transactions to the ledger with the message `Submit`, and read all the contents of the ledger with the message `Read`. The adversary $\mathcal{A}$ can permute the contents of the current transaction buffer, which translates to rearranging the order of the transactions that will be included in the next block.

We define the predicate $\mathsf{Validate}$ that validates the transactions contents and format against the current chain before including it in the transactions buffer. In existing systems such as Bitcoin, the $\mathsf{Validate}$ predicate checks the signature of the user spending funds. The function $\mathsf{Blockify}$, as in [KZZ16], handles the processing of the transaction buffer and "packs" it nicely into blocks.

Global Functionality The simulator for our protocol will not be able to act on behalf of $\mathcal{G}_{\mathsf{Ledger}}$. In particular the simulator is only able to use the functionality with the same priviliges as a party running the real protocol.

2.2 Miscellaneous

We use $(\mathbb{G}, G, q)$ to denote a curve group; the curve $\mathbb{G}$ is generated by G and is of order q. Throughout the paper we use additive notation for curve group operations.

Lagrange Coefficients $\lambda_i^j(x), \lambda_j^i(x)$ are the Lagrange coefficients for interpolating the value of a degree-1 polynomial f at location x using the evaluation of f at points i and j. In particular,

$$\lambda_i^j(x) \cdot f(i) + \lambda_j^i(x) \cdot f(j) = f(x) \qquad \forall x, i, j \in \mathbb{Z}_q$$

Each $\lambda_i^j(x)$ is easy to compute once i, j, x are specified.

Signature format A signature under public key pk comprises of (R, σ) where $R \in \mathbb{G}$ and $\sigma \in \mathbb{Z}_q$. Note that in practice, some ECDSA/Schnorr implementations will only contain the x-coordinate of R instead of the whole value. This is only done for efficiency reasons with no implications for security, and does not affect compatibility with our protocols.

3 Defining Offline Refresh

A notion of offline refresh that is not a priori too restrictive or offers too weak a security guarantee is tricky to define. Existing definitions (eg. [ADN06]) require that the refresh procedure always terminate successfully when honest parties receive the instruction. This can be viewed as the proactive analog of the well-studied MPC notion of Guaranteed Output Delivery (GOD). It is immediate from foundational results on dishonest majority coin tossing [Cle86] that if there is no honest majority involved in the refresh procedure that achieves GOD, then the resulting randomness for proactivization is succeptible to unacceptable bias.

One may consider instead a proactive analog of the MPC notion of security with abort. This notion allows the adversary to abort the computation if it so desires, possibly receiving output while depriving honest parties of it. Efficient dishonest majority MPC protocols that achieve security with abort are known in the literature [DPSZ12, KOS16] indicating that this notion may be the correct one.

However one must be careful when defining exactly what power to allow the adversary in aborting the refresh procedure. Security with abort in the standard MPC setting comes with a fine-grained separation between *selective* and *unanimous* abort [FGH$^+$02], the difference being that in the former some honest parties may get output while others not, while in the latter all honest parties agree on whether or not to abort. In standard MPC protocol design the choice between these two security notions offers a meaningful tradeoff: selective abort while offering strictly weaker security is sufficient for many applications, and is much more efficient in round complexity and/or use of broadcast [GL05]. When translated to the setting of proactive security however we argue that this distinction is much more drastic, to the point of making selective abort patently undesirable.

Refresh with selective abort is insufficient Consider the following adaptation of security with selective abort: at the end of the refresh protocol, the adversary has the power to choose exactly which (honest) parties successfully advance to the next epoch. This gives the adversary the power to execute attacks on the honest parties' private state that were not feasible without the proactivization protocol. In particular an adversary could for instance convince one half of the honest parties to advance to the next epoch while the remaining honest parties do not. As the parties that advance erase their state from the previous epoch, their secrets will no longer be correlated with the parties that do not advance. This means that even if the system has an honest majority of parties (which in the static setting means the shared secret can always be reconstructed/used if desired), the refresh procedure gives the adversary a window to throw the parties out of sync and 'erase' the common secret from the system's distributed state.

Concretely this could translate to attacks where a single malformed message or network issue causes a threshold wallet to permanently erase the common secret key, which in many cases could mean an irreversible loss of funds.

Refresh with unanimous erasure We settle on 'unanimous erasure' as the correct definition for proactive security, as the analog of security with unanimous abort. Informally, this means that the adversary has the power to decide whether or not to move to the next epoch, but crucially *all honest parties agree on the epoch*

with the caveat that they may not be activated synchronously. Offline refresh is captured by allowing the adversary to advance the epoch arbitrarily many times (and even change corruptions) without activating *all* honest parties, however any honest party if activated must 'catch up' non-interactively to the current epoch.

Corruption Caveats Defining a meaningful model that allows different parties to stay "offline" (and therefore effectively exist in different epochs at the same time) while simultaneously honouring the assumption that only a threshold number of parties are corrupt at any given epoch requires particular care. We handle this issue by requiring that the adversary allows a party to "update" before corrupting it. While this appears to weaken the model, a definition without this restriction would be inherently unachievable, as an adversary would be able to effectively "travel in time". For instance, if some party P is offline from epoch i onward, an adversary who corrupts it after the the system has progressed to epoch $i+1$ will obtain this party's state at epoch i even after that epoch has passed. This would be problematic if the adversary had already corrupted (and subsequently uncorrupted) $t-1$ different parties at epoch i, as gaining P's state for epoch i will completely reveal the system's secrets, all without violating the assumption that only $t-1$ parties may be corrupt at any given point in time. See the paragraph on *Corruptions* in the formal definition that follows for further discussion.

Parameters The system consists of n parties, of which t are necessary to `operate` by accessing the secret. The adversary may corrupt at most $t-1$ parties. The `refresh` procedure is run by activating t_ρ parties.

With these security notions in mind, we formalize the definition of proactive security with unanimous erasure and offline refresh in the UC model, and defer the technical details to Appendix D.

4 Instantiating Offline Refresh

With the model and definitions in place, we now incrementally work towards our protocol via a sequence of stepping stones to introduce which tools we use and why.

4.1 Simple Honest Majority Instantiation

We begin by sketching a 'baby protocol' for proactive secret sharing with $t_\rho = 2t-1$ and $n = t_\rho + 1$, i.e. where the refresh protocol is run by an honest majority of online parties and one party (labelled P_{off}) stays offline.

Network It is immediate that a necessary underlying assumption is a forward secure channel that supports delivery to offline parties. Formally, this is captured by having offline parties accumulate messages in a buffer that they read when they become online. In practice an offline party may not literally be disconnected from the network and need a buffer, just that the refresh protocol does not require its participation. Alternatively message delivery may be aided by a server as in the Signal protocol [MP, ACD19]. We assume that the t_ρ online parties share a broadcast channel (which is not necessarily visible to P_{off}).

Cryptographic tools As a parameter of the protocol, parties agree on an elliptic curve $\mathbb{G}$ generated by G and of order q, where the Discrete Logarithm problem is assumed to be hard. We assume two protocol primitives:

- $\pi^{\mathsf{DKG}}_{\mathsf{Setup}}$ is a protocol where at the end each party P_j holds $\mathsf{sk}_j = f(j) \in \mathbb{Z}_q$ where f is a degree $t-1$ polynomial with the common secret defined as $\mathsf{sk} = f(0)$. Additionally every party knows $\mathsf{pk}_j = F(j) = f(j) \cdot G$ for each $j \in [n]$. This is a common tool [Fel87, Ped91] and we recall a canonical instantiation in Appendix B.
- $\mathsf{Reshare}(i)$ is a protocol run by t_ρ parties each of whom have local secret shares $f(j)$ and public shares $F(j)$ as created by $\pi^{\mathsf{DKG}}_{\mathsf{Setup}}$ above, in order to create a fresh and independent sharing of the same format

where the secret is $f(i)$. In particular, at the end of $\mathsf{Reshare}(i)$, each party P_j holds $f'(j) \in \mathbb{Z}_q$ where f' is a degree $t-1$ polynomial with $f'(0) = f(i)$. This is a common tool as well, and so we refer the reader to Gennaro et al. [GRR98] for further details.

As before, let $\mathsf{off} = t_\rho + 1$ index the offline party. The refresh protocol is run among t_ρ online parties as follows:

1. Parties $P_1, \cdots P_{t_\rho}$ run $\mathsf{Reshare}(0)$ in order to obtain fresh shares the secret key, i.e. they agree on a public degree $t-1$ polynomial F over $\mathbb{G}$ and each P_j obtains $f(j)$ such that $f(j) \cdot G = F(j)$. It holds that $F(0) = \mathsf{pk}$. They overwrite $\mathsf{sk}_j = f(j)$ and $\mathsf{pk}_j = F(j)$ for each $j \in [n]$.
2. They then run $\mathsf{Reshare}(\mathsf{off})$ to jointly sample a fresh degree $t-1$ polynomial f' such that $f'(0) = \mathsf{sk}_{\mathsf{off}}$ and each P_j knows $f'(j)$ and every public $F'(j) = f'(j) \cdot G$.
3. Each P_j for $j \in [t_\rho]$ sends $\vec{\mathsf{pk}} = (\mathsf{pk}_j)_{j \in [t_\rho]}, f'(j), F'$ privately to P_{off}.

It holds that since there are t honest parties who execute the final step, upon waking up P_{off} will find at least t messages that agree on $\vec{\mathsf{pk}}, F'$ accompanied by as many correct evaluations $f'(j)$ which can be verified by checking $f'(j) \cdot G \stackrel{?}{=} F'(j)$. Note that since there are at most $t-1$ malicious parties, they can't collude to create a sufficiently large set to fool P_{off}. It is immediate that P_{off} can therefore interpolate the correct $\mathsf{sk}_{\mathsf{off}}$ and 'catch up' on all the refreshes that it missed. This protocol can easily be extended for an arbitrary number of offline parties by generating a new reshared polynomial for each of them.

Hence we have shown that offline refresh is easy to satisfy in the presence of an online honest majority.

4.2 Dishonest Majority with Offline Broadcast

Folklore techniques such as Cleve [Cle86] give strong evidence that unanimous erasure in a $(2, 3)$ system is impossible to achieve over private channels alone. We give a rough sketch here as to why this is the case.

Consider a system comprising $P_0, P_1, P_{\mathsf{off}}$ in which P_{off} is offline, one of P_0 or P_1 may be corrupt, and the honest party and P_{off} must either agree on a random bit (successful termination) or agree to abort. The non-degeneracy requirement is that an honest execution does not induce an abort. Additionally the parties have access to arbitrary correlated randomness generated in some offline phase, which rules out direct application of the $t < n/3$ consensus lower bound [PSL80]. This system and its constraints captures a simplified notion of unanimous erasure.

We will argue that if P_0 is corrupt, then P_1 and P_{off} can not meet the constraints of the system. Observe that in the event of successful termination the private communication from P_0 to P_{off} is by itself sufficient to 'convince' P_{off} not to abort; if this were not true then a corrupt P_1 could simply erase its entire private channel, which forces P_{off} to abort while honest P_0 who is unaware of this terminates successfully. We call a transcript from either one of P_0 or P_1 to P_{off} as 'convincing' if it induces P_{off} to terminate successfully with an output bit instead of aborting. Without loss of generality there must be some round in the protocol where P_0 gains the ability to produce a convincing transcript, but P_1 has not yet acquired this ability (either party having this ability from round 0 would clearly admit trivial attacks). Therefore if P_0 simply halts the protocol with P_1 at this point, P_1 will have no way of knowing whether P_0 will choose to convince P_{off} to abort or to terminate successfully.

Offline Broadcast In order to overcome this challenge we introduce a powerful notion of an 'offline broadcast channel', which is a broadcast channel shared by $P_0, P_1, P_{\mathsf{off}}$ but crucially is invisible to the adversary if none of the parties are corrupt. Our final protocol will not use so strong a tool, but it provides an instructive stepping stone.

Leaking the Difference Polynomial We observe that *any* proactivization protocol where an adversary corrupts t parties has the following property: define $f_\delta(i) = f'(i) - f(i)$, i.e. the polynomial that encodes the difference between old and new shares. Given $f(i), f'(i)$ for any $t-1$ values of i (which the adversary

has by virtue of corrupting $t-1$ parties) one can compute $f_\delta(x)$ for any x. This is because $f_\delta(0) = 0$ (as $f(0) = f'(0)$) and f_δ is a degree $t-1$ polynomial of which one now has t points.

Given an offline broadcast channel, designing a refresh protocol for $P_0, P_1, P_{\mathsf{off}}$ using the above observation is as simple as sampling the difference polynomial on the broadcast channel. In particular the refresh protocol proceeds as follows:

1. P_0 samples a uniform $f_{\delta,0}$ and offline-broadcasts a commitment to $f_{\delta,0}$.
2. P_1 samples a uniform $f_{\delta,1}$ and offline-broadcasts it.
3. P_0 decommits f_δ on the offline-broadcast channel.
4. Each party (either immediately, or upon waking up) defines $f_\delta = f_{\delta,0} + f_{\delta,1}$ and updates its local share as $f'(i) = f(i) + f_\delta(i)$

It is clear that the above offline refresh protocol tolerates a mobile malicious adversary that corrupts at most one party at any given time (which is optimal in a $t = 2$ system). In particular the offline broadcast channel allows for the following properties:

- The online parties and P_{off} use the same criteria to compute f_δ and so are always in agreement.
- Since the offline broadcast channel is invisible to the adversary when switching corruptions, the uniform choice of f_δ ensures that the resulting refreshed polynomial is distributed independently of any parties' view from earlier.

Unfortunately this offline broadcast primitive is an unreasonably strong assumption to make in practice. Broadcast is either implemented via interactive protocols, or inherently public when using a ledger/blockchain. We therefore carefully design a protocol that somewhat achieves the effect of this offline broadcast channel; we will use *private channels to communicate candidate f_δ values along with a public ledger to reach agreement on whether or not to use them*, and rely on the intrinsic entropy of certain common threshold signatures to bind the public and private components.

A Note on Parameters As our subsequent constructions are explicitly for $t = t_\rho = 2$, we drop the t, t_ρ notation until we revisit the general multiparty setting in Section 10.

5 Threshold Signature Abstraction

A threshold signature scheme [Des87] allows the power of producing a digital signature to be delegated to multiple parties, so that a threshold number of them must work together in order to produce a signature. Specifically a (t, n) signature scheme is a system in which n parties hold shares of the signing key, of which any t must collaborate to sign a message. In this section we focus on $(2, n)$ threshold versions of the ECDSA [Kra93] and Schnorr [Sch89] Signature schemes. As our techniques are general and not specific to any one threshold signature scheme, we use an abstraction of such protocols for ease of exposition.

5.1 Abstraction

We assume that a $(2, n)$ threshold signature over group $(\mathbb{G}, G, q)$ can be decomposed in a triple of algorithms $(\pi^{\mathsf{DKG}}_{\mathsf{Setup}}, \pi^{\mathsf{R}}_{\mathsf{Sign}}, \pi^{\sigma}_{\mathsf{Sign}})$ of the following formats:

- $(\mathsf{sk}_i \in \mathbb{Z}_q,\ \mathsf{pk} \in \mathbb{G}) \leftarrow \pi^{\mathsf{DKG}}_{\mathsf{Setup}}(\kappa)$
  This protocol is run with n parties and has each honest party P_i obtain public output pk and private output sk_i. In addition to this, there must exist a degree-1 polynomial f over $\mathbb{Z}_q$ such that $\forall i \in [n]$, $\mathsf{sk}_i = f(i)$.

- $(R \in \mathbb{G},\ \mathsf{state}_b \in \{0,1\}^*) \leftarrow \pi^{\mathsf{R}}_{\mathsf{Sign}}(\mathsf{pk}, \mathsf{sk}_b, 1-b, m)$
  Run by party P_b with P_{1-b} as counterparty, to sign message m. Both parties output the same R when honest, with private state state_b.

- $(\sigma \in \mathbb{Z}_q) \leftarrow \pi^{\sigma}_{\mathsf{Sign}}(\mathsf{state}_b)$
  Completes the signature started by $\pi^{\mathsf{R}}_{\mathsf{Sign}}$ when both parties are honest, i.e. σ verifies as a signature on message m with R as the public nonce and pk as the public key.

Note that $\pi^{\mathsf{DKG}}_{\mathsf{Setup}}$ captures a specific kind of secret sharing, i.e. the kind where the signing key is Shamir-shared among the parties. Multiplicative shares for instance are not captured by this abstraction. The (2,2) threshold ECDSA protocols of Lindell [Lin17] and Castagnos et al. [CCL+19] are not captured by our abstraction for this reason. Additionally signature schemes that do not have randomized signing algorithms such as BLS [BLS04] can not be decomposed as per this abstraction.

Finally these protocols must realize the relevant threshold signature functionality. In particular let $\mathsf{Sign} \in \{\mathsf{Sign}^H_{\mathrm{ECDSA}}, \mathsf{Sign}^H_{\mathrm{Schnorr}}\}$ where

$$\mathsf{Sign}^H_{\mathrm{ECDSA}}(\mathsf{sk}, k, m) = \frac{H(m) + \mathsf{sk} \cdot r_x}{k}$$
$$\mathsf{Sign}^H_{\mathrm{Schnorr}}(\mathsf{sk}, k, m) = H(R||m) \cdot \mathsf{sk} + k$$

where r_x is the x-coordinate of $k \cdot G$ in the ECDSA signing equation. We therefore define functionality $\mathcal{F}^{n,2}_{\mathsf{Sign}}$ to work as follows:

Functionality 2: $\mathcal{F}^{n,2}_{\mathsf{Sign}}$

This functionality is parameterized by the party count n, the elliptic curve $(\mathbb{G}, G, q)$, a hash function H, and a signing algorithm Sign. The setup phase runs once with n parties, and the signing phase may be run many times between (varying) subgroups of parties indexed by $i, j \in [n]$.

Setup On receiving (`init`) from all parties,

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

Signing On receiving $(\texttt{sign}, \mathsf{id}^{\mathsf{sig}}, (i,j), m)$ from both parties indexed by $i, j \in [n]$ $(i \neq j)$, if (`ready`) exists in memory but $(\texttt{complete}, \mathsf{id}^{\mathsf{sig}})$ does not exist in memory, then

1. Sample $k \leftarrow \mathbb{Z}_q$ and store it as the instance key.
2. Wait for $(\texttt{get-instance-key}, \mathsf{id}^{\mathsf{sig}})$ from both parties P_i, P_j.
3. Compute
$$R := k \cdot G$$
and send $(\texttt{instance-key}, \mathsf{id}^{\mathsf{sig}}, R)$ to parties P_i, P_j. Let $(r_x, r_y) = R$.
4. Wait for $(\texttt{proceed}, \mathsf{id}^{\mathsf{sig}})$ from both parties P_i, P_j.
5. Compute
$$\sigma := \mathsf{Sign}^H(\mathsf{sk}, k, m)$$
6. Send $(\texttt{signature}, \mathsf{id}^{\mathsf{sig}}, \sigma)$ to both parties P_i, P_j as adversarially-delayed private output.
7. Store $(\texttt{complete}, \mathsf{id}^{\mathsf{sig}})$ in memory.

To make concrete the role of each protocol ($\pi^{\mathsf{DKG}}_{\mathsf{Setup}}$,$\pi^{\mathsf{R}}_{\mathsf{Sign}}$,$\pi^{\sigma}_{\mathsf{Sign}}$), we restrict access of their corresponding simulators ($\mathcal{S}^{\mathsf{DKG}}_{\mathsf{Setup}}$,$\mathcal{S}^{\mathsf{R}}_{\mathsf{Sign}}$,$\mathcal{S}^{\sigma}_{\mathsf{Sign}}$) to $\mathcal{F}^{n,2}_{\mathsf{Sign}}$. Specifically $\mathcal{S}^{\mathsf{DKG}}_{\mathsf{Setup}}$ can only send (`init`) on behalf of a corrupt party and receive ($\texttt{public-key}, \mathsf{pk}$) in response. The messages ($\texttt{sign}, \mathsf{id}^{\mathsf{sig}}, (i,j), m$) and ($\texttt{get-instance-key}, \mathsf{id}^{\mathsf{sig}}$) can be sent and ($\texttt{instance-key}, \mathsf{id}^{\mathsf{sig}}, R$) received only by $\mathcal{S}^{\mathsf{R}}_{\mathsf{Sign}}$. Finally ($\texttt{proceed}, \mathsf{id}^{\mathsf{sig}}$) can be sent and ($\texttt{signature}, \mathsf{id}^{\mathsf{sig}}, \sigma$) received only by $\mathcal{S}^{\sigma}_{\mathsf{Sign}}$.

An implication of this restriction is that $\pi^{\mathsf{R}}_{\mathsf{Sign}}$ has to be simulatable without the signature σ, therefore it cannot leak any information about this value. (The approach of splitting the simulator into several simulators to limit what kind of information can be leaked in different stages of the protocol has been used before e.g., in secret-sharing based MPC protocols to claim that the protocol does not leak any information about the output until the reconstruction phase performed in the last round of the protocol). This abstraction was chosen deliberately to enforce this property; one of our key techniques in this work (Section 7) relies on $\pi^{\mathsf{R}}_{\mathsf{Sign}}$ keeping σ hidden.

Threshold Schnorr We recall a folklore instantiation of $\mathcal{F}^{n,2}_{\mathsf{Sign}}$ for $\mathsf{Sign}_{\mathrm{Schnorr}}$ in Appendix B (note that this also works for EdDSA).

Threshold ECDSA We note that the recent protocols of Gennaro and Goldfeder [GG18], Lindell et al. [LNR18], and Doerner et al. [DKLs19] for $\mathsf{Sign}_{\mathrm{ECDSA}}$ can also be cast in the above framework if required. However due to the non-linearity of $\mathsf{Sign}_{\mathrm{ECDSA}}$ the corresponding realization of $\mathcal{F}^{n,2}_{\mathsf{ECDSA}}$ requires use of a multiplication functionality $\mathcal{F}_{\mathsf{MUL}}$ (or equivalent protocol). Since $\mathcal{F}_{\mathsf{MUL}}$ is expensive to instantiate for one-time use, these threshold ECDSA protocols run some preprocessing for $\mathcal{F}_{\mathsf{MUL}}$ in parallel with $\pi^{\mathsf{DKG}}_{\mathsf{Setup}}$ and make use of this preprocessed state for more efficient online computation. As this adds additional persistent state to be protected against a mobile adversary, we need to deal with it carefully. We discuss this in further detail and give an efficient solution to this problem in Section 8.

6 Coordinating Two Party Refresh

As the final protocol combines two independent concepts: using the blockchain for synchronization, and authenticating communication to offline parties, we first present a base protocol for the former for a $(2,2)$ access structure and augment it with the latter to obtain a $(2,n)$ protocol. In this section, we describe the malicious secure protocol for two parties to coordinate an authenticated refresh of the secret key shares. The $(2,2)$ protocol is described with Shamir secret shares (points on a polynomial) rather than just additive shares so as to allow for a smoother transition to the $(2,n)$ setting.

Intuition The two parties begin by running the first half of the threshold signing protocol $\pi^{\mathsf{R}}_{\mathsf{Sign}}$ to obtain the signing nonce R that will be used for the subsequent threshold signature itself. They then sample a new candidate (shared) polynomial f' by publicly sampling the difference polynomial f_δ and store their local share $\mathsf{sk}'_b = f'(b)$ tagged with R and the epoch number epoch in a list rpool. Specifically rpool is a list of $(R, \mathsf{sk}'_b, \mathsf{epoch})$ values that are indexed by R as the unique identifying element. Following this, they complete the threshold signing by running $\pi^{\sigma}_{\mathsf{Sign}}$ and a designated party sends the resulting signature (and message) to $\mathcal{G}_{\mathsf{Ledger}}$, i.e. posts them to the public ledger.

Protocol 1: $\pi^{(2,2)}_{\rho\text{-}\mathsf{sign}}$

Parameters: Elliptic Curve Group $(\mathbb{G}, G, q)$
Parties: P_b, P_{1-b} (recall $b \in \{1,2\}$ is the index of the current party and $1-b$ is a shorthand for the index of the counterparty)
Ideal Oracles: $\mathcal{F}^{R_{\mathsf{DL}}}_{\mathsf{Com\text{-}ZK}}$, $\mathcal{G}_{\mathsf{Ledger}}$
Inputs:

- **Common**: Message to be signed $m \in \{0,1\}^*$, public key $\mathsf{pk} \in \mathbb{G}$, each party's share in the exponent

$\mathsf{pk}_b = \lambda_b^{1-b}(0) \cdot F(b)$ where F is the polynomial over $\mathbb{G}$ passing through $(0, \mathsf{pk})$ and $(b, f(b) \cdot G)$, epoch index $\mathsf{epoch} \in \mathbb{Z}^+$

- **Private**: Each party P_b has private input $\mathsf{sk}_b = \lambda_b^{1-b}(0) \cdot f(b) \in \mathbb{Z}_q$

---

1. **Tag R from Threshold Signature:**
   i. Run the first half of the threshold signing protocol
   $$(R, \mathsf{state}_b) \leftarrow \pi_{\mathsf{Sign}}^{\mathsf{R}}(\mathsf{sk}_b, 1-b, m)$$
2. **Sample New Polynomial:**
   i. Send $(\texttt{sample-element}, \mathsf{id}^{\mathsf{coin}}, q)$ to $\mathcal{F}_{\mathsf{Coin}}$ and wait for response $(\mathsf{id}^{\mathsf{coin}}, \delta)$
   ii. Define degree-1 polynomial f_δ over $\mathbb{Z}_q$ such that
   $$f_\delta(0) = 0 \quad \text{and} \quad f_\delta(1) = \delta$$
   iii. Compute
   $$\mathsf{sk}'_b = \mathsf{sk}_b + f_\delta(b)$$
3. **Store Tagged Refresh:**
   i. Retrieve Epoch index epoch
   ii. Append $(R, \mathsf{sk}'_b, \mathsf{epoch})$ to rpool
4. Complete the threshold signature protocol by running $\sigma \leftarrow \pi_{\mathsf{Sign}}^{\sigma}$
5. If $\sigma \neq \perp$ then set $\mathsf{tx} = (m, R, \sigma)$ and send $(\texttt{Submit}, \mathsf{sid}, \mathsf{tx})$ to $\mathcal{G}_{\mathsf{Ledger}}$

Note that in Step 5 it is sufficient for only one party to send the transaction tx to the ledger.

While the above protocol generates candidate refresh polynomials, choosing which one to use from rpool (and when to delete old shares) is done separately. The idea is that when a new block is obtained from $\mathcal{G}_{\mathsf{Ledger}}$ the parties each scan it to find signatures under their shared public key pk. The signatures are cross-referenced with rpool tuples stored in memory by matching R (no two signatures will have the same R) and the ones without corresponding tuples are ignored. If any such signatures are found, the one occurring first in the block is chosen to signal the next refresh; in particular the corresponding sk'_b overwrites sk_b stored in memory, rpool is erased, and the epoch counter is incremented.

Protocol 2: $\pi^{(2,2)}_{\rho\text{-update}}$

Parameters: Elliptic Curve Group $(\mathbb{G}, G, q)$
Parties: P_i (local refresh protocol)
Ideal Oracles: $\mathcal{G}_{\mathsf{Ledger}}$
Inputs: Epoch counter epoch, a list $\mathsf{rpool} = \{(\mathsf{epoch}, \mathsf{sk}'_i, R)\}$, private key share sk_i.

---

1. Send $(\texttt{Read})$ to $\mathcal{G}_{\mathsf{Ledger}}$ and receive $(\texttt{Read}, b)$. Set BLK to be the latest block occurring in b
2. Search for the first signature (σ, R) occurring in BLK under pk such that $\exists (R, \mathsf{sk}'_i, \mathsf{epoch}) \in \mathsf{rpool}$
3. Overwrite $\mathsf{sk}_i = \mathsf{sk}'_i$ and erase rpool
4. Set $\mathsf{epoch} = \mathsf{epoch} + 1$

It is clear that this protocol achieves all desired properties when both parties are honest. We give a proof of the extended $(2, n)$ protocol directly in the next section. However we make a few observations at this point that will aid in building the proof for the extended protocol.

Before and after a refresh the view of an adversary corrupting P_b when $\mathsf{epoch} = x$ is completely independent of the view when corrupting P_{1-b} after $\mathsf{epoch} = x + 1$. This is clear as polynomials f and f' are independently distributed, and so $\mathsf{sk}_b = f(b)$ can not be meaningfully combined with $\mathsf{sk}'_{1-b} = f'(1-b)$.

No two entries in rpool will have the same R by virtue of each R being chosen uniformly for each entry, the likelihood of there being two entries with the same R value in rpool is negligible, with about $\sqrt{q}$ signatures having to be generated before a collision occurs.

7 $(2, n)$ Refresh With Two Online

In this section, we give the malicious secure protocol for two online parties to coordinate an authenticated refresh of the secret key for arbitrarily many offline parties. We now describe how to ensure that offline parties can get up to speed upon waking up, crucially in a way that every party is in agreement about which polynomial to use so that sk_i erasures are always safe.

Goal Observe that if every party is in agreement about rpool, then the rest of the refresh procedure is deterministic and straightforward. Therefore it suffices to construct a mechanism to ensure that for each (R, $\mathsf{sk}'_b, \mathsf{epoch}$) tuple an online party P_b appends to its rpool, each offline party P_i is able to append a consistent value $(R, \mathsf{sk}'_i, \mathsf{epoch})$ to its own rpool. Here 'consistent' means that the points $(0, \mathsf{sk})$, (b, sk'_b), (i, sk'_i) are collinear.

An Attempt at a Solution We first note that since either one of the online parties P_b may be malicious and therefore unreliable, it simplifies matters to design the refresh protocol so that they both send the same message to an offline P_i. The message itself should deliver $f_\delta(i)$ (so that P_i can compute sk'_i) along with R. Simultaneously it must be ensured that a malicious party is unable to spoof such a message and confuse P_i.

In order to solve this problem, we take advantage of the fact that the parties already share a distributed key setup; as any two parties must be able to sign a message in a $(2, n)$ threshold signature scheme, we take advantage of this feature to authenticate sent messages with threshold signatures *internal* to the protocol. In particular, when any P_b, P_{1-b} agree on an entry (R, sk_b) to add to rpool, they also produce a threshold signature z under the shared public key pk authenticating this entry. Each P_b is instructed to send the new rpool entry accompanied by its signature z to every offline party. If at least one of P_b, P_{1-b} follows the protocol (note that only one may be corrupt), every offline party will have received the new rpool entry when it wakes up. Additionally due to the same reason that $(2, n)$ signatures are unforgeable by an adversary corrupting a single party, such an adversary will be unable to convince any offline P_i to add an entry to rpool that was not approved by an honest party. An implication of this unforgeability feature is that an offline party can safely ignore received messages that are malformed.

A Subtle Attack Again the inherent unfairness of two-party computation stands in the way of achieving a consistent rpool. In particular an adversary corrupting P_b^* may choose to abort the computation the moment she receives the internal threshold signature z, denying the online honest party P_{1-b} this value and therefore removing its ability to convince its offline friends to add the new rpool entry. This is a dangerous situation, as P_b^* now has the power to control whether the offline parties update rpool or not, i.e. by choosing whether or not to send the new rpool entry (which it can convince offline parties to use as it has z). While this will not immediately constitute a breach of privacy, the fact that honest parties do not agree on rpool could violate unanimous erasure; at best this requires *all* honest parties to come online to re-share the secret, and at worst this could mean that the secret key is lost forever (e.g. in the (2,3) cold storage use case).

Our Solution This is where it is crucial that the first half of the threshold signing protocol ($\pi^{\mathsf{R}}_{\mathsf{Sign}}$) is simulatable without the signature σ itself; in fact it is the entire reason for this choice of abstraction. Assume that P_{1-b} updates its rpool with the new value before even producing z. Following this, P_{1-b} will

refuse to instruct $\mathcal{F}_{\mathsf{Sign}}^{n,2}$ to reveal the signature σ until it is in possession of the local threshold signature z to send to offline parties. There are now two choices that P_b^* has when executing the attack described above:

- **Update rpool of offline parties**: i.e. the adversary chooses to add (R, f_δ) to the rpool of some/all offline parties. In this case, in order to actually exploit the inconsistency between rpool of different honest parties, the adversary must trigger a refresh that produces different outcomes for different rpool. Specifically, the signature σ under public key pk and the nonce R must appear on the blockchain; i.e. the same R that P_b interrupted signing with P_{1-b} but sent to offline parties. However since protocol $\pi_{\mathsf{Sign}}^{\mathsf{R}}$ by itself keeps σ completely hidden and P_{1-b} does not continue with $\pi_{\mathsf{Sign}}^{\sigma}$, the task of the adversary is essentially to produce σ under a specific uniformly chosen R (of unknown discrete logarithm). We show that this amounts to solving the discrete logarithm problem in the curve $\mathbb{G}$.
- **Do not update rpool of offline parties**: All honest parties have the same rpool anyway, and there is no point of concern.

Therefore instead of using complicated mechanisms (eg. forcing everyone to come online, extra messages on the blockchain, etc.) to ensure that every honest party agrees on the same rpool, we design our protocol so that any inconsistencies in rpool are inconsequential.

We present the protocol below, which includes some optimizations and notation omitted from the above explanation.

Protocol 3: $\pi_{\rho\text{-}\mathsf{sign}}^{(2,n)}$

Parameters: Elliptic Curve Group $(\mathbb{G}, G, q)$
Parties: P_b for $b \in [n]$
Ideal Oracles: $\mathcal{F}_{\mathsf{Com\text{-}ZK}}^{R_{\mathsf{DL}}}$, $\mathcal{G}_{\mathsf{Ledger}}$, random oracle RO
Inputs:

- **Common**: Message to be signed $m \in \{0,1\}^*$, public key $\mathsf{pk} \in \mathbb{G}$, each party's share in the exponent $\mathsf{pk}_b = \lambda_b^{1-b}(0) \cdot F(b)$ where F is the polynomial over $\mathbb{G}$ passing through $(0, \mathsf{pk})$ and $(b, f(b) \cdot G)$, epoch index $\mathsf{epoch} \in \mathbb{Z}^+$
- **Private**: Each party P_b has private input $\mathsf{sk}_b = \lambda_b^{1-b}(0) \cdot f(b) \in \mathbb{Z}_q$

---

1. **Tag R from Threshold Signature:** (*identical to* $\pi_{\rho\text{-}\mathsf{sign}}^{(2,2)}$)
2. **Sample New Polynomial:** (*identical to* $\pi_{\rho\text{-}\mathsf{sign}}^{(2,2)}$)
3. **Store Tagged Refresh:**
   i. Append $(R, \mathsf{sk}_b', \mathsf{epoch})$ to rpool
   ii. Establish common nonce $K \in \mathbb{G}$ along with an additive sharing of its discrete logarithm:
      a. Sample $k_b \leftarrow \mathbb{Z}_q$, set $K_b = k_b \cdot G$ and send $(\texttt{com-proof}, \mathsf{id}_b^{\mathsf{com\text{-}zk}}, k_b, K_b)$ to $\mathcal{F}_{\mathsf{Com\text{-}ZK}}^{R_{\mathsf{DL}}}$
      b. Upon receiving $(\texttt{committed}, 1-b, \mathsf{id}_{1-b}^{\mathsf{com\text{-}zk}})$ from $\mathcal{F}_{\mathsf{Com\text{-}ZK}}^{R_{\mathsf{DL}}}$, send $(\texttt{open}, \mathsf{id}_b^{\mathsf{com\text{-}zk}})$ to $\mathcal{F}_{\mathsf{Com\text{-}ZK}}^{R_{\mathsf{DL}}}$
      c. Wait to receive $(\texttt{decommitted}, 1-b, \mathsf{id}_{1-b}^{\mathsf{com\text{-}zk}}, K_{1-b} \in \mathbb{G})$ from $\mathcal{F}_{\mathsf{Com\text{-}ZK}}^{R_{\mathsf{DL}}}$
      d. Set $K = K_b + K_{1-b}$
   iii. Compute
   $$e = \mathsf{RO}(R||K||\delta||\mathsf{epoch})$$
   $$z_b = e \cdot \mathsf{sk}_b + k_b$$
   iv. Send z_b to P_{1-b} and wait for z_{1-b}, upon receipt verifying that
   $$z_{1-b} \cdot G = e \cdot \mathsf{pk}_{1-b} + K_{1-b}$$
   and compute $z = z_b + z_{1-b}$
   v. Set $\mathsf{msg} = (R, \mathsf{epoch}, \delta, K, z)$
   vi. For each $i \in [n] \setminus \{b, 1-b\}$, send msg to P_i
4. Complete the threshold signature protocol by running $\sigma \leftarrow \pi_{\mathsf{Sign}}^{\sigma}$
5. If $\sigma \neq \perp$ then set $\mathsf{tx} = (m, R, \sigma)$ and send $(\texttt{Submit}, \mathsf{sid}, \mathsf{tx})$ to $\mathcal{G}_{\mathsf{Ledger}}$

We now specify the refresh procedure for a party P_i to process its received messages, reconstruct rpool, and shift to the latest shared polynomial. This refresh procedure is general so that parties who were offline for a number of epochs can catch up.

Protocol 4: $\pi^{(2,n)}_{\rho\text{-update}}$

Parameters: Elliptic Curve Group $(\mathbb{G}, G, q)$
Parties: P_i (local refresh protocol)
Ideal Oracles: $\mathcal{G}_{\mathsf{Ledger}}$
Inputs: Epoch counter epoch, a list $\mathsf{rpool} = \{(\mathsf{epoch}, \mathsf{sk}'_i, R)\}$, public key pk, private key share sk_i (define $\mathsf{pk}_i = \mathsf{sk}_i \cdot G$).

1. For each unique msg received when offline do the following:
   i. Parse $(R, \mathsf{epoch}', \delta, K, z) \leftarrow \mathsf{msg}$ and if $\mathsf{epoch}' < \mathsf{epoch}$ ignore this msg
   ii. Compute $e = \mathsf{RO}(R||K||\delta||\mathsf{epoch}')$ and verify that
   $$z \cdot G = e \cdot \mathsf{pk} + K$$
   iii. Define degree-1 polynomial f_δ over $\mathbb{Z}_q$ such that
   $$f_\delta(0) = 0 \quad \text{and} \quad f_\delta(1) = \delta$$
   and interpolate $\delta_i = f_\delta(i)$
   iv. If $\mathsf{epoch}' = \mathsf{epoch}$, compute
   $$\mathsf{sk}'_i = \mathsf{sk}_i + \delta_i$$
   and append $(R, \mathsf{sk}'_i, \mathsf{epoch})$ to rpool
   v. Otherwise $\mathsf{epoch}' > \mathsf{epoch}$ so append $(\mathsf{epoch}', \delta_i, R)$ to fpool
2. Send $(\mathtt{Read})$ to $\mathcal{G}_{\mathsf{Ledger}}$ and receive $(\mathtt{Read}, b)$ in response. Set BLK to be the latest blocks occurring in b since last awake, and in sequence from the earliest block, for each (σ, R) under pk encountered do the following:
   i. Find $(R, \mathsf{sk}'_i, \mathsf{epoch}) \in \mathsf{rpool}$ (match by R), ignore σ if not found
   ii. Overwrite $\mathsf{sk}_i = \mathsf{sk}'_i$, set $\mathsf{epoch} = \mathsf{epoch} + 1$, and set $\mathsf{rpool} = \varnothing$
   iii. For each $(\mathsf{epoch}, \delta_i, R) \in \mathsf{fpool}$ (i.e. matching current epoch) do:
      (i) Set $\mathsf{sk}'_i = \mathsf{sk}_i + \delta_i$
      (ii) Append $(R, \mathsf{sk}'_i, \mathsf{epoch})$ to rpool
      (iii) Remove this entry from fpool

In the above refresh protocol $\pi^{(2,n)}_{\rho\text{-update}}$, the set rpool will always be consistent across honest parties (except for inconsequential differences) and fpool will be empty by the end. This is due to the fact that fpool contains candidate refresh values intended for epoch values further than the one "caught up with" so far; no honest party will approve a candidate with a higher epoch counter than its own, and every honest party reaches the same epoch value upon refresh. Further details can be found in the section addressing non-degeneracy of the protocol in the proof that follows.

Theorem 7.1. *If* $(\pi^{\mathsf{DKG}}_{\mathsf{Setup}}, \pi^{\mathsf{R}}_{\mathsf{Sign}}, \pi^{\sigma}_{\mathsf{Sign}})$ *is a threshold signature scheme for signing equation* Sign*, and the discrete logarithm problem is hard in* $\mathbb{G}$*, then* $(\pi^{\mathsf{DKG}}_{\mathsf{Setup}}, \pi^{(2,n)}_{\rho\text{-update}})$ *UC-realizes* $\mathcal{F}^{n,2}_{\mathsf{Sign}}$ *in the* $(\mathcal{G}_{\mathsf{Ledger}}, \mathcal{F}^{R_{\mathsf{DL}}}_{\mathsf{Com\text{-}ZK}})$*-hybrid model in the presence of a mobile adversary corrupting one party, with offline refresh.*

Proof. (Sketch) The protocol $\pi^{\mathsf{DKG}}_{\mathsf{Setup}}$ can be simulated the standard way, with the corrupt party P_i's key share sk_i remembered as output. We now describe the simulator $\mathcal{S}^{(2,n)}_{\rho\text{-sign}}$for protocol $\pi^{(2,n)}_{\rho\text{-sign}}$. This simulator is given sk_i as input, and outputs (R, sk'_i).

Simulator 1: $\mathcal{S}^{(2,n)}_{\rho\text{-sign}}$

Parameters: Elliptic Curve Group $(\mathbb{G}, G, q)$
Ideal Oracles Controlled: $\mathcal{F}^{R_{\mathsf{DL}}}_{\mathsf{Com\text{-}ZK}}$, random oracle RO
Ideal Oracles Not Controlled: $\mathcal{G}_{\mathsf{Ledger}}$
Inputs:

- **Common**: Message to be signed $m \in \{0,1\}^*$, public key $\mathsf{pk} \in \mathbb{G}$, each party's share in the exponent $F(b) = f(b) \cdot G$, epoch index $\mathsf{epoch} \in \mathbb{Z}^+$
- **Private**: P_b's key share $\mathsf{sk}_b = f(b) \in \mathbb{Z}_q$

---

1. **Tag R from Threshold Signature:**
   i. Simulate the first half of the threshold signing protocol
   $$(R, \mathsf{state}_b) \leftarrow \mathcal{S}^{\mathsf{R}}_{\mathsf{Sign}}(\mathsf{sk}_b, 1-b, m)$$
   relaying $(\texttt{get-instance-key}, \mathsf{id}^{\mathsf{sig}})$ and $(\texttt{instance-key}, \mathsf{id}^{\mathsf{sig}}, R)$ between $\mathcal{S}^{\mathsf{R}}_{\mathsf{Sign}}$ and $\mathcal{F}^{n,2}_{\mathsf{Sign}}$ when required.
2. **Sample New Polynomial:** (*identical to* $\pi^{(2,2)}_{\rho\text{-sign}}$)
   i. Sample $\delta \leftarrow \mathbb{Z}_q$ and send $(\mathsf{id}^{\mathsf{coin}}, \delta)$ to P_b on behalf of $\mathcal{F}_{\mathsf{Coin}}$
   ii. Define degree-1 polynomial f_δ over $\mathbb{Z}_q$ such that
   $$f_\delta(0) = 0 \quad \text{and} \quad f_\delta(1) = \delta$$
   iii. Compute
   $$\mathsf{sk}'_b = \mathsf{sk}_b + f_\delta(b)$$
3. **Store Tagged Refresh:**
   i. Simulate a signature $R, \delta, \mathsf{epoch}$ under pk_{1-b}:
      a. Sample $z_{1-b} \leftarrow \mathbb{Z}_q$ and $e \leftarrow \mathbb{Z}_q$ uniformly at random
      b. Compute
      $$K = z \cdot G - e \cdot \mathsf{pk}_{1-b}$$
      c. Program $\mathsf{RO}(R||K||\delta||\mathsf{epoch}) = e$
   ii. Establish common nonce $K \in \mathbb{G}$:
      a. Send $(\texttt{committed}, 1-b, \mathsf{id}^{\mathsf{com\text{-}zk}}_{1-b})$ to P^*_b on behalf of $\mathcal{F}^{R_{\mathsf{DL}}}_{\mathsf{Com\text{-}ZK}}$
      b. Receive $(\texttt{com-proof}, \mathsf{id}^{\mathsf{com\text{-}zk}}_b, k_b, K_b)$ on behalf of $\mathcal{F}^{R_{\mathsf{DL}}}_{\mathsf{Com\text{-}ZK}}$
      c. Set $K_{1-b} = K - K_b$
      d. Send $(\texttt{decommitted}, 1-b, \mathsf{id}^{\mathsf{com\text{-}zk}}_{1-b}, K_{1-b} \in \mathbb{G})$ to P^*_b on behalf of $\mathcal{F}^{R_{\mathsf{DL}}}_{\mathsf{Com\text{-}ZK}}$
      e. Wait for $(\texttt{open}, \mathsf{id}^{\mathsf{com\text{-}zk}})$ from P_b, upon receipt sending z_{1-b} in response
   iii. Wait for z_b, upon receipt verifying that
   $$z_b = e \cdot \mathsf{sk}_b + k_b$$
4. Simulate the rest of the threshold signature protocol by running $\mathcal{S}^{\sigma}_{\mathsf{Sign}}(\mathsf{state}_b)$ relaying $(\texttt{proceed}, \mathsf{id}^{\mathsf{sig}})$ and $(\texttt{signature}, \mathsf{id}^{\mathsf{sig}}, \sigma)$ between P^*_b and $\mathcal{F}^{n,2}_{\mathsf{Sign}}$ as necessary.
5. If P^*_b asks $\mathcal{F}^{n,2}_{\mathsf{Sign}}$ to release σ to P_{1-b}, then set $\mathsf{tx} = (m, R, \sigma)$ and send $(\texttt{Submit}, \mathsf{sid}, \mathsf{tx})$ to $\mathcal{G}_{\mathsf{Ledger}}$
6. Output (R, sk'_b)

Simulating $\pi^{(2,n)}_{\rho\text{-update}}$is simple: every time the adversary $\mathcal{Z}$ sends a $(\texttt{sign}, m, i, j)$ command to a pair of honest parties, the simulator obtains a signature R, σ from $\mathcal{F}^{n,2}_{\mathsf{Sign}}$, samples $\delta \leftarrow \mathbb{Z}_q$, and simulates a

local signature z under pk to authenticate $R, \delta, \mathsf{epoch}$ just as in Step i. of Simulator $\mathcal{S}^{(2,n)}_{\rho\text{-}\mathsf{sign}}$above. It sets $\mathsf{msg} = (R, \mathsf{epoch}, \delta, K, z)$ and makes msg available to the corrupt party.

We now sketch an argument that the distribution of the real protocol is computationally indistinguishable from the ideal one.

We can progressively substitute each instance of $\pi^{(2,n)}_{\rho\text{-}\mathsf{sign}}$run with honest parties belonging to an epoch with $\mathcal{S}^{(2,n)}_{\rho\text{-}\mathsf{sign}}$run with $\mathcal{F}^{n,2}_{\mathsf{Sign}}$. The distinguishing advantage of $\mathcal{Z}$ at each step is bounded by the advantage of a PPT adversary distinguishing $(\pi^{\mathsf{DKG}}_{\mathsf{Setup}}, \pi^{\mathsf{R}}_{\mathsf{Sign}}, \pi^{\sigma}_{\mathsf{Sign}})$ from the corresponding ideal executions with $\mathcal{F}^{n,2}_{\mathsf{Sign}}$ as produced by simulators $(\mathcal{S}^{\mathsf{DKG}}_{\mathsf{Setup}}, \mathcal{S}^{\mathsf{R}}_{\mathsf{Sign}}, \mathcal{S}^{\sigma}_{\mathsf{Sign}})$, which is assumed to be negligible. In order to extend this strategy to a mobile adversary, it suffices to argue that the polynomials f, f' used to share sk appear independently distributed before and after a refresh. This follows immediately from the fact that an adversary who jumps from party P_i to P_j is given $f(i)$ and $f'(j)$ but does not see the difference f_δ between f, f', just as discussed in the (2,2) case in Section 6.

It remains to be argued that the protocol is not degenerate. The non-degeneracy property is achieved by fulfilling two important requirements:

System Epoch Increments When the parties executing $\pi^{(2,n)}_{\rho\text{-}\mathsf{sign}}$are honest, the system epoch will always increment upon the next refresh command, i.e. if $\pi^{(2,n)}_{\rho\text{-}\mathsf{sign}}$is run by honest parties with counter epoch, then every subsequent execution of $\pi^{(2,n)}_{\rho\text{-}\mathsf{update}}$by any party in the system will result in a local epoch counter of at least $\mathsf{epoch} + 1$. This is easy to see for this protocol, as honest parties executing $\pi^{(2,n)}_{\rho\text{-}\mathsf{sign}}$will always produce a signature σ which will subsequently appear on the blockchain (after delay T as per $\mathcal{G}_{\mathsf{Ledger}}$). Simultaneously every party will find a corresponding update to rpool sent to it, which will be applied by $\pi^{(2,n)}_{\rho\text{-}\mathsf{update}}$when σ appears on the blockchain.

Consistency Every honest party outputs the same epoch counter upon executing $\pi^{(2,n)}_{\rho\text{-}\mathsf{update}}$simultaneously. As alluded to earlier in Section 7 proving this amounts to showing that the state of rpool maintained by each honest party differs inconsequentially. In particular, let P_i and P_j be honest parties maintaining rpool_i and rpool_j respectively such that $\exists (R, \mathsf{sk}'_i, \mathsf{epoch}) \in \mathsf{rpool}_i$ but $\nexists (R, \mathsf{sk}'_j, \mathsf{epoch}) \in \mathsf{rpool}_j$. First we claim that $(R, \mathsf{sk}'_i, \mathsf{epoch})$ can be traced to a unique execution of $\pi^{(2,n)}_{\rho\text{-}\mathsf{sign}}$between a corrupt party P^*_b and honest party P_{1-b}. There are only two alternative events: (1) that there is a collision in R values generated by two protocol instances (occurs with probability $|\vec{m}|^2/2q$ where $|\vec{m}|$ is the number of messages signed), or (2) P_i received z authenticating this entry without any honest party's help in its creation; the exact same technique to prove (threshold) Schnorr signatures secure can be employed here to construct a reduction to the Discrete Logarithm problem in curve $\mathbb{G}$ (if this event occurs with probability ϵ then there is a reduction to DLog successful with probability $\epsilon/|\vec{m}|$). Given that $(R, \mathsf{sk}'_i, \mathsf{epoch})$ can be traced to a unique execution of $\pi^{(2,n)}_{\rho\text{-}\mathsf{sign}}$between P^*_b and P_{1-b} it must be the case that P^*_b aborted the comptation at Step iv., i.e. P^*_b received z to authenticate this entry but withheld this value from P_{1-b} (or else P_j would have received this entry when offline as well due to P_{1-b}). Observe that this inconsistency in $\mathsf{rpool}_i, \mathsf{rpool}_j$ is consequential only if (σ, R) appears on $\mathcal{G}_{\mathsf{Ledger}}$, despite the fact that P_{1-b} will not execute $\pi^{\sigma}_{\mathsf{Sign}}$to produce this value. We show that if this event happens with probability ϵ then there is an adversary for the DLog problem successful with probability $\epsilon/|\vec{m}|$. This is because R is chosen uniformly in $\pi^{(2,n)}_{\rho\text{-}\mathsf{sign}}$(ie. internally by $\pi^{\mathsf{R}}_{\mathsf{Sign}}$ as it realizes $\mathcal{F}^{n,2}_{\mathsf{Sign}}$) and the task of $\mathcal{Z}$ is to produce σ that verifies under uniformly chosen nonce R and public key pk. We can use such a $\mathcal{Z}$ to solve the DLog problem in $\mathbb{G}$ as follows:

1. Receive $X \in \mathbb{G}$ from the DLog challenger.
2. Choose $\mathsf{sk} \leftarrow \mathbb{Z}_q$, set $\mathsf{pk} = \mathsf{sk} \cdot G$
3. Run $\mathcal{S}^{\mathsf{DKG}}_{\mathsf{Setup}}$ for $\mathcal{Z}$ with pk programmed to be the public key.
4. For each message $m \in \vec{m}$ except one, run $\mathcal{S}^{(2,n)}_{\rho\text{-}\mathsf{sign}}$ as required to simulate $\pi^{(2,n)}_{\rho\text{-}\mathsf{sign}}$while also acting on behalf of $\mathcal{F}^{n,2}_{\mathsf{Sign}}$
5. For one randomly chosen instance of $\pi^{(2,n)}_{\rho\text{-}\mathsf{sign}}$, use $\mathcal{S}^{\mathsf{R}}_{\mathsf{Sign}}$ to program X as the signing nonce R.

6. If the correct instance of $\pi^{(2,n)}_{\rho\text{-sign}}$is chosen, P^*_b will abort this protocol before the corresponding σ has to be released, and yet σ still appears on $\mathcal{G}_{\mathsf{Ledger}}$

7. If σ is obtained from $\mathcal{G}_{\mathsf{Ledger}}$, solve for x such that $x \cdot G = X$ as a function of σ, sk as per the signing equation Sign. This is dependent on the equation Sign itself, but it is straightforward how to retrieve the instance key x given the secret key sk and signature σ as per $\mathsf{Sign}_{\text{ECDSA}}$ and $\mathsf{Sign}_{\text{ECDSA}}$.

The above reduction succeeds when $\mathcal{Z}$ induces this event (probability ϵ) and the correct instance of $\pi^{(2,n)}_{\rho\text{-sign}}$is chosen (probability $1/|\vec{m}|$) bringing the total success probability to $\epsilon/|\vec{m}|$.

As the simulated distribution is indistinguishable from the execution of the real protocol and the protocol is non-degenerate, this proves the theorem.

■

An Optimization We note that one can save a query to $\mathcal{F}_{\mathsf{Coin}}$ and a $\mathbb{Z}_q$ element from being having to be sent by defining $\delta = \mathsf{RO}(R||K||\mathsf{epoch})$ instead of computing it separately from the internal threshold signature z. As (R, K) guarantee κ bits of entropy, the resulting δ will be distributed uniformly.

8 Proactive $(2, n)$ ECDSA

Computing $(2, n)$ ECDSA signatures is significantly more difficult than Schnorr, due to the non-linear nature of the ECDSA signing equation. As a result, all such recent threshold ECDSA protocols [GG18, LNR18, DKLs18, DKLs19] make use of a secure multiplication functionality (or equivalent protocol) $\mathcal{F}_{\mathsf{MUL}}$ in their signing phases. If $\mathcal{F}_{\mathsf{MUL}}$ were to be instantiated independently for each threshold ECDSA signature produced, we could just use the same strategy as in the previous section, since the $\pi^{\mathsf{R}}_{\mathsf{Sign}}$ protocol would take only key shares as arguments. However $\mathcal{F}_{\mathsf{MUL}}$ is expensive to realize for individual invocations, and given that threshold signature protocols already need a "preprocessing" phase for key generation (ie. $\pi^{\mathsf{DKG}}_{\mathsf{Setup}}$), all the cited works make use of this phase to also run some preprocessing for $\mathcal{F}_{\mathsf{MUL}}$ to make its invocation during signing cheaper. Therefore, we also need to change how we deal with proactively refreshing the shares. In a nutshell, the main technical challenge we address in this section is that now the parties, on top of their key shares, also include in their persistent storage some state information for the $\mathcal{F}_{\mathsf{MUL}}$ protocol and that this state is a new target for a mobile adversary. Therefore, the state needs to be refreshed as well.

We start by abstracting the two-party multiplication protocol ($\pi^{\mathsf{Setup}}_{\mathsf{MUL}}$,$\pi^{\mathsf{Online}}_{\mathsf{MUL}}$) used within ECDSA threshold protocols. The protocols are run by party P_i with P_j as the counterparty as follows,

- $(\mathsf{state}^{i,j}_{\mathsf{MUL}} \in \{0,1\}^*) \leftarrow \pi^{\mathsf{Setup}}_{\mathsf{MUL}}(j)$
- $(t_i \in \mathbb{Z}_q) \leftarrow \pi^{\mathsf{Online}}_{\mathsf{MUL}}\left(\mathsf{state}^{i,j}_{\mathsf{MUL}}, x_j\right)$

The pair of protocols ($\pi^{\mathsf{Setup}}_{\mathsf{MUL}}$,$\pi^{\mathsf{Online}}_{\mathsf{MUL}}$) must realize $\mathcal{F}_{\mathsf{MUL}}$. As per the functionality specification, $t_i + t_j = x_i \cdot x_j$ after $\pi^{\mathsf{Online}}_{\mathsf{MUL}}$ is run, and this can be done arbitrarily many times for different inputs. Every pair of parties in the system shares an instantiation of $\mathcal{F}_{\mathsf{MUL}}$, and so P_i maintains $\mathsf{state}^{i,j}_{\mathsf{MUL}}$ for each $j \in [n] \setminus i$. Therefore in our abstraction for threshold ECDSA protocols ($\pi^{\mathsf{DKG}}_{\mathsf{Setup}}, \pi^{\mathsf{Setup}}_{\mathsf{MUL}}, \pi^{\mathsf{R}}_{\text{ECDSA}}, \pi^{\sigma}_{\text{ECDSA}}$) we include the state required by P_i for multiplication with P_j as an argument for online signing. We avoid rewriting the formal abstraction for readability, as it is essentially a reproduction of Section 5 with the inclusion of $\mathsf{state}^{i,j}_{\mathsf{MUL}}$ as an argument/output in the correct places.

The same restrictions on the simulators for these protocols hold, see Section 5 for details. It is not hard to show that the recent protocols of Lindell et al. [LNR18], Gennaro and Goldfeder [GG18], and Doerner et al. [DKLs19] fit these characterizations. The inclusion of $\{\mathsf{state}^{i,j}_{\mathsf{MUL}}\}_{j\in[n]}$ as persistent state that parties must maintain across signatures creates an additional target that must be defended from a mobile adversary. We show how here to refresh $\{\mathsf{state}^{i,j}_{\mathsf{MUL}}\}_{j\in[n]}$ required by the OT-based instantiation of $\mathcal{F}_{\mathsf{MUL}}$ (as in Doerner et al. [DKLs19]) and consequently upgrade compatible threshold ECDSA protocols [DKLs19, GG18, LNR18] to proactive security.

Approach The setup used by the multiplier of Doerner et al. consists of a number of base OTs which are "extended" for use online [KOS15]. These base OTs are the only component of their multiplier which requires each party to keep private state. Therefore re-randomizing these OTs in the interval between an adversary's jump from one party to the other is sufficient to maintain security. The central idea to implement this re-randomization is to apply the approach introduced by Beaver [Bea95] of "adjusting" preprocessed OTs once inputs are known online.

8.1 Proactive Secure Multiplication

We begin by describing how two parties can re-randomize OT itself, and then describe how to apply this technique to re-randomize OT Extensions.

Re-randomizing Oblivious Transfer Assume that Alice has two uniform κ-bit strings r_0, r_1, and Bob has a bit b and correspondingly the string r_b. Let $rand \leftarrow \{0,1\}^{2\kappa+1}$ be a uniformly chosen string that is parsed into chunks $r'_0, r'_1 \in \{0,1\}^\kappa$ and $b' \in \{0,1\}$ by both parties. The re-randomization process for Alice ($\mathsf{Refresh\_OT_A}$) and Bob ($\mathsf{Refresh\_OT_B}$) is non-interactive (given $rand$) and proceeds as follows:

1. $\mathsf{Refresh\_OT_A}\left((r_0, r_1), rand\right)$: output $r''_0 = r_{b'} \oplus r'_0$ and $r''_1 = r_{1-b'} \oplus r'_1$
2. $\mathsf{Refresh\_OT_B}\left((b, r_b), rand\right)$: output $b'' = b \oplus b'$ and $r''_{b''} = r_b \oplus r'_{b''}$
3. Alice now holds (r''_0, r''_1) and Bob holds $b'', r''_{b''}$

It is clear to see that Alice and Bob learn nothing of each other's private values, only the offsets r'_0, r'_1, b' between the new and old ones. Consider the view of a mobile adversary that jumps from one party to the other.

- Alice $\rightarrow$ Bob: (r_0, r_1) before the refresh, and $(b'', r''_{b''})$ after the refresh.
- Bob $\rightarrow$ Alice: (b, r_b) before the refresh, and (r''_0, r''_1) after the refresh.

Assuming that r'_0, r'_1, b' are hidden and that these values are uniformly chosen, in both the above cases the adversary's view before and after the refresh are completely independent.

Re-randomizing OT Extensions The persistent state maintained by OT Extension protocols based on that of Ishai et al. [IKNP03] consists of the result of a number of OTs performed during a preprocessing phase. Re-randomizing this state can be done by simply repeating the above protocol for each preprocessed OT instance. Indeed, the instantiation of OT Extension implemented by Doerner et al. is the protocol of Keller et al. [KOS15] which is captured by this framework.

Re-randomizing multipliers There is no further persistent state maintained across $\mathcal{F}_{\mathsf{MUL}}$ invocations by the protocol of Doerner et al. [DKLs19], and so we leave implicit the construction of $\mathsf{state_{MUL}}' \leftarrow \mathsf{Refresh\_MUL}(\mathsf{state_{MUL}}, rand)$. The only missing piece is how $rand$ is chosen; in the context of the multipliers in isolation, this value can be thought of coming from a coin-tossing protocol that is invisible to the adversary (when neither party is corrupt).

8.2 Multiplier Refresh in $(2, n)$ ECDSA

The previous subsection describes how to realize $\mathcal{F}_{\mathsf{MUL}}$ with proactive security when a mechanism to agree on when/which $rand$ to use is available. Fortunately the protocol described in Section 7 provides exactly such a mechanism for the $(2, n)$ threshold signature setting. We briefly describe how to augment Protocol 7 to produce the randomness $rand$ required to proactivize multipliers in addition to the distributed key shares.

$(2, n)$ **Offline Refresh** The two online parties P_b, P_{1-b} engage in a coin-tossing protocol in the **Sample New Polynomial** phase to produce a uniform κ-bit value seed. In the **Store Tagged Refresh** phase they include seed to be stored in rpool along with corresponding $\mathsf{epoch}, \mathsf{sk}'_b, R$ (and communicate seed to offline parties along with these values). If the signature using R is used to signal a refresh, then seed is expanded by every pair of parties to produce *rand* as necessary.

We give the entire protocol in Appendix E for completeness.

9 Performance and Implementation

We discuss here the concrete overhead our refresh protocol adds to existing state of the art threshold ECDSA schemes, as most cryptocurrencies today (Bitcoin, Ethereum, etc.) use ECDSA as their canonical signature scheme. As at this point we are discussing specific protocols, we make the following observation: In the protocols of Lindell et al. [LNR18], Doerner et al. [DKLs19], and Gennaro and Goldfeder [GG18] the extra messages added by $\pi^{(2,n)}_{\rho\text{-}\mathsf{sign}}$can be sent in parallel with the main ECDSA protocols. In particular, each $\pi^{\mathsf{R}}_{\text{ECDSA}}$ has at least two rounds which can be used to generate K and δ in parallel, and each $\pi^{\sigma}_{\text{ECDSA}}$has at least one round before σ is released during which z can be constructed and verified.

9.1 Cost Analysis

In Table 1 we recall the costs of the ($\pi^{\mathsf{R}}_{\text{ECDSA}}$,$\pi^{\sigma}_{\text{ECDSA}}$) combined protocols of Doerner et al. [DKLs19] and Lindell et al. [LNR18] (OT-based) for perspective, and then give the overhead induced by $\pi^{(2,n)}_{\rho\text{-}\mathsf{sign}}$.

Protocol	Rounds	EC Mult.s	Comm.
Lindell et al. [LNR18]	8	239	195 KiB
Doerner et al. [DKLs19]	7	6	118 KiB
$\pi^{(2,n)}_{\rho\text{-}\mathsf{sign}}$**overhead**	0	6	192 Bytes

Table 1: Overhead of applying $\pi^{(2,n)}_{\rho\text{-}\mathsf{sign}}$to proactivize $(2, n)$ ECDSA protocols instantiated with 256-bit curves. Figures are per-party and do not include cost of implementing proactive channels to communicate 160 bytes to each offline party every refresh.

Finally the update procedure $\pi^{(2,n)}_{\rho\text{-}\mathsf{update}}$first requires reading the blockchain and scanning for signatures under the common public key since last awake– essentially the same operation as required to update balance of funds available in a wallet. Additionally one has to read messages received when offline and perform two curve multiplications for each refresh missed.

9.2 Implementation

In order to demonstrate the compatibility and efficiency of our refresh procedure, we implemented it to augment two different recent threshold ECDSA protocols; specifically those of Doerner et al. [DKLs19] and Gennaro and Goldfeder [GG18]. We present the results in this section.

We ran both sets of experiments on Amazon's AWS EC2 using a pair of t3.small machines located in the same datacenter for uniformity. However as the implementations of the base threshold ECDSA protocols came from different codebases, we stress that the important metric is the overhead added by our protocol in each case, and that comparison of the concrete times across the ECDSA protocols is not necessarily meaningful.

9.2.1 Proactivizing Doerner et al. [DKLs19]

As Doerner et al. natively utilize OT based multipliers, augmenting their threshold ECDSA signing with our refresh procedure yields a *fully* proactivized ECDSA wallet. We ran three experiments, during which we measured wall-clock time, including latency costs, collecting 100,000 samples and averaging them. We first

ran their signing protocol unmodified, which took an average of 5.303ms to produce a signature. We then ran the same protocol augmented with our refresh generation procedure (i.e. $\pi^{(2,n)}_{\rho\text{-sign}}$) and found it to take an average of 6.587ms, i.e. a 24.2% increase. Finally we measured the cost of applying an update upon waking up (i.e. $\pi^{(2,n)}_{\rho\text{-update}}$) to be 0.381ms. Note that this figure does not account for the costs of the proactive channels or $\mathcal{G}_{\mathsf{Ledger}}$ (which is done anyway to update one's balance); the point of this benchmark is to demonstrate the efficiency of applying updates in isolation.

9.2.2 Gennaro and Goldfeder [GG18]

In order to understand the overhead added by the refresh procedure to the communication pattern of a different $(2, n)$ ECDSA based wallet, we implemented the protocol of Gennaro and Goldfeder [GG18] and augmented it with our refresh procedure during signing. Note their protocol makes use of a Paillier-based multiplier which we do not proactivize (see Canetti et al. [CGG+20] for how this can be done), and the cost of proactivizing an OT-based multiplier is negligible (0.381ms as shown previously). This is representative of the $(2, 3)$ cold storage application where the multipliers need not be offline-refreshed. We refer to the original ($\pi^{\mathsf{R}}_{\text{ECDSA}}$,$\pi^{\sigma}_{\text{ECDSA}}$) as GG and the augmented $\pi^{(2,n)}_{\rho\text{-sign}}$as GG'.

We did not implement forward secure channels, we instead simulated it with reads from disk. We collected twenty samples for each configuration and found the average execution time of GG to be 1.433s and that of GG' to be 1.635s. In particular, $\pi^{(2,n)}_{\rho\text{-sign}}$incurs a 14.09% overhead in computation. Note that this figure does not include network latency, but in the LAN setting the measurements were within margin of error.

The code can be found in `https://gitlab.com/neucrypt/mpecdsa/` (full proactivization of [DKLs19] by Jack Doerner) and `https://github.com/KZen-networks/multi-party-ecdsa/tree/gg_pss` (proactivization in KZen library).

10 General (t, n) Impossibility

We showed in Section 4.1 that an honest majority protocol is easy to construct, and so we assume for the rest of the discussion that we are in a setting where there is no online honest majority.

Many proactive secret sharing protocols in the literature have fundamentally followed the same approach: the refresh protocol runs roughly the same protocol that was used to share the secret, with new randomness incorporated to create an independent sharing of the same value. Therefore the ability to run verifiable secret sharing (VSS) in a given setting has always translated well to construct a refresh protocol for the same setting. Non-interactive VSS where only t online parties speak, with resiliency to $t - 1$ corruptions are known in the literature [GMW91, Sta96] suggesting that their translation to our setting would yield an offline refresh protocol.

Unfortunately this intuition turns out to be false. Recall that a central principle in offline refresh is that all (honest) parties must be in agreement about whether or not to progress to the next epoch, i.e. 'unanimous erasure'. We discussed in Section 3 why anything less than this is undesirable, as even a simple network failure could induce permanent loss of the shared secret. However even this notion turns out to require the power of an honest majority to realize (barring the $(2, n)$ case) and we give intuition as to why below.

Recall that the refresh protocol π_ρ is run by t_ρ online parties, of whom $t - 1$ may be corrupt, and we define $h = t_\rho - t + 1$ to denote how many are honest. Assume the weakest form of dishonest majority, i.e. one more corrupt party than honest, so $h = t - 2$. The communication pattern of a single refresh phase is as follows: the online parties run π_ρ, following which each online party sends a message to each of the offline parties, who upon waking up will be able to catch up to the same epoch. The unanimous erasure property requires that all honest parties stay in agreement about the epoch; i.e. no one party is falsely convinced to prematurely erase their old state. Informally, we call a message or set of messages 'convincing' if they induce an offline party to progress to the next epoch and erase their old state.

Relating Unanimous Erasure to π_ρ It is instructive to view π_ρ as an MPC protocol to produce a convincing message for offline parties to progress. As we mandate unanimous erasure, it must never be the

case that π_ρ permits an adversary to produce a convincing message while depriving online honest parties of it. In particular if π_ρ produces a convincing message then it must be visible and verifiable within the online honest parties' joint view (i.e. any subset of size h). Otherwise an adversary could at its discretion choose to induce an offline party to prematurely erase its state, and honest parties would not be able to tell either way. This property strongly suggests that π_ρ must achieve a form of *fairness* which does not bode well given that it must tolerate a dishonest majority.

A General Attack Now we hone in on exactly how an adversary can exploit the above facts. Assume that P_{off} is an offline party. Observe that the adversary is allowed to corrupt $h+1$ parties given the dishonest majority setting, and so it has the budget to keep h online parties corrupt as well as corrupt P_{off} initially, say in epoch 0. The adversary un-corrupts P_{off} and π_ρ is run successfully to move the system to epoch 1, keeping h parties corrupt (but behaving honestly) through the process. Now recall that the convincing message to P_{off} will be visible to any h online parties. Since the adversary has both: the state of P_{off} from epoch 0, as well as a 'convincing message' addressed to P_{off} by virtue of corrupting h parties during π_ρ, it is able to derive P_{off}'s refreshed state for epoch 1 despite not corrupting P_{off} in that epoch. Now simply corrupting one additional party in epoch 1 completely reveals the secret, as $h + 2 = t$ parties' private states are available to the adversary for that epoch.

Translating this intuition to a formal proof, or even a well-formed theorem, sees a number of subtle issues arise. For instance, we can not unconditionally prove that it is impossible to realize $\mathcal{F}^{n,t}_{\mathsf{ECDSA}}$ with offline refresh for $t > 2$; doing so would require proving that ECDSA itself is a signature scheme.[2]. To see why, consider a 'signature scheme' where the verification algorithm Vrfy outputs 1 on *all* inputs. Clearly, realizing a threshold version of this 'signature scheme', even with proactive security, is trivial; all parties simply output "0" when instructed to sign a message, then there is no private state to refresh. Therefore we formulate our theorem more carefully: we prove that if it possible to offline-refresh a given threshold signature scheme ($t > 2$) with a dishonest online majority, then the given signature scheme itself is succeptible to forgery.

We state our theorem in the $(\mathcal{G}_{\mathsf{Ledger}}, \mathcal{F}_{\mathsf{RO}})$ model, for the following reasons:

- $\mathcal{G}_{\mathsf{Ledger}}$ represents that this barrier can not be circumvented even with a consensus primitive as strong as an ideal ledger.
- $\mathcal{F}_{\mathsf{RO}}$ gives the power to compute any efficiently computable function [CLOS02] and so represents the ability to produce arbitrary correlated randomness during the preprocessing phase (i.e. during key generation) and also compute any function securely (albeit without robustness [Cle86]) during the refresh protocol itself.

Additionally both ideal oracles are trivial to implement when running the environment in a reduction.

Theorem 10.1. *Let* $\mathsf{Sig} = (\mathsf{KeyGen}, \mathsf{Sign}, \mathsf{Vrfy})$ *be a triple of algorithms that satisfies the completeness definition of signature schemes. If there exists a protocol* $\pi^{(t,n)}_{\rho\text{-}\mathsf{sign}}$ *in the* $(\mathcal{G}_{\mathsf{Ledger}}, \mathcal{F}_{\mathsf{RO}})$*-hybrid model that UC-realizes* $\mathcal{F}^{n,t}_{\mathsf{Sign}}$ *with* $n > t_\rho \geq t > 2$ *in the presence of a mobile adversary actively corrupting* $t-1$ *parties where* $t_\rho < 2(t-1)$*, then there exists a forger for* Sig *that succeeds with overwhelming probability.*

We delegate the proof to Appendix A. This closes the question of the gap between an honest and a dishonest majority of corruptions in the setting of offline refresh.

11 Conclusion

With the increasing adoption of threshold wallets comes the need to defend them against mobile attackers. In this work we define an "offline refresh" model for proactivizing threshold wallets with an optimal communication pattern, and study this fine-grained notion of message complexity in the proactive setting.

[2]At the moment ECDSA is known to be a signature only in the generic group model [Bro05], and not even in the random oracle model.

We show feasibility of honest majority offline refresh, and give a comprehensive treatment of the dishonest majority setting: for the $(2, n)$ setting we devise a novel efficient protocol to proactivize many standard signature schemes with offline refresh, and implement it to show that it adds little overhead in practice. Finally we show that it is impossible to have the refresh protocol tolerate a dishonest majority of participants, without having all parties come online at least at some point in each epoch. We develop new techniques to prove this theorem, and believe that they will find application in reasoning about proactive security in other contexts. However there may be relaxations of the model, physical hardware assumptions, or nonstandard trust models that are still reasonable in practice; we leave open the problem of identifying such models and tailoring constructions for them.

12 Acknowledgements

The authors would like to thank Jack Doerner for augmenting the Threshold ECDSA implementation from Doerner et al. [DKLs19] with our refresh procedure, and providing us with the benchmarks for that protocol reported in this paper. They would also like to thank the anonymous reviewers for their feedback, which was useful in improving the paper.

References

[ACD19] Joël Alwen, Sandro Coretti, and Yevgeniy Dodis. The double ratchet: Security notions, proofs, and modularization for the signal protocol. In *EUROCRYPT 2019*, pages 129–158, 2019.

[ADN06] Jesús F. Almansa, Ivan Damgård, and Jesper Buus Nielsen. Simplified threshold rsa with adaptive and proactive security. In *EUROCRYPT '06*, pages 593–611, 2006.

[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, Sep 2012.

[Bea95] Donald Beaver. Precomputing oblivious transfer. In *CRYPTO '95*, pages 97–109, 1995.

[BGG+20] Fabrice Benhamouda, Craig Gentry, Sergey Gorbunov, Shai Halevi, Hugo Krawczyk, Chengyu Lin, Tal Rabin, and Leonid Reyzin. Can a public blockchain keep a secret? In Rafael Pass and Krzysztof Pietrzak, editors, *Theory of Cryptography - 18th International Conference, TCC 2020, Durham, NC, USA, November 16-19, 2020, Proceedings, Part I*, volume 12550 of *Lecture Notes in Computer Science*, pages 260–290. Springer, 2020.

[BLS04] Dan Boneh, Ben Lynn, and Hovav Shacham. Short signatures from the weil pairing. *J. Cryptology*, 17(4):297–319, 2004.

[BMTZ17] Christian Badertscher, Ueli Maurer, Daniel Tschudi, and Vassilis Zikas. Bitcoin as a transaction ledger: A composable treatment. In *CRYPTO 2017*, pages 324–356, 2017.

[Bro05] Daniel R. L. Brown. Generic groups, collision resistance, and ECDSA. *Des. Codes Cryptography*, 2005.

[Can01] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *FOCS*, 2001.

[CCL+19] Guilhem Castagnos, Dario Catalano, Fabien Laguillaumie, Federico Savasta, and Ida Tucker. Two-party ECDSA from hash proof systems and efficient instantiations. In *Advances in Cryptology - CRYPTO 2019 - 39th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 18-22, 2019, Proceedings, Part III*, pages 191–221, 2019.

[CCL+20] Guilhem Castagnos, Dario Catalano, Fabien Laguillaumie, Federico Savasta, and Ida Tucker. Bandwidth-efficient threshold EC-DSA. In Aggelos Kiayias, Markulf Kohlweiss, Petros Wallden, and Vassilis Zikas, editors, *Public-Key Cryptography - PKC 2020 - 23rd IACR International Conference on Practice and Theory of Public-Key Cryptography, Edinburgh, UK, May 4-7, 2020, Proceedings, Part II*, volume 12111 of *Lecture Notes in Computer Science*, pages 266–296. Springer, 2020.

[CGG+20] Ran Canetti, Rosario Gennaro, Steven Goldfeder, Nikolaos Makriyannis, and Udi Peled. UC non-interactive, proactive, threshold ECDSA with identifiable aborts. In Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna, editors, *CCS '20: 2020 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, USA, November 9-13, 2020*, pages 1769–1787. ACM, 2020.

[CHH00] Ran Canetti, Shai Halevi, and Amir Herzberg. Maintaining authenticated communication in the presence of break-ins. *J. Cryptology*, 13(1):61–105, 2000.

[CKLS02] Christian Cachin, Klaus Kursawe, Anna Lysyanskaya, and Reto Strobl. Asynchronous verifiable secret sharing and proactive cryptosystems. In *Proceedings of the 9th ACM Conference on Computer and Communications Security, CCS 2002, Washington, DC, USA, November 18-22, 2002*, pages 88–97, 2002.

[Cle86] R Cleve. Limits on the security of coin flips when half the processors are faulty. In *STOC '86*, 1986.

[CLOS02] Ran Canetti, Yehuda Lindell, Rafail Ostrovsky, and Amit Sahai. Universally composable two-party and multi-party secure computation. In John H. Reif, editor, *Proceedings on 34th Annual ACM Symposium on Theory of Computing, May 19-21, 2002, Montréal, Québec, Canada*, pages 494–503. ACM, 2002.

[CM19] Jing Chen and Silvio Micali. Algorand: A secure and efficient distributed ledger. *Theor. Comput. Sci.*, 777:155–183, 2019.

[Des87] Yvo Desmedt. Society and group oriented cryptography: A new concept. In *CRYPTO*, 1987.

[DKLs18] Jack Doerner, Yashvanth Kondi, Eysa Lee, and abhi shelat. Secure two-party threshold ecdsa from ecdsa assumptions. In *IEEE S&P*, 2018.

[DKLs19] Jack Doerner, Yashvanth Kondi, Eysa Lee, and abhi shelat. Threshold ecdsa from ecdsa assumptions: The multiparty case. In *IEEE S&P*, 2019.

[DLS88] Cynthia Dwork, Nancy A. Lynch, and Larry J. Stockmeyer. Consensus in the presence of partial synchrony. *J. ACM*, 35(2):288–323, 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 Liqun Chen, Ninghui Li, Kaitai Liang, and Steve A. Schneider, editors, *Computer Security - ESORICS 2020 - 25th European Symposium on Research in Computer Security, ESORICS 2020, Guildford, UK, September 14-18, 2020, Proceedings, Part II*, volume 12309 of *Lecture Notes in Computer Science*, pages 654–673. Springer, 2020.

[DPSZ12] Ivan Damgård, Valerio Pastro, Nigel P. Smart, and Sarah Zakarias. Multiparty computation from somewhat homomorphic encryption. In Reihaneh Safavi-Naini and Ran Canetti, editors, *Advances in Cryptology - CRYPTO 2012 - 32nd Annual Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2012. Proceedings*, volume 7417 of *Lecture Notes in Computer Science*, pages 643–662. Springer, 2012.

[EOPY18] Karim Eldefrawy, Rafail Ostrovsky, Sunoo Park, and Moti Yung. Proactive secure multiparty computation with a dishonest majority. In Dario Catalano and Roberto De Prisco, editors, *Security and Cryptography for Networks - 11th International Conference, SCN 2018, Amalfi, Italy, September 5-7, 2018, Proceedings*, volume 11035 of *Lecture Notes in Computer Science*, pages 200–215. Springer, 2018.

[Fel87] Paul Feldman. A practical scheme for non-interactive verifiable secret sharing. In *28th Annual Symposium on Foundations of Computer Science, Los Angeles, California, USA, 27-29 October 1987*, pages 427–437. IEEE Computer Society, 1987.

[FGH+02] Matthias Fitzi, Daniel Gottesman, Martin Hirt, Thomas Holenstein, and Adam D. Smith. Detectable byzantine agreement secure against faulty majorities. In Aleta Ricciardi, editor, *Proceedings of the Twenty-First Annual ACM Symposium on Principles of Distributed Computing, PODC 2002, Monterey, California, USA, July 21-24, 2002*, pages 118–126. ACM, 2002.

[FGMY97] Yair Frankel, Peter Gemmell, Philip D. MacKenzie, and Moti Yung. Proactive rsa. In Burton S. Kaliski, editor, *CRYPTO '97*, 1997.

[Fis05] Marc Fischlin. Communication-efficient non-interactive proofs of knowledge with online extractors. In *CRYPTO*, 2005.

[FS86] Amos Fiat and Adi Shamir. How to prove yourself: Practical solutions to identification and signature problems. In *CRYPTO*, 1986.

[GG18] Rosario Gennaro and Steven Goldfeder. Fast multiparty threshold ECDSA with fast trustless setup. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*, pages 1179–1194, 2018.

[GJKR01] Rosario Gennaro, Stanislaw Jarecki, Hugo Krawczyk, and Tal Rabin. Robust threshold DSS signatures. *Inf. Comput.*, 164(1):54–84, 2001.

[GKL15] Juan A. Garay, Aggelos Kiayias, and Nikos Leonardos. The bitcoin backbone protocol: Analysis and applications. In *EUROCRYPT 2015*, pages 281–310, 2015.

[GKM+20] Vipul Goyal, Abhiram Kothapalli, Elisaweta Masserova, Bryan Parno, and Yifan Song. Storing and retrieving secrets on a blockchain. Cryptology ePrint Archive, Report 2020/504, 2020. `https://eprint.iacr.org/2020/504`.

[GL05] Shafi Goldwasser and Yehuda Lindell. Secure multi-party computation without agreement. *J. Cryptology*, 18(3):247–287, 2005.

[GMW91] Oded Goldreich, Silvio Micali, and Avi Wigderson. Proofs that yield nothing but their validity or all languages in np have zero-knowledge proof systems. *J. ACM*, 38(3):690–728, July 1991.

[GRR98] Rosario Gennaro, Michael O. Rabin, and Tal Rabin. Simplified VSS and fast-track multiparty computations with applications to threshold cryptography. In Brian A. Coan and Yehuda Afek, editors, *Proceedings of the Seventeenth Annual ACM Symposium on Principles of Distributed Computing, PODC '98, Puerto Vallarta, Mexico, June 28 - July 2, 1998*, pages 101–111. ACM, 1998.

[HJJ+97] Amir Herzberg, Markus Jakobsson, Stanislaw Jarecki, Hugo Krawczyk, and Moti Yung. Proactive public key and signature systems. In *CCS '97, Proceedings of the 4th ACM Conference on Computer and Communications Security, Zurich, Switzerland, April 1-4, 1997.*, pages 100–110, 1997.

[HJKY95] Amir Herzberg, Stanislaw Jarecki, Hugo Krawczyk, and Moti Yung. Proactive secret sharing or: How to cope with perpetual leakage. In *Advances in Cryptology - CRYPTO '95, 15th Annual International Cryptology Conference, Santa Barbara, California, USA, August 27-31, 1995, Proceedings*, pages 339–352, 1995.

[IKNP03] Yuval Ishai, Joe Kilian, Kobbi Nissim, and Erez Petrank. Extending oblivious transfers efficiently. In *CRYPTO '03*, pages 145–161, 2003.

[KMOS21] Yashvanth Kondi, Bernardo Magri, Claudio Orlandi, and Omer Shlomovits. Refresh when you wake up: Proactive threshold wallets with offline device. In *IEEE S&P*, 2021.

[KOS15] Marcel Keller, Emmanuela Orsini, and Peter Scholl. Actively secure OT extension with optimal overhead. In *CRYPTO*, 2015.

[KOS16] Marcel Keller, Emmanuela Orsini, and Peter Scholl. MASCOT: faster malicious arithmetic secure computation with oblivious transfer. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, pages 830–842. ACM, 2016.

[Kra93] D.W. Kravitz. Digital signature algorithm, jul 1993. US Patent 5,231,668.

[KZZ16] Aggelos Kiayias, Hong-Sheng Zhou, and Vassilis Zikas. Fair and robust multi-party computation using a global transaction ledger. In *EUROCRYPT 2016*, pages 705–734, 2016.

[Lin17] Yehuda Lindell. Fast secure two-party ecdsa signing. In *CRYPTO*, 2017.

[LNR18] Yehuda Lindell, Ariel Nof, and Samuel Ranellucci. Fast secure multiparty ECDSA with practical distributed key generation and applications to cryptocurrency custody. *IACR Cryptology ePrint Archive*, 2018:987, 2018.

[MP] Moxie Marlinspike and Trevor Perrin. The double ratchet algorithm, 11 2016. In *https://signal.org/docs/specifications/x3dh/x3dh.pdf*.

[MZW+19] Sai Krishna Deepak Maram, Fan Zhang, Lun Wang, Andrew Low, Yupeng Zhang, Ari Juels, and Dawn Song. CHURP: dynamic-committee proactive secret sharing. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019*, pages 2369–2386, 2019.

[Nak09] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. *Cryptography Mailing list at https://metzdowd.com*, 03 2009.

[NN05] Ventzislav Nikov and Svetla Nikova. On proactive secret sharing schemes. In Helena Handschuh and M. Anwar Hasan, editors, *Selected Areas in Cryptography*, pages 308–325, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[OY91] Rafail Ostrovsky and Moti Yung. How to withstand mobile virus attacks (extended abstract). In *PODC '91*, 1991.

[Ped91] Torben Pryds Pedersen. A threshold cryptosystem without a trusted party. In *EUROCRYPT '91*, pages 522–526, 1991.

[PSL80] Marshall C. Pease, Robert E. Shostak, and Leslie Lamport. Reaching agreement in the presence of faults. *J. ACM*, 27(2):228–234, 1980.

[Sch89] Claus-Peter Schnorr. Efficient identification and signatures for smart cards. In *CRYPTO*, 1989.

[Sho00] Victor Shoup. Practical threshold signatures. In *Proceedings of the 19th International Conference on Theory and Application of Cryptographic Techniques*, EUROCRYPT'00, pages 207–220, Berlin, Heidelberg, 2000. Springer-Verlag.

[Sta96] Markus Stadler. Publicly verifiable secret sharing. In *Advances in Cryptology - EUROCRYPT '96, International Conference on the Theory and Application of Cryptographic Techniques, Saragossa, Spain, May 12-16, 1996, Proceeding*, pages 190–199, 1996.

[Woo] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger.

A Full Proof of Multiparty Impossibility

Theorem 10.1. *Let* $\mathsf{Sig} = (\mathsf{KeyGen}, \mathsf{Sign}, \mathsf{Vrfy})$ *be a triple of algorithms that satisfies the completeness definition of signature schemes. If there exists a protocol* $\pi^{(t,n)}_{\rho\text{-}\mathsf{sign}}$ *in the* $(\mathcal{G}_{\mathsf{Ledger}}, \mathcal{F}_{\mathsf{RO}})$*-hybrid model that UC-realizes* $\mathcal{F}^{n,t}_{\mathsf{Sign}}$ *with* $n > t_\rho \geq t > 2$ *in the presence of a mobile adversary actively corrupting* $t-1$ *parties where* $t_\rho < 2(t-1)$*, then there exists a forger for* Sig *that succeeds with overwhelming probability.*

Proof. We prove this theorem by first constructing an attack on the 'real' protocol $\pi^{(t,n)}_{\rho\text{-}\mathsf{sign}}$, and then using the simulator $\mathcal{S}_{\mathsf{Sign}}$ to translate this attack to the ideal protocol in order to construct a forger for Sig.

Consider an instantiation with parameters $n > t_\rho \geq t \geq 2$ such that $t_\rho < 2(t-1)$, i.e. less than half the parties in the refresh protocol are guaranteed to be honest. Define an experiment $\mathsf{EXEC}_{\pi^{(t,n)}_{\rho\text{-}\mathsf{sign}}, \mathcal{Z}}(1^\kappa)$ with environment $\mathcal{Z}_{n,t_\rho}$ as follows:

1. Send `init` to all parties.
2. Send $(\texttt{refresh}, [1, t_\rho])$ to each party P_i where $i \in [1, t_\rho]$.
3. Send $(\texttt{wake})$ to all parties.

All instructions are implemented with the protocol $\pi^{(t,n)}_{\rho\text{-}\mathsf{sign}}$. Let $\tau_{i,j}$ denote the transcript of the private channel from party P_i to P_j. Let state_i denote the private state of party P_i after the `init` command, and state'_i denote the private state of P_i after the $(\texttt{wake})$ command (note that state'_i is essentially the 'refreshed' state for the next epoch). Let 'off' index a canonical offline party, say $\mathsf{off} = t_\rho + 1$. Finally, let pk denote the public key produced when the `init` command is run.

We now show how to construct two algorithms: Ext to extract the state of P_{off} in an epoch of the protocol where it is not corrupted, and Sign^* that uses this state in conjunction with $t-1$ corrupt parties' states to sign any given message.

Lemma A.1. *Define* $\tau_{i,j}, \mathsf{state}_i, \mathsf{state}'_i$ *for* $i, j \in [n]$*, and* off *as above, and let and* $h = t_\rho - t + 1$*. There is a pair of PPT algorithms* Ext *and* Sign^* *defined as follows:*

- Ext *:* $(\tau_{i,\mathsf{off}})_{i\in[h]}, \mathsf{state}_{\mathsf{off}} \mapsto \mathsf{state}'_{\mathsf{off}}$
- Sign^* *:* $m, \mathsf{state}'_{\mathsf{off}}, \{\mathsf{state}'_i\}_{i\in I} \mapsto \sigma$
  Where $I \subset [n] \setminus \{\mathsf{off}\}$ *and* $|I| = t-1$*, and* $m \in \{0,1\}^*$*.*

It holds that the following probability is overwhelming in κ*:*

$$\Pr\left[\ \mathsf{Vrfy}\ (\mathsf{pk},\sigma,m)=1 :\ \begin{array}{rl} \mathsf{state}'_{\mathsf{off}} \leftarrow & \mathsf{Ext}((\tau_{i,\mathsf{off}})_{i\in[h]}, \mathsf{state}_{\mathsf{off}}) \\ \sigma \leftarrow & \mathsf{Sign}^*(m, \mathsf{state}'_{\mathsf{off}}, \{\mathsf{state}'_i\}_{i\in I}) \end{array}\ \right]$$

Proof. In order to prove this lemma, we will show how to construct these algorithms.

First, some clarification on the parameters: Observe that since the maximum number of corruptions is $t-1$, the value $h = t_\rho - (t-1)$ represents the maximum guaranteed number of honest online parties in the refresh procedure. Additionally since $t > \lfloor t_\rho/2 \rfloor + 1$ it holds that the adversary may corrupt more than h

parties. For ease of exposition, assume $2h+1 = t_\rho$ so that the adversary may corrupt up to $h+1$ parties and only h parties in the refresh protocol are honest in the worst case.

Consider the same experiment $\mathsf{EXEC}_{\pi^{(t,n)}_{\rho\text{-}\mathsf{sign}}, \mathcal{Z}^*}$ run with an alternative environment $\mathcal{Z}^*_{n,t_\rho}$ that corrupts each P_i for $i \in [h+1, 2h+1]$ and issues the same commands as $\mathcal{Z}_{n,t_\rho}$, with the caveat that corrupt parties do not transmit anything on their private channels to P_{off}, i.e. $(\tau_{i,\mathsf{off}} = \bot)_{i\in[h+1,2h+1]}$.

Observe that the view of the honest parties $P_1, \cdots, P_h$ is distributed identically in both executions. This is because the private channel between each corrupt P_i for $i \in [h+1, 2h+1]$ to P_{off} is hidden by definition, and P_{off} itself does not send any messages in this experiment. This fact has the following implications:

- The transcript of honest parties' private channels to P_{off}, i.e. $(\tau_{i,\mathsf{off}})_{i\in[h]}$ is distributed in both executions.
- The collection of private states of honest parties at the end of the experiment, i.e. $(\mathsf{state}'_i)_{i\in[h]}$, is distributed the same in both experiments. In particular, at the end of both experiments, parties P_1, $\cdots, P_h$ successfully advance to the next epoch. As all honest parties must agree on the epoch when activated, it holds that P_{off} advances to the next epoch in both experiments. In particular, for any $I \subset [n] \setminus \mathsf{off}$ such that $|I| = t-1$, it must hold that implementing the instruction $(\mathtt{sign}, m, I \cup \{\mathsf{off}\})$ via $\pi^{(t,n)}_{\rho\text{-}\mathsf{sign}}$ produces a valid signature σ of m under pk.

Note that the view of P_{off} is characterized entirely by the private channel communication from $P_1, \cdots, P_h$, i.e. $(\tau_{i,\mathsf{off}})_{i\in[h]}$ which is the same in both experiments, and $\mathsf{state}_{\mathsf{off}}$ its own private state from the start of the experiment (also the same in both experiments).

As we have argued that P_{off} must successfully advance to the next epoch in both experiments, we are ready to define Ext and Sign^* as follows:

- Ext implements the $\mathtt{wake}$ instruction for P_{off} via $\pi^{(t,n)}_{\rho\text{-}\mathsf{sign}}$, using as input the entire view of P_{off}, characterized by $(\tau_{i,\mathsf{off}})_{i\in[h]}, \mathsf{state}_{\mathsf{off}}$, and outputs the private state of P_{off} for the next epoch, $\mathsf{state}'_{\mathsf{off}}$.
- Sign^* implements the $(\mathtt{sign}, m, I \cup \{\mathsf{off}\})$ instruction for $(P_i)_{i\in I}$ and P_{off} via $\pi^{(t,n)}_{\rho\text{-}\mathsf{sign}}$, using as input the private states of all of these parties $(\mathsf{state}'_i)_{i\in I\cup\{\mathsf{off}\}}$

By completeness and unanimous erasure of the protocol $\pi^{(t,n)}_{\rho\text{-}\mathsf{sign}}$, both the above algorithms succeed with overwhelming probability. This completes the proof of this lemma. ■

We now construct the environment that will actually be used by the forger. Consider an instantiation with the same parameters as earlier, $n > t_\rho \geq t \geq 2$ such that $t > \lfloor t_\rho/2 \rfloor + 1$, i.e. less than half the parties in the refresh protocol are guaranteed to be honest, and define $\mathsf{off} = t_\rho + 1$ and $h = t_\rho - t + 1$ as earlier. Define the environment $\mathcal{Z}^*$ controlling adversary $\mathcal{A}$ as follows:

1. Instruct $\mathcal{A}$ to corrupt $P_1, P_2, \cdots, P_h$ and P_{off}.
2. Send $\mathtt{init}$ to all parties.
3. Instruct $\mathcal{A}$ to uncorrupt P_{off}.
4. Send $(\mathtt{refresh}, [1, t_\rho])$ to each party P_i where $i \in [1, t_\rho]$.
5. Send $(\mathtt{wake})$ to all parties.
6. Instruct $\mathcal{A}$ to corrupt P_{h+1}.
7. The adversary $\mathcal{A}$ outputs its entire view.
8. $\mathcal{Z}^*$ outputs whatever $\mathcal{A}$ outputs.

Note that unlike the usual specification for the real/ideal process in UC [Can01] in which the environment only outputs a bit, the output of $\mathcal{Z}^*$ here is a more complex string. This is done for ease of exposition as the output of $\mathcal{Z}^*$ will be used by the forger ($\mathcal{Z}^*$ acts as a passthrough for the output of $\mathcal{A}$), there is no meaningful advantage in the real/ideal distinguishing game.

Define $\tau_{i,j}, \mathsf{state}_i, \mathsf{state}'_i$ for $i, j \in [n]$ as earlier. The output of $\mathcal{A}$ at the end of this experiment is the complete views of parties $P_1, P_2, \cdots, P_h$, the view of P_{off} prior to the `refresh` instruction, and the view of P_{h+1} after the refresh instruction. These values are sufficiently characterized by $(\tau_{i,\mathsf{off}}, \mathsf{state}_i, \mathsf{state}'_i)_{i\in[h]}$, $\mathsf{state}_{\mathsf{off}}$, and state'_{h+1} respectively.

When the instructions of $\mathcal{Z}^*$ are implemented with the protocol $\pi^{(t,n)}_{\rho\text{-}\mathsf{sign}}$, we denote the output of the resulting experiment as $\mathsf{REAL}_{\pi^{(t,n)}_{\rho\text{-}\mathsf{sign}},\mathcal{A},\mathcal{Z}^*}$. As $\pi^{(t,n)}_{\rho\text{-}\mathsf{sign}}$ UC-realizes $\mathcal{F}^{n,t}_{\mathsf{Sign}}$, there must exist a simulator $\mathcal{S}_{\mathsf{Sign}}$ which interacts with $\mathcal{Z}^*$ in place of $\mathcal{A}$, and queries $\mathcal{F}^{n,t}_{\mathsf{Sign}}$ instead of interacting with honest parties, with the output of the resulting experiment denoted $\mathsf{IDEAL}_{\mathcal{F}^{n,t}_{\mathsf{Sign}},\mathcal{S}_{\mathsf{Sign}},\mathcal{Z}^*}$. It must hold that $\mathsf{REAL}_{\pi^{(t,n)}_{\rho\text{-}\mathsf{sign}},\mathcal{A},\mathcal{Z}^*} \approx \mathsf{IDEAL}_{\mathcal{F}^{n,t}_{\mathsf{Sign}},\mathcal{S}_{\mathsf{Sign}},\mathcal{Z}^*}$. We make use of this fact when constructing the forger, i.e. the forger will run the simulator $\mathcal{S}_{\mathsf{Sign}}$ with the adversary to sample from $\mathsf{IDEAL}_{\mathcal{F}^{n,t}_{\mathsf{Sign}},\mathcal{S}_{\mathsf{Sign}},\mathcal{Z}^*}$, as it can not sample from $\mathsf{REAL}_{\pi^{(t,n)}_{\rho\text{-}\mathsf{sign}},\mathcal{A},\mathcal{Z}^*}$ without instantiating honest parties in $\pi^{(t,n)}_{\rho\text{-}\mathsf{sign}}$, for which their secret states (and hence the secret key) must be known. Additionally the challenger's public key pk can be embedded in the ideal computation using $\mathcal{F}^{n,t}_{\mathsf{Sign}}$.

We are finally ready to construct the forger for the signature scheme, which forges a signature on a given message m under a public key pk received from the challenger.

$\mathsf{Forge}(1^\kappa, \mathsf{pk}, m)$:

1. Sample $(\tau_{i,\mathsf{off}}, \mathsf{state}_i, \mathsf{state}'_i)_{i\in[h]}, \mathsf{state}_{\mathsf{off}}, \mathsf{state}'_{h+1} \leftarrow \mathsf{IDEAL}_{\mathcal{F}^{n,t}_{\mathsf{Sign}},\mathcal{S}_{\mathsf{Sign}},\mathcal{Z}^*}$ with the caveat that $\mathcal{F}^{n,t}_{\mathsf{Sign}}$ is programmed to output pk as the public key when `init` is queried by $\mathcal{S}_{\mathsf{Sign}}$. The ideal oracle $\mathcal{G}_{\mathsf{Ledger}}$ if used, is implemented as per its specification.
2. Compute $\mathsf{state}'_{\mathsf{off}} \leftarrow \mathsf{Ext}((\tau_{i,\mathsf{off}})_{i\in[h]}, \mathsf{state}_{\mathsf{off}})$
3. Compute $\sigma \leftarrow \mathsf{Sign}^*(m, \mathsf{state}'_{\mathsf{off}}, (\mathsf{state}'_i)_{i\in[h+1]})$
4. Output σ

Lemma A.2. *For all $m \in \{0,1\}^*$, the following probability is overwhelming in κ:*

$$\Pr\left[\ \mathsf{Vrfy}\ (\mathsf{pk},\sigma,m)=1 :\ \begin{array}{rl}(\mathsf{sk},\mathsf{pk}) \leftarrow & \mathsf{KeyGen}(1^\kappa)\\ \sigma \leftarrow & \mathsf{Forge}(\mathsf{pk}, m)\end{array}\ \right]$$

Proof. We have previously shown in Lemma A.1 that it is possible to forge a message under a public key pk' produced by running the real protocol $\pi^{(t,n)}_{\rho\text{-}\mathsf{sign}}$. We now show how to translate this ability in order to forge a message under a public key pk received from an external challenger (i.e. the signature experiment) using $\mathcal{S}_{\mathsf{Sign}}$ to replace honest parties from $\pi^{(t,n)}_{\rho\text{-}\mathsf{sign}}$ as well as program pk into the view of the adversary. We prove this lemma via a sequence of hybrid experiments.

Hybrid $\mathcal{H}_1$. In this hybrid experiment, Forge is run as specified, except that Step 1 is implemented using $\mathsf{REAL}_{\pi^{(t,n)}_{\rho\text{-}\mathsf{sign}},\mathcal{A},\mathcal{Z}^*}$. Let the public key produced by running $\pi^{(t,n)}_{\rho\text{-}\mathsf{sign}}$ in REAL be pk'. By Lemma A.1, the output of Forge is a valid signature on m under pk' with overwhelming probability.

Hybrid $\mathcal{H}_2$. This hybrid experiment is the same as the last, except that Step 1 is implemented using $\mathsf{IDEAL}_{\mathcal{F}^{n,t}_{\mathsf{Sign}},\mathcal{S}_{\mathsf{Sign}},\mathcal{Z}^*}$ instead. As $\mathsf{REAL}_{\pi^{(t,n)}_{\rho\text{-}\mathsf{sign}},\mathcal{A},\mathcal{Z}^*} \approx \mathsf{IDEAL}_{\mathcal{F}^{n,t}_{\mathsf{Sign}},\mathcal{S}_{\mathsf{Sign}},\mathcal{Z}^*}$, the output of Forge is distributed indistinguishably to the last experiment (i.e. a valid signature under pk' chosen by $\mathcal{F}^{n,t}_{\mathsf{Sign}}$).

Hybrid $\mathcal{H}_3$. This hybrid experiment is the same as the last, with the caveat that $\mathcal{F}^{n,t}_{\mathsf{Sign}}$ is programmed to output pk as the public key when `init` is queried by $\mathcal{S}_{\mathsf{Sign}}$, instead of pk' that $\mathcal{F}^{n,t}_{\mathsf{Sign}}$ sampled internally.

As pk and pk' are both sampled by running KeyGen with uniform randomness (by the challenger and $\mathcal{F}_{\mathsf{Sign}}^{n,t}$ respectively) it holds that $\{\mathsf{pk}\} \equiv \{\mathsf{pk}'\}$ which has the following implication:

$$\begin{aligned}
&\Pr\left[\ \mathsf{Vrfy}(\mathsf{pk},\sigma,m)=1 :\ \begin{array}{rl}(\mathsf{sk},\mathsf{pk}) \leftarrow & \mathsf{KeyGen}(1^\kappa)\\ \sigma \leftarrow & \mathcal{H}_3(m,\mathsf{pk})\end{array}\ \right]\\
&= \Pr\left[\ \mathsf{Vrfy}(\mathsf{pk}',\sigma',m)=1 :\ \begin{array}{rl}(\mathsf{sk},\mathsf{pk}) \leftarrow & \mathsf{KeyGen}(1^\kappa)\\ \sigma' \leftarrow & \mathcal{H}_2(m,\mathsf{pk})\end{array}\ \right]\\
&= 1-\mathrm{negl}(\kappa)
\end{aligned}$$

The final hybrid $\mathcal{H}_3$ is exactly the code of Forge, and outputs a valid signature on m under pk supplied by the challenger, which proves the lemma. ■

The existence of an overwhelmingly successful forger for Sig given the existence of a protocol realizing $\mathcal{F}_{\mathsf{Sign}}^{n,t}$ with offline refresh, where $n > t_\rho \geq t > 2$, in the presence of a mobile adversary where $t > \lfloor t_\rho/2 \rfloor + 1$, is guaranteed by Lemma A.2. The theorem is hence proven. ■

B Realizing $\mathcal{F}_{\mathsf{Sign}}^{n,2}$ for Schnorr

We recall below the folklore instantiation of threshold Schnorr signatures.

Protocol 5: $\pi_{\mathsf{Setup}}^{\mathsf{DKG}}$

Parameters: Elliptic Curve Group $(\mathbb{G}, G, q)$
Parties: P_i for $i \in [n]$
Ideal Oracles: $\mathcal{F}_{\mathsf{Com\text{-}ZK}}^{R_{\mathsf{DL}}}$
Outputs:

- **Common**: Public key $\mathsf{pk} \in \mathbb{G}$
- **Private**: Secret key share sk_i

---

1. Each party P_i samples a random degree-1 polynomial f_i over $\mathbb{Z}_q$
2. For all pairs of parties P_i and P_j, P_i sends $f_i(j)$ to P_j and receives $f_j(i)$ in return.
3. Each party P_i computes its point
$$f(i) := \sum_{j\in[1,n]} f_j(i)$$
4. Each P_i computes
$$T_i := f(i) \cdot G$$
and sends $(\texttt{com-proof}, \mathsf{id}_i^{\mathsf{com\text{-}zk}}, f(i), T_i)$ to $\mathcal{F}_{\mathsf{Com\text{-}ZK}}^{R_{\mathsf{DL}}}$, using a fresh, unique value for $\mathsf{id}_i^{\mathsf{com\text{-}zk}}$.
5. Upon being notified of all other parties' commitments, each party P_i releases its proof by sending $(\texttt{decom-proof}, \mathsf{id}_i^{\mathsf{com\text{-}zk}})$ to $\mathcal{F}_{\mathsf{Com\text{-}ZK}}^{R_{\mathsf{DL}}}$.
6. Each party P_i receives $(\texttt{accept}, \mathsf{id}_j^{\mathsf{com\text{-}zk}}, T_j)$ from $\mathcal{F}_{\mathsf{Com\text{-}ZK}}^{R_{\mathsf{DL}}}$ for each $j \in [1,n] \setminus \{i\}$ if P_j's proof of knowledge is valid. P_i aborts if it receives $(\texttt{fail}, \mathsf{id}_j^{\mathsf{com\text{-}zk}})$ instead for any proof, or if there exists an index $x \in [3,n]$ such that
$$\lambda_1^2(x) \cdot T_1 + \lambda_2^1(x) \cdot T_2 \neq T_x$$
7. The parties compute the shared public key as
$$\mathsf{pk} := \lambda_1^2(0) \cdot T_1 + \lambda_2^1(0) \cdot T_2$$

The above protocol is a reproduction of the distributed key generation protocol of Pedersen [Ped91], adjusted for context.

Protocol 6: $\pi^{\mathsf{R}}_{\text{Schnorr}}(\mathsf{pk}, \mathsf{sk}_b, 1-b, m)$

Parameters: Elliptic Curve Group $(\mathbb{G}, G, q)$
Parties: P_b, P_{1-b} for $b, 1-b \in [n]$
Ideal Oracles: $\mathcal{F}^{R_{\mathsf{DL}}}_{\mathsf{Com\text{-}ZK}}$
Inputs:

- **Common**: Message to be signed $m \in \{0,1\}^*$, public key $\mathsf{pk} \in \mathbb{G}$, each party's share in the exponent $\mathsf{pk}_b = \lambda_b^{1-b}(0) \cdot F(b)$ where F is the polynomial over $\mathbb{G}$ passing through $(0, \mathsf{pk})$ and $(b, f(b) \cdot G)$
- **Private**: Each party P_b has private input $\mathsf{sk}_b = \lambda_b^{1-b}(0) \cdot f(b) \in \mathbb{Z}_q$

Outputs:

- **Common**: Signing nonce $R \in \mathbb{G}$
- **Private**: Each party P_b has private output $\mathsf{state}_b \in \mathbb{Z}_q$

---

1. Include all inputs in state_n
2. Sample $k_b \leftarrow \mathbb{Z}_q$ and send $(\texttt{commit}, k_b, R_b = k_b \cdot G)$ to $\mathcal{F}^{R_{\mathsf{DL}}}_{\mathsf{Com\text{-}ZK}}$ with fresh identifier $\mathsf{id}^{\mathsf{com\text{-}zk}}_b$
3. Upon receiving $(\texttt{committed}, 1-b, \mathsf{id}^{\mathsf{com}}_{1-b})$ from $\mathcal{F}^{R_{\mathsf{DL}}}_{\mathsf{Com\text{-}ZK}}$, instruct $\mathcal{F}^{R_{\mathsf{DL}}}_{\mathsf{Com\text{-}ZK}}$ to release R_b
4. Upon receiving $(\texttt{decommitted}, 1-b, \mathsf{id}^{\mathsf{com}}_{1-b}, R_{1-b})$ from $\mathcal{F}^{R_{\mathsf{DL}}}_{\mathsf{Com\text{-}ZK}}$ if $R_{1-b} \in \mathbb{G}$ then compute
$$R = R_b + R_{1-b}$$
5. Include k_b in state_b
6. Output state_b, R

Protocol 7: $\pi^{\sigma}_{\text{Schnorr}}(\mathsf{state}_b)$

Parameters: Elliptic Curve Group $(\mathbb{G}, G, q)$
Parties: P_b, P_{1-b} for $b, 1-b \in [n]$
Ideal Oracles: $\mathcal{F}^{R_{\mathsf{DL}}}_{\mathsf{Com\text{-}ZK}}$
Inputs: (Encoded in state_b)

- **Common**: Message to be signed $m \in \{0,1\}^*$, public key $\mathsf{pk} \in \mathbb{G}$
- **Private**: Each party P_b has private input $\mathsf{sk}_b = \lambda_b^{1-b}(0) \cdot f(b) \in \mathbb{Z}_q$

---

1. Parse $k_b, m, \mathsf{sk}_b \leftarrow \mathsf{state}_b$
2. Compute
$$\sigma_b = H(R||m) \cdot \mathsf{sk}_b + k_b$$
and send σ_b to P_{1-b}
3. Upon receiving $\sigma_{1-b} \in \mathbb{Z}_q$ from P_{1-b} compute
$$\sigma = \sigma_b + \sigma_{1-b}$$
and if (σ, R) is a valid Schnorr signature under public key pk then output σ

By the linearity of the Schnorr signing equation, it is easy to verify correctness as

$$\begin{aligned}
\sigma &= \sigma_b + \sigma_{1-b} \\
&= \left(H(R||m) \cdot \lambda_b^{1-b}(0) \cdot \mathsf{sk}_b + k_b\right) \\
&\quad + \left(H(R||m) \cdot \lambda_{1-b}^{b}(0) \cdot \mathsf{sk}_{1-b} + k_{1-b}\right) \\
&= H(R||m) \cdot \left(\lambda_b^{1-b}(0) \cdot \mathsf{sk}_b + \lambda_{1-b}^{b}(0) \cdot \mathsf{sk}_{1-b}\right) \\
&\quad + (k_b + k_{1-b}) \\
&= H(R||m) \cdot \mathsf{sk} + k
\end{aligned}$$

Theorem B.1. *(Informal) The protocol* $(\pi^{\mathsf{DKG}}_{\mathsf{Setup}}, \pi^{\mathsf{R}}_{Schnorr}, \pi^{\sigma}_{Schnorr})$ *UC-realizes* $\mathcal{F}^{n,2}_{\mathsf{Sign}}$ *for* $\mathsf{Sign} = \mathsf{Sign}_{Schnorr}$ *in the* $\mathcal{F}_{\mathsf{Com}}, \mathcal{F}^{R_{\mathsf{DL}}}_{\mathsf{Com\text{-}ZK}}$*-hybrid model.*

The simulation strategy is straightforward: $\mathcal{S}^{\mathsf{R}}_{\text{Schnorr}}$ upon receiving R from the functionality sends $R_{1-b} = R - R_b$ to P_b (on behalf of $\mathcal{F}^{R_{\mathsf{DL}}}_{\mathsf{Com\text{-}ZK}}$). The simulator $\mathcal{S}^{\sigma}_{\text{Schnorr}}$ upon receiving σ from the functionality sends $\sigma_{1-b} = \sigma - \sigma_b$ to P_b on behalf of P_{1-b}. Note here that σ_b is computed by the simulator as instructed by Step 2 of $\pi^{\sigma}_{\text{Schnorr}}$ using the value k_b received on behalf of $\mathcal{F}^{R_{\mathsf{DL}}}_{\mathsf{Com\text{-}ZK}}$ in Step 2 of $\pi^{\mathsf{R}}_{\text{Schnorr}}$.

C Required Functionalities

$\mathcal{F}^{R_{\mathsf{DL}}}_{\mathsf{Com\text{-}ZK}}$ The commitment functionality allows a party P_i to commit to a value $X \in \mathbb{G}$ and reveal it to parties $\{P_j\}$ at a later point if desired, along with a proof that that P_i knows $x \in \mathbb{Z}_q$ such that $x \cdot G = X$.

Functionality 3: $\mathcal{F}^{R_{\mathsf{DL}}}_{\mathsf{Com\text{-}ZK}}$

The functionality is parameterized by the group $\mathbb{G}$ of order q generated by G, and runs with a group of parties $\vec{\mathsf{P}}$.

Commit Proof On receiving $\left(\texttt{commit-proof}, \mathsf{id}^{\mathsf{com\text{-}zk}}, x, X_i\right)$ from P_i, where $x \in \mathbb{Z}_q$ and $X_i \in \mathbb{G}$, store $\left(\mathsf{id}^{\mathsf{com\text{-}zk}}, x, X_i\right)$ and send $(\texttt{committed}, i)$ to all parties.

Decommit Proof On receiving $\left(\texttt{decom-proof}, \mathsf{id}^{\mathsf{com\text{-}zk}}\right)$ from P_i,

1. If $X = x \cdot G$, send $\left(\texttt{decommitted}, \mathsf{id}^{\mathsf{com\text{-}zk}}, i\right)$ to each $P_j \in \vec{\mathsf{P}}$
2. Otherwise send $\left(\texttt{fail}, \mathsf{id}^{\mathsf{com\text{-}zk}}, i\right)$ to each $P_j \in \vec{\mathsf{P}}$

Note that multiple parties P_j may participate.

This is a standard functionality that can be instantiated in the random oracle model to obtain folklore commitments, along with Schnorr's sigma protocol plugged into either the Fiat-Shamir [FS86] or Fischlin [Fis05] transformations to obtain a non-interactive zero-knowledge proof of knowledge of discrete logarithm. It is easy to see that the usual commitment functionality $\mathcal{F}_{\mathsf{Com}}$ can be obtained by ommitting a few components of $\mathcal{F}^{R_{\mathsf{DL}}}_{\mathsf{Com\text{-}ZK}}$.

$\mathcal{F}_{\mathsf{Coin}}$ This is a coin tossing functionality, which allows any pair of parties to publicly sample a uniform $\mathbb{Z}_q$ element.

Functionality 4: $\mathcal{F}_{\mathsf{Coin}}$

This functionality is run with two parties P_0, P_1.
On receiving $(\texttt{sample-element}, \mathsf{id}^{\mathsf{coin}}, q)$ from both P_0, P_1, sample $x \leftarrow \mathbb{Z}_q$ uniformly and send $(\mathsf{id}^{\mathsf{coin}}, x)$ to both parties as adversarially delayed output.

Realizing this functionality is easy in the $\mathcal{F}_{\mathsf{Com}}$-hybrid model: P_0 samples $x_0 \leftarrow \mathbb{Z}_q$ and sends it to $\mathcal{F}_{\mathsf{Com}}$, following which P_1 samples $x_1 \leftarrow \mathbb{Z}_q$ and sends it to P_0. Finally P_0 instructs $\mathcal{F}_{\mathsf{Com}}$ to release x_0 and the output is defined as $x = x_0 + x_1$.

$\mathcal{F}_{\mathsf{MUL}}$ Secure two party multiplication functionality, in simplified form.

Functionality 5: $\mathcal{F}_{\mathsf{MUL}}$

This functionality is run with two parties P_0, P_1.
On receiving $(\texttt{input}, \mathsf{id}^{\mathsf{coin}}, x_0)$ from P_0 and $(\texttt{input}, \mathsf{id}^{\mathsf{coin}}, x_1)$ from P_1 such that $x_0, x_1 \in \mathbb{Z}_q$, sample a uniform $(t_0, t_1) \leftarrow \mathbb{Z}_q^2$ conditioned on

$$t_0 + t_1 = x_0 \cdot x_1$$

and send t_0 to P_0 and t_1 to P_1 as adversarially delayed output.

For a more nuanced functionality that can be efficiently instantiated, along with such an instantiation based on Oblivious Transfer (which we describe how to proactivize in this work), we refer the reader to the work of Doerner et al. [DKLs19].

D Formal Definition of Offline Refresh

We build on the definition of Almansa et al. [ADN06] to a notion of mobile adversaries that accommodates 'offline' parties. We do this by having each party maintain a counter epoch written on a special tape, and define the state of the system relative to these epoch values. While in our definition the adversary $\mathcal{Z}$ may choose to activate parties in sequences that leave them in different epochs, the definition of Almansa et al. does not permit this. In particular their definition requires all honest parties to first agree that they have all successfully reached the latest epoch before the adversary is permitted to change corruptions.

Epochs Each party has a special "epoch tape" on which it writes an integer epoch. At the start of the protocol, this tape contains the value 0 for all honest parties. We use the term "system epoch" to refer to the largest epoch value written on any honest party's tape.

Operations There are two kinds of commands that the environment $\mathcal{Z}$ can send to a party: `operate`, `refresh`, and `update`. Intuitively `operate` corresponds to use of the system's service, `refresh` the candidate proactivization generation, and `update` the application of this proactivization to rerandomize parties' private state. The `operate` command will be issued to t parties simultaneously (in any realization this will require them to interact), `refresh` to t_ρ parties (also requiring interaction), and `operate` will be individual and non-interactive in its realization.

Non-degeneracy Upon being given the `refresh` command, an honest party must write the current system epoch on its epoch tape. In order to rule out degenerate realizations, we also require that if any t honest parties are given the `operate` command, the next `refresh` command sent to an honest party P will result in the system epoch being incremented.

Corruptions At any given time, there can be at most $t-1$ parties controlled by $\mathcal{Z}$. Mobility of corruptions must adhere to the following rule: $\mathcal{Z}$ may decide to "uncorrupt" a party P at any time, however before corrupting a new party $P' \in \vec{\mathsf{P}}$ it must first "leave" P, then send `refresh` to any t_ρ parties without aborting (i.e. increments the epoch counter), and finally `update` to P' before being given its internal state (and full control over subsequent actions). Note that omitting this final `update` message (i.e. allowing $\mathcal{Z}$ to corrupt P' before it has refreshed) will give $\mathcal{Z}$ the views of both P and P' from the same system epoch, in which case the system will be fully compromised. This is implied by any standard definition of proactive security. In fact, our revised definition grants $\mathcal{Z}$ more power than that of Almansa et al. [ADN06], as here not every party need refresh before $\mathcal{Z}$ changes corruptions.

Crucially we allow the system epoch to be pushed forward by *any* t_ρ *parties*, i.e. consecutive epoch increments may be enabled by completely non-overlapping sets of parties. This captures our notion of "offline refresh" where not all parties in the system need be online to move the system forward; any t_ρ parties can keep the epoch counter progressing while the others catch up at their own speed.

Offline-refresh must be non-interactive A direct implication of our definition is that one can not wait for offline parties to respond before incrementing the epoch counter. This inherently rules out standard interactive verifiable secret sharing (VSS) approaches where parties 'complain' if an adversary tries to cheat them. Previous proactive secret sharing protocols can be viewed as implementing such a VSS between epochs (either explicitly by complaints against misbehaviour, or implicitly by voting for 'good' sharings), and so a fundamentally different approach is required for the offline-refresh setting.

E Proactive Threshold ECDSA Protocol

We give the full proactive ECDSA protocol below. It shares many similarities with $\pi^{(2,n)}_{\rho\text{-}\mathsf{sign}}$ and so we underline changes in this protocol.

Protocol 8: $\pi^{(2,n)}_{\rho\text{-ECDSA}}$

Parameters: Elliptic Curve Group $(\mathbb{G}, G, q)$
Parties: P_b for $b \in [n]$
Ideal Oracles: $\mathcal{F}^{R_{\mathsf{DL}}}_{\mathsf{Com\text{-}ZK}}$, $\mathcal{G}_{\mathsf{Ledger}}$, random oracle RO
Inputs:

- **Common**: Message to be signed $m \in \{0,1\}^*$, public key $\mathsf{pk} \in \mathbb{G}$, each party's share in the exponent $\mathsf{pk}_b = \lambda_b^{1-b}(0) \cdot F(b)$ where F is the polynomial over $\mathbb{G}$ passing through $(0, \mathsf{pk})$ and $(b, f(b) \cdot G)$, epoch index $\mathsf{epoch} \in \mathbb{Z}^+$
- **Private**: Each party P_b has private input $\mathsf{sk}_b = \lambda_b^{1-b}(0) \cdot f(b) \in \mathbb{Z}_q$

---

1. **Tag R from Threshold Signature:**
   i. Run the first half of the threshold signing protocol
   $$(R, \mathsf{state}_b) \leftarrow \pi^{\mathsf{R}}_{\mathsf{Sign}}\left(\mathsf{sk}_b, 1-b, \underline{\mathsf{state}^{b,1-b}_{\mathsf{MUL}}}, m\right)$$
2. **Sample New Polynomial:**
   i. Send $(\texttt{sample-element}, \mathsf{id}^{\mathsf{coin}}_1, q)$ and $\underline{(\texttt{sample-element}, \mathsf{id}^{\mathsf{coin}}_2, q)}$ to $\mathcal{F}_{\mathsf{Coin}}$ and wait for responses $(\mathsf{id}^{\mathsf{coin}}_1, \delta)$ and $\underline{(\mathsf{id}^{\mathsf{coin}}_2, \mathsf{seed})}$ respectively
   ii. Define degree-1 polynomial f_δ over $\mathbb{Z}_q$ such that
   $$f_\delta(0) = 0 \quad \text{and} \quad f_\delta(1) = \delta$$
   iii. Compute
   $$\mathsf{sk}'_b = \mathsf{sk}_b + f_\delta(b)$$

3. **Store Tagged Refresh:**
   i. Append $(R, \mathsf{sk}'_b, \underline{\mathsf{seed}}, \mathsf{epoch})$ to rpool
   ii. Establish common nonce $K \in \mathbb{G}$ along with an additive sharing of its discrete logarithm:
      a. Sample $k_b \leftarrow \mathbb{Z}_q$, set $K_b = k_b \cdot G$ and send $(\texttt{com-proof}, \mathsf{id}_b^{\mathsf{com\text{-}zk}}, k_b, K_b)$ to $\mathcal{F}_{\mathsf{Com\text{-}ZK}}^{R_{\mathsf{DL}}}$
      b. Upon receiving $(\texttt{committed}, 1-b, \mathsf{id}_{1-b}^{\mathsf{com\text{-}zk}})$ from $\mathcal{F}_{\mathsf{Com\text{-}ZK}}^{R_{\mathsf{DL}}}$, send $(\texttt{open}, \mathsf{id}_b^{\mathsf{com\text{-}zk}})$ to $\mathcal{F}_{\mathsf{Com\text{-}ZK}}^{R_{\mathsf{DL}}}$
      c. Wait to receive $(\texttt{decommitted}, 1-b, \mathsf{id}_{1-b}^{\mathsf{com\text{-}zk}}, K_{1-b} \in \mathbb{G})$ from $\mathcal{F}_{\mathsf{Com\text{-}ZK}}^{R_{\mathsf{DL}}}$
      d. Set $K = K_b + K_{1-b}$
   iii. Compute

$$e = \mathsf{RO}(R||K||\underline{\mathsf{seed}}||\delta||\mathsf{epoch})$$
$$z_b = e \cdot \mathsf{sk}_b + k_b$$

   iv. Send z_b to P_{1-b} and wait for z_{1-b}, upon receipt verifying that

$$z_{1-b} \cdot G = e \cdot \mathsf{pk}_{1-b} + K_{1-b}$$

   and compute $z = z_b + z_{1-b}$
   v. Set $\mathsf{msg} = (R, \mathsf{epoch}, \delta, \underline{\mathsf{seed}}, K, z)$
   vi. For each $i \in [n] \setminus \{b, 1-b\}$, send msg to P_i
4. Complete the threshold signature protocol by running $\sigma \leftarrow \pi_{\mathsf{Sign}}^{\sigma}$
5. If $\sigma \neq \perp$ then set $\mathsf{tx} = (m, R, \sigma)$ and send $(\texttt{Submit}, \mathsf{sid}, \mathsf{tx})$ to $\mathcal{G}_{\mathsf{Ledger}}$

Update:

1. For each unique msg received when offline do the following:
   i. Parse $(R, \mathsf{epoch}', \delta, \underline{\mathsf{seed}}, K, z) \leftarrow \mathsf{msg}$ and if $\mathsf{epoch}' < \mathsf{epoch}$ ignore this msg
   ii. Compute $e = \mathsf{RO}(R||K||\underline{\mathsf{seed}}||\delta||\mathsf{epoch}')$ and verify that

$$z \cdot G = e \cdot \mathsf{pk} + K$$

   iii. Define degree-1 polynomial f_δ over $\mathbb{Z}_q$ such that

$$f_\delta(0) = 0 \quad \text{and} \quad f_\delta(1) = \delta$$

   and interpolate $\delta_i = f_\delta(i)$
   iv. If $\mathsf{epoch}' = \mathsf{epoch}$, compute

$$\mathsf{sk}'_i = \mathsf{sk}_i + \delta_i$$

   and append $(R, \mathsf{sk}'_i, \underline{\mathsf{seed}}, \mathsf{epoch})$ to rpool
   v. Otherwise $\mathsf{epoch}' > \mathsf{epoch}$ so append $(\mathsf{epoch}', \delta_i, \mathsf{seed}, R)$ to fpool
2. Send $(\texttt{Read})$ to $\mathcal{G}_{\mathsf{Ledger}}$ and receive $(\texttt{Read}, b)$ in response. Set BLK to be the latest blocks occurring in b since last awake, and in sequence from the earliest block, for each (σ, R) under pk encountered do the following:
   i. Find $(R, \mathsf{sk}'_i, \underline{\mathsf{seed}}, \mathsf{epoch}) \in \mathsf{rpool}$ (match by R), ignore σ if not found
   ii. Overwrite $\mathsf{sk}_i = \mathsf{sk}'_i$, set $\mathsf{epoch} = \mathsf{epoch} + 1$, and set $\mathsf{rpool} = \varnothing$
   iii. For each $j \in [n] \setminus i$ compute

$$\underline{rand_{ij} = \mathsf{RO}(i, j, \mathsf{seed})}$$

   and overwrite

$$\underline{\mathsf{state}_{\mathsf{MUL}\,ij} = \mathsf{Refresh\_MUL}(\mathsf{state}_{\mathsf{MUL}\,ij}, rand_{ij})}$$

   iv. For each $(\mathsf{epoch}, \delta_i, \underline{\mathsf{seed}}, R) \in \mathsf{fpool}$ (i.e. matching current epoch) do:
      (i) Set $\mathsf{sk}'_i = \mathsf{sk}_i + \delta_i$
      (ii) Append $(R, \mathsf{sk}'_i, \underline{\mathsf{seed}}, \mathsf{epoch})$ to rpool
      (iii) Remove this entry from fpool