How to Back Up High-Value Secret Keys

Sanjam Garg*
sanjamg@berkeley.edu
University of California, Berkeley
Berkeley, California, USA
Exponential Science Foundation
Cayman Islands

Noemi Glaeser†
noemi.glaeser@tu-darmstadt.de
Technical University of Darmstadt
Darmstadt, Germany

Abhishek Jain
abhishek@cs.jhu.edu
Johns Hopkins University
Baltimore, Maryland, USA
NTT Research
Sunnyvale, California, USA

Michael Lodder
redmike7@gmail.com
Tectonic Labs
Miami, Florida, USA

Hart Montgomery
hart.montgomery@gmail.com
Linux Foundation
San Francisco, California, USA

Abstract

Consider a cryptocurrency exchange that secures the bulk of its reserves under a small set of keys, each of which is only used to transfer cryptocurrency once a year; or the backup codes for an account login or a password manager, which are again rarely used but provide access to crucial systems or information. Securing such infrequently-used high-value secrets is crucial, but existing solutions, such as threshold wallets and 'cold' (offline) wallets, are unsatisfactory.

In this work, we envision a system that allows users to conveniently back up their rarely-used, high-value keys. This new setting necessitates a novel set of design requirements. Specifically:

- We allow user keys to be threshold secret-shared among a large number of custodians where each custodian wallet comprises of a hot (i.e., online) and a cold (i.e., offline) portion. The cold part of the wallet is not touched during the backup process (thus, it is independent of the number of system users) but must be accessed for recovery.
- We provide a mechanism to continually assure users that their keys are safely stored. This feature is critical because our system is not designed for frequent key use. We also enable proactive key refresh.
- Finally, in our approach, restoring a backed-up key is equivalent to generating a signature. Thus, signatures made by users of this system should look the same as "normal" signatures to avoid exposing holders of high-value keys to targeted attacks.

Based on these requirements, we develop new security definitions and a UC-secure protocol that implements threshold BLS signatures in our new model. Our protocol is practically efficient for the envisioned large numbers of custodians: for a 67-out-of-100 threshold configuration, creating a new backup takes 10s, while recovery takes less than 2ms.

CCS Concepts

• **Security and privacy** → **Digital signatures**; *Key management*.

Keywords

Backups, Threshold signature, BLS signature, secret sharing

ACM Reference Format:
Sanjam Garg, Noemi Glaeser, Abhishek Jain, Michael Lodder, and Hart Montgomery. 2026. How to Back Up High-Value Secret Keys . In *Proceedings of the 2026 ACM SIGSAC Conference on Computer and Communications Security (CCS '26), November 15–19, 2026, The Hague, Netherlands.* ACM, New York, NY, USA, 28 pages. https://doi.org/10.1145/3830454.3832658

1 Introduction

Reliable storage of cryptographic secret keys is challenging. This is particularly evident in the cryptocurrency ecosystem, where the stakes are especially high: typically, anyone who controls the signing key for an account can take arbitrary action on the user's behalf, including transferring out large sums of money with no recourse. The recent Bybit/Safe Wallet hack [26, 72], where 1.5 billion USD in cryptocurrency was lost to (presumably) North Korean hackers, is just the latest example of a high-profile compromise of a cryptocurrency transaction signing process.

Numerous cryptocurrency wallets have been developed to allow users to safeguard their keys while preserving ease of use. *Threshold wallets* use threshold secret-sharing systems [69] to offer an elegant middle ground between accessibility and security: user keys are secret-shared among n parties which we refer to as *custodians*, and access to t of them is required to access the key. Thus, a user's key can be recovered even if $n - t$ of the custodians become non-responsive. At the same time, an attacker needs to access t key shares to recover the key.

Custodian threshold wallets offer robust security and reliability guarantees. However, at least t out of n custodians need to remain online at all times in case a user decides to perform a transaction involving a secret key held by the custodians on the user's behalf. This requirement that the custodians are always online poses additional risks. In particular, a software vulnerability in the custodian servers could jeopardize the secrecy of the held secret shares, and

*Portions of this work were done at NTT Research.
†Portions of this work were done at NTT Research and as a student at the University of Maryland and the Max Planck Institute for Security and Privacy (MPI-SP).

This work is licensed under a Creative Commons Attribution 4.0 International License.
CCS '26, The Hague, Netherlands
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2871-6/2026/11
https://doi.org/10.1145/3830454.3832658

such a vulnerability could be easy to exploit since these servers are necessarily online and listening for requests.

Cold (i.e., offline) *wallets* avoid the aforementioned limitation of custodian hot (i.e., always online) wallets because they do not always stay online. However, this affects usability. A typical compromise is to keep limited funds in an online wallet, with most of a user's assets kept in a highly secure air-gapped wallet.

While hot and cold wallets provide reasonable security and usability tradeoffs, all systems built upon them today are inadequate for storing *rarely-used* high-value secret keys. Other approaches such as distributed storage are similarly unsuitable to our setting. We compare our solution to existing approaches in Section 1.2.

In this work, we consider users or institutions who want to securely and reliably store high-value asset keys they don't frequently need to access. Our system is also designed for users who may want to back up keys from other systems to enable recovery in case of catastrophic losses in the parent system. This is an important use case that arises in several situations, including outside the cryptocurrency ecosystem (see Appendix A).

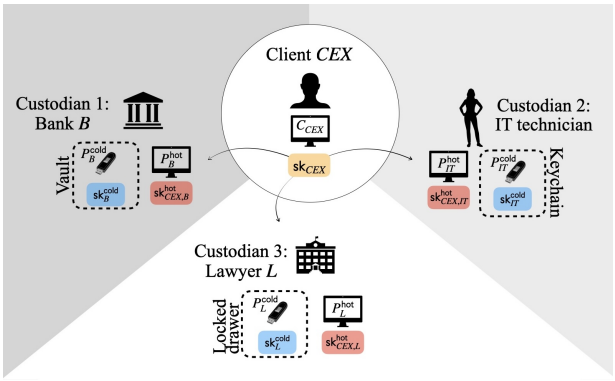


Figure 1: An example of an auditable hot-cold threshold backup with three custodians to store a high-value secret key in accordance with our novel design requirements. As shown, each custodian possesses a cold wallet which contributes to securing its online (hot wallet) portion.

1.1 Our Contribution

Our contribution is threefold: a set of design requirements and resulting system model for high-value key backups, a practical BLS-based construction called Throback, and an efficient open-source implementation. We explain each contribution in more detail below.

1.1.1 Novel Design Requirements. To our knowledge, we are the first to consider the problem of high-value key backups. Therefore, a significant part of our contribution is also to identify design requirements for this new scenario.

As we discussed, threshold wallets are not fully suited to our setting, but the idea of a distributed network is still useful. In particular, high-value keys should be split into secret shares which are spread over a network which is much more highly distributed than is common with threshold wallets. This includes geographic distribution to avoid key-share losses from natural disasters, disparate jurisdictions to safeguard against government actors, and multiple machine configurations including different hardware manufacturers and varying operating systems. Furthermore, the system must be highly scalable to more custodians than is typical (e.g., 67-out-of-100) in order to offer hardened security and high availability.

Moreover, traditional wallet systems (especially threshold wallets) prioritize easy accessibility. In contrast, our cryptographic secret keys are used infrequently, necessitating a novel set of design requirements aimed at defending against well-resourced adversaries (e.g., nation-states). In particular:

R1. **Security of each secret share is backed by a cold wallet portion.** Online machines get hacked rather frequently. Furthermore, in the case of nation-state adversaries, the exploited vulnerability can be quite sophisticated and could remain undetected for years. Thus, we require that each custodian holds each user-key's secret share jointly in a hot (i.e., online) and a cold (i.e., offline) portion of the wallet. The cold wallet should always stay offline, potentially on an air-gapped device in a secure location. This way, an adversary must physically break into the offline machine (in addition to the online hot wallet) in order to compromise a custodian's secret share.
Of course, the cold wallet must be accessed if recovery is initiated, but these (rare) instances should be the only times it will be used. Overall, recovery will access t cold wallet portions, one for each custodian involved. Consequently, the recovery process will be slow. This is not a deal-breaker, however; in our setting, recovery requests are infrequent. In fact, this additional lockup period gives time to perform due diligence in checking the veracity of a recovery request.

R2. **New backups are created without access to cold parties.** Seemingly in contradiction with the previous requirement, we insist that the backup creation process avoids accessing the cold portion of the wallet. This is crucial in practice: *accessing* any cold storage holding such high-value material (i.e., recovery) should be cumbersome by definition, but real-world systems may need to *create* key backups frequently. Implicitly, this requirement dictates that no key-specific information is stored on the cold portion of the wallet, with the upside that its memory requirements are minimal—in particular, independent of the number of users whose key shares are being stored by the custodian. This is particularly desirable since cold parties are normally resource-constrained devices such as hardware wallets.

R3. **Continual assurance of a key's safekeeping.** We also need a mechanism to continually assure users that their key shares are still safely stored. This feature is critical because a backup system for rarely-used keys is, by definition, not designed to allow users to easily make transactions (i.e., use their keys), which typically also serve as a way for users to check the availability of their keys. Instead, we require custodians to provide "proofs of remembrance" upon request, i.e., zero-knowledge proofs that they still possess their (hot or cold) key material. This enables a client to intermittently and independently audit its custodians and ensure that its key material has not been overwritten or forgotten.

Even with these new requirements, we would ideally still offer the desirable features which are common in existing (threshold) wallet systems. This includes post-compromise security (PCS) via *proactive refreshes* [60] and creating standard signatures[1] to hide system users, making it harder for attackers to identify and target users with high-value keys.

Lastly, we remark that any secret key storage solution is only as good as the strength of the keys it stores. Thus, we insist that the keys stored in our system are themselves generated via a distributed key generation (DKG) protocol, and the key shares are delivered to the custodians directly.

Going forward, we will refer to a backup system meeting the above design requirements as an *auditable hot-cold threshold backup*. An illustration of such a system is given in Figure 1.

1.1.2 Construction Based on Threshold BLS. To close this gap, we introduce an auditable hot-cold threshold backup system called Throback ("THReshold Online/offline BACKups"). Throback meets all the design requirements stated above with a construction based on threshold BLS signatures [9]. Furthermore, it inherits the non-interactivity of the signature scheme, making it highly efficient and scalable even to large, highly distributed networks of custodians. Throback produces standard BLS signatures, so it is not possible to identify users of the system (i.e., holders of high-value cryptographic keys). Finally, Throback is still compatible with standard proactive refresh of the (hot) secret shares, offered by many "standard" threshold constructions.

We prove that Throback realizes BLS threshold signatures in the universal composability (UC) model [11], that is, we reduce the security of our construction to the security of threshold BLS signatures. This allows us to avoid reproving the security of BLS and also gives us security in the concurrent setting.

1.1.3 Efficient Open-Source Implementation. We implemented Throback and show that it is practically efficient and highly scalable to essentially all reasonable settings of (t, n). Custodians can register their cold wallets in $\approx$ 500μs and produce a cold proof of remembrance in the same amount of time; computing a hot proof of remembrance takes $\approx$ 1.5ms. These times are independent of the number of custodians n. Creating a new key backup is the most expensive operation, measuring just over 10s for our largest setting ($n = 100$). However, recovery takes less than 2ms and is again independent of n. Our implementation is publicly available in a Linux Foundation lab and is Apache 2.0-licensed.

1.2 Related Work

Throback has wide applicability in any setting where high-value keys are held in escrow, offering strong features which are not met by any other existing system. Although many solutions exist for secure secret storage, including various threshold wallet implementations (e.g., Dfns [47], Zengo [77], Fireblocks [36], Coinbase [44], Web3Auth [73]) and hardware (cold) wallets (e.g., Ledger [53], Trezor [67], Coldcard [24]), none of them meet all of the necessary design requirements (Section 1.1.1), as shown in Table 1. For instance, while threshold wallets are by definition distributed (thus offering redundancy/reliable recovery), they are normally not designed to scale to hundreds of parties, with most supporting or suggesting 2-of-3 or, at most, 3-of-5 threshold schemes [74]. Similarly, auditability is not normally a design goal and is instead achieved in a roundabout way by requesting a signature on a small transaction. Thus, these two properties are marked with ◐ in Table 1 to indicate they are only somewhat achieved. As mentioned previously, however, threshold custodians are expected to remain online at all times, so this approach does not have an offline portion (R1) (but therefore vacuously meets R2). Finally, as discussed, most threshold wallets store Shamir secret shares [69] of the key, which can be proactively refreshed to offer PCS.

	Threshold wallets	Cold wallets	Threshold cold wallet	**Our system: Throback**
Highly distributed	◐	○	◐	●
Redund./reliable rec.	●	○	●	●
Offline portion (R1)	○	●	●	●
Cold-ind. backup (R2)	●	○	○	●
Auditable (R3)	◐	●	◐	●
PCS	●	○	●	●

Table 1: A comparison of secret storage approaches. ● indicates that a property is met, ○ that it is not met, and ◐ that it is only partially fulfilled (see text for details). The second row is short for "redundancy/reliable recovery".

Meanwhile, cold wallets are intentionally difficult to access, making backup creation cumbersome. They also offer no redundancy and are therefore a single point of failure, jeopardizing reliable recovery[2] and making PCS impossible. However, as the sole key holder, a cold wallet can easily be audited by checking whether the stored sk corresponds to the public key vk, e.g., whether $vk = g^{sk}$. Thus, this option only achieves R1 and R3.

A alternative hypothetical combination of these two approaches (threshold and cold wallets) would be a threshold of cold parties, which achieves the (limited) distribution, the redundancy, and PCS of threshold wallets with the added security of offline parties. Unlike Throback, however, backup creation in such a construction is by definition cumbersome, since the normally-offline cold parties must come online or be otherwise accessed whenever a new backup is created, which may happen quite frequently. Auditability is still achieved indirectly (◐) as with threshold wallets. Furthermore, a cold-only threshold solution places high storage demands on the cold wallet hardware, which is normally resource-limited. By contrast, Throback simultaneously inherits the same positive attributes of both threshold and cold wallets while offering a seamless backup creation process, minimizing the cold parties' storage, and implementing native auditability.

1.2.1 Other Related Work. Blokh et al. [6] describe yet another threshold signing protocol which combines hot and cold parties. Their protocol is tailored to ECDSA signatures, while we use BLS, and uses n hot parties and single cold party. These $(n+1)$ parties are

[1] Recall that, in our system, recovery will be equivalent to signature generation.

[2] As of 2024, it was estimated that $100 billion USD worth of Bitcoin is locked up in lost or inaccessible hardware wallets [66].

independent (i.e., the cold party is not paired with any hot party, as in our setting) and all hold equivalent shares of the ECDSA signing key; thus, their security guarantee is still a classic t-out-of-n (in their case, $n+1$) threshold guarantee, unlike our protocol, where the key remains secret up to corruption of t *pairs*. Furthermore, in their protocol, hot parties engage in MPC to generate a pre-signature, which is then finalized and output by the cold party. In Throback, on the other hand, the cold parties send messages to their corresponding hot parties, and the *hot* parties output the final signature (or signature shares).

A different line of research focuses on distributed storage systems whose goal is to outsource the storage of large amounts of data. For example, Filecoin [34] allows users to "rent" unused disk space in exchange for cryptocurrency. Data saved in such systems is replicated by multiple nodes. The result is also a highly distributed system, but unlike Throback, every node holds the *complete* data instead of a data share. This makes the system unsuitable for storing *small* amounts of *high-value* data like cryptographic keys: each storage node learns the data being stored, broadening the attack surface by exposing multiple single points of failure and also completely undermining the privacy of the data. Encrypting the information is not a solution either, since it only leads to a chicken-and-egg dilemma of how to securely store the decryption key used. Indeed, another promising application unlocked by Throback is reliable encrypted storage based on existing distributed storage solutions.

1.2.2 Open problems. BLS signatures have the advantage of being simple to understand and deploy, and they offer a fully non-interactive threshold signing process. This has led to their widespread use in production systems, including Ethereum's consensus protocol [30, §2.9.1], Filecoin [34], transactions on the Chia Network [18], and a BLS smart contract wallet [64]. With the event of account abstraction on Ethereum [32], users can specify alternative signature schemes for their transactions, further easing the adoption of BLS-based wallets. However, ECDSA [1, 3, 12, 15, 16, 25, 28, 37–39, 55, 63] and Schnorr [4, 5, 49] are also popular threshold signature schemes in the literature. We leave open the exciting problem of extending our results to these signature schemes.

2 Technical Overview

Let sk be a high-value, rarely-used secret key belonging to a user (client), and let $I_1, \ldots, I_n$ be parties (each representing a custodian institution, like a bank) who will store shares of sk. For the rest of the paper, we assume sk is a BLS signing key; in Appendix A, we explain how this is equivalent to backing up *arbitrary* (high-entropy) secrets, allowing our solution to cover backups for a general class of high-value secrets.

Each custodian institution I_i controls two parties: a hot wallet (P_i^{hot}) and a cold wallet (P_i^{cold}). Thus we represent an institution by the tuple $I_i = (P_i^{\text{hot}}, P_i^{\text{cold}})$. As in the standard threshold wallet setting, we assume the hot parties have access to a secure, authenticated broadcast channel (e.g., a blockchain). For simplicity, we model this as a shared bulletin board, which can be implemented via a (potentially permissioned) blockchain. We assume each client communicates with each of its hot custodians, and vice versa, via an authenticated encrypted channel[3] (represented by solid lines in our figures). This includes system commands (e.g., setup, recovery) sent to the custodians and messages to the bulletin board. Occasionally, a custodian's cold wallet portion will have to communicate with the hot portion; this can be done either by briefly exposing the cold wallet to online attacks, or without risking exposure by using airgapped data transfer methods such as QR codes, microSD cards, or USB drives [48]. We indicate such normally-absent, potentially airgapped connections with dotted arrows.

We assume P_i^{hot} has more storage space and computational power, while P_i^{cold} has limited storage (in particular, we want the space complexity to be independent of the number of users; see R2). Therefore, our protocol aims to minimize storage and computation on the part of the cold party.

Our system backs up the signing key sk by sharing it among a set of hot-cold custodian *pairs*. To obtain a custodian's key share sk_i, an attacker must compromise *both* its hot and cold components, and t such pairs to learn sk. Similarly, recovery (i.e., producing a signature using sk) requires both the hot *and* cold components of a threshold number of custodians to participate. This setup is different from a generic $2t$-out-of-$2n$ threshold wallet, since the attacker cannot forge a signature by corrupting any arbitrary set of $2t$ parties: corrupting, e.g., $2t$ hot parties should be of no use in forging a signature (in fact, even corrupting all n hot parties should be useless). Indeed, our threat model is instead comparable to a Boolean signing policy which requires at least t pairs of hot *and* cold parties to contribute, and is therefore much more difficult to attack since the cold components are almost always offline.

Our starting point. Recall that a BLS signature [9] on a message m consists of a single group element $\sigma := H(m)^{sk}$, where $H : \{0,1\}^* \to \mathbb{G}_1$ is a hash function. In the threshold version of BLS [7], parties holding a t-out-of-n threshold secret sharing $sk_1, \ldots, sk_n$ of the secret key can non-interactively produce threshold signatures in the same way by computing and broadcasting $\sigma_i := H(m)^{sk_i}$. These partial signatures can be verified just like a full signature, and combining t valid threshold signatures yields σ. We formally recall (threshold) BLS signatures in Section 3.1.

A natural starting point for a BLS threshold signature with the desired hot-cold access structure is for each hot-cold pair to hold a 2-out-of-2 (additive) sharing of one of the threshold shares sk_i of the high-value secret (R1). Assume for now that the hot and cold parties already hold such shares $sk_i^{\text{cold}} := r_i$ and $sk_i^{\text{hot}} := sk_i + r_i$. Due to the non-interactive and homomorphic properties of BLS, signing in this setting is relatively straightforward: to sign a message m, the cold party uses its share to compute $\sigma_i^{\text{cold}} := H(m)^{sk_i^{\text{cold}}}$, which it sends to its corresponding hot party. The hot party uses its share to compute $\sigma_i^{\text{hot}} := H(m)^{sk_i^{\text{hot}}}$ and "pre-aggregates" the two signature components to get a standard partial BLS signature $\sigma_i := \sigma_i^{\text{hot}}/\sigma_i^{\text{cold}} = H(m)^{sk_i}$. At this point, the full signature can be reconstructed non-interactively as usual from these partial signatures, allowing the system to scale to large numbers of custodians.

[3] Such channels can be established in a variety of ways (TLS certificates, biometric identification, etc.) and our system is agnostic to the specific (and potentially disparate) authentication solutions implemented by each custodian. Thus, we will not explicitly mention authentication procedures except when we wish to highlight their importance.

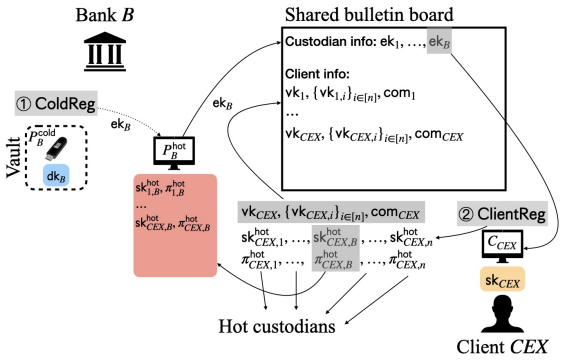


Figure 2: Cold registration & backup generation (client registration) in Throback.

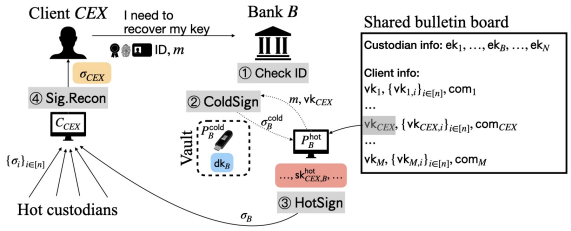


Figure 3: Secret recovery (threshold signing) in Throback. We sometimes omit the client subscript for readability.

Challenge 1: Backup generation. Setting up a new backup with our current approach requires all the parties, including the cold parties, to be online to either receive (assuming a trusted dealer) or participate in generating (via a DKG) their shares of the high-value secret key. This conflicts with our goal that generating a new backup should not require access to the cold parties (R2) and inherently precludes secret-independent cold shares and their associated storage and efficiency advantages.

To realize our desired additive sharing with secret-independent cold shares, each cold party instead samples an encryption keypair (ek_i, dk_i) when it first joins the system. It publishes ek_i and stores dk_i as a "universal" cold share for all backups. On the other hand, hot parties will store a separate value for every sk being backed up, namely an ElGamal-like encryption of the ith share sk_i under ek_i.

Recall that ElGamal ciphertexts [31] consist of two group elements of the form $(g^\rho, m \cdot ek^\rho)$, where ρ is some per-ciphertext randomness and the message m and encryption key ek are both elements of a group $\mathbb{G}$. To maintain the additive structure of the hot-cold sharings, it will be useful to have an additive mask instead of a multiplicative mask, i.e., $\mathrm{Enc}(ek, m) := (g^\rho, m + ek^\rho)$. However, $ek \in \mathbb{G}$ and our plaintexts m will be elements Shamir shares [69], which are elements of a field $\mathbb{F}$. To enable addition, we will make use of an injective function $\mathcal{H} : \mathbb{G} \to \mathbb{F}$ and redefine $\mathrm{Enc}(ek, m) := (g^\rho, m + \mathcal{H}(ek^\rho))$. Now we can set the ith hot party's share as the second element of $\mathrm{Enc}(ek_i, sk_i)$, and the cold party's share as the additive mask $\mathcal{H}(ek_i^\rho)$, which the party can compute "on-the-fly" from the first ciphertext element and the decryption key dk_i as $\mathcal{H}((g^\rho)^{dk_i}) = \mathcal{H}(ek_i^\rho)$. The pair's secret share sk_i is recovered as $sk_i^{\mathrm{hot}} - sk_i^{\mathrm{cold}} = (sk_i + \mathcal{H}(ek_i^\rho)) - \mathcal{H}(ek_i^\rho) = sk_i$. As before, the pair can also threshold-sign a message under sk_i by computing their hot and cold signatures and combining them as $\sigma_i = \sigma_i^{\mathrm{hot}} / \sigma_i^{\mathrm{cold}}$, which decrypts the key share "in the exponent".

At the moment, this construction still requires each cold party to be involved at backup generation so it can reliably store the value g^ρ necessary for re-deriving its share. This means the cold storage also scales linearly with the number of secrets being backed up, since each one requires storing a separate value g^ρ. To avoid this, we make one more tweak to the encryption scheme. Instead of sampling fresh randomness ρ for each ciphertext, ρ will be based on the backed-up secret itself (i.e., the high-value signing key sk). In particular, we use sk directly as the randomness and let the cold party's share (and the ciphertext mask) be $\mathcal{H}(ek_i^{sk})$. We also generate the encryption key $ek_i := g^{dk_i}$ using the same group $\mathbb{G}$ and generator g as the backed-up signing keypair $(sk, vk = g^{sk})$. Now the cold share can be recomputed using vk (which is a fixed, public value and does not need to be stored separately) and the decryption key dk_i as $\mathcal{H}(vk^{dk_i}) = \mathcal{H}((g^{dk_i})^{sk}) = \mathcal{H}(ek_i^{sk})$.

With this change, each cold party stores only a single element, namely dk_i, regardless of the number of clients. When it receives a signing request, it re-computes its share sk_i^{cold} as above. Consequentially, we have also achieved our goal of creating new backups without the participation of the cold parties: when backing up the high-value key sk with threshold shares $sk_1, \ldots, sk_n$, the cold shares are already implicitly fixed and we need only use the cold parties' published encryption keys to compute the corresponding hot shares as $sk_i^{\mathrm{hot}} := sk_i + \mathcal{H}(ek_i^{sk})$.

One detail still needs to be addressed: because we are no longer sampling fresh randomness for every ciphertext, the input to the function $\mathcal{H}$ is no longer uniformly random. Thus, $\mathcal{H}$ is now required to be a function whose output is close to uniformly random on high-entropy inputs (which sk is). We give an efficient construction of such a function using the leftover hash lemma in Section 4.1.[4]

Challenge 2: Proofs of remembrance. The next challenge is to devise a mechanism for hot and cold parties to prove to a client that they still have access to their shares of the high-value key (R3).

Our starting point is the notion of polynomial commitments [45], which we will see leads to a more efficient solution than general-purpose zero-knowledge proofs. Recall that a polynomial commitment allows a party to commit to some polynomial f and later verifiably open it at a point x by providing the evaluation y and a proof π which attests that $y = f(x)$. This is useful for proving knowledge of Shamir threshold secret shares [69] since such shares lie on a degree-$(t-1)$ polynomial: a dealer could publish a commitment to the polynomial used for the sharing and distribute verifiable openings, which each party can later use to prove knowledge and well-formedness of its share sk_i.

However, there are several hurdles that we must overcome to devise proofs of remembrance using this approach. First, a party proving knowledge of a share sk_i cannot respond directly with the

[4]This actually requires us to expand the encryption key to 2 elements, which we ignore in the presentation here for simplicity. Please see the body of the paper for full details.

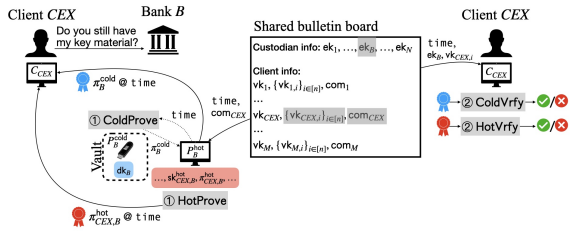


Figure 4: Proofs of remembrance in Throback.

opening proof, since the secret opening value—in this case sk_i—would be required to verify the proof. Thus, our approach requires a polynomial commitment scheme which supports private openings. Fortunately, previous work [76] has shown how to compute blinded openings for the popular KZG polynomial commitment scheme [45]. A second hurdle is that to truly show remembrance (knowledge) of sk_i at any point in time, a party needs to provide a *non-replayable* proof of the opening: simply forwarding the proof given to it by the dealer should not suffice, since the party may have deleted the key share in the meantime. This is normally achieved with an unpredictable random value, such as the output of a randomness beacon [20, 65] or a designated blockchain's most recent block hash. Specifically, in protocols using the Fiat-Shamir transform [33], the unpredictable value is included as an input to the hash function when computing the challenge. As we will see, both our hot and cold proofs fall into this class of protocols.

Furthermore, our setting is more complicated than the case of t-out-of-n secret sharing. In our construction, hot parties have shares $sk_i^{\text{hot}} := sk_i + r_i$, which are no longer guaranteed to lie on a degree-$(t-1)$ polynomial. A priori, it is unclear how a hot party with an opening proof for the original Shamir share sk_i could independently prove knowledge of the sum $sk_i + r$ without knowing sk_i or r_i, even if it was also given the value g^{r_i}. Our key observation is that while the n hot shares do not lie on a degree-$(t-1)$ polynomial, they trivially lie on a degree-$(n-1)$ polynomial, which a dealer can interpolate at backup creation. We can therefore apply our idea of polynomial commitments with private openings to the interpolated polynomial: the dealer publishes a commitment and distributes verifiable openings of the evaluations sk_i^{hot}, which each hot party can later use to create a blinded opening on demand as its proof of remembrance; a detailed description is given in Section 4.2.

As for the cold proofs, our universal cold shares mean that a cold party can use a simple proof-of-knowledge of discrete logarithm to prove remembrance of its decryption key dk_i. This suffices to prove that the party can recover the cold share $\mathcal{H}(vk^{dk_i})$ for any verification key vk, i.e., any client.

Putting it all together: Throback. We now summarize our final protocol, which is also illustrated in Figures 2 to 5. As shown, a shared bulletin board stores the public keys of each of the N total custodians registered in the system (indexed by i) and some public information for each of the M total client secrets being backed up (for simplicity, we assume each client stores a single secret and index both clients and secrets by j).

To begin (Figure 2), ① any custodian (e.g., a bank) wishing to register in the system generates an encryption keypair (ek_i, dk_i) on its cold wallet (P_i^{cold}) and posts the public key $ek_i := g^{dk_i}$ on the public bulletin board. This keypair will be reused for every backup. Recall that we instantiate the encryption scheme in the same group as the BLS signature, so encryption keypairs are in fact identical to BLS signing keypairs. Thus a custodian can simply retrieve and post the public key from its existing BLS cold wallet.

When a client C_j wishes to back up a new secret sk_j ② it specifies a set of $n \leq N$ custodians[5] (each consisting of a hot and a cold party) and a threshold t. The protocol generates t-out-of-n shares $sk_{j,1}, \ldots, sk_{j,n}$ of the secret and uses the cold parties' encryption keys $ek_1, \ldots, ek_n$ to compute the hot shares $sk_{j,i}^{\text{hot}} := sk_{j,i} + \mathcal{H}(ek_i^{sk_j})$. It interpolates the degree-$(n-1)$ polynomial defined by the hot shares and publishes a commitment com_j to it, along with the verification key $vk_j := g^{sk_j}$ and partial verification keys $vk_{j,1}, \ldots, vk_{j,n}$, where $vk_{j,i} := g^{sk_{j,i}}$. Each custodian receives $sk_{j,i}^{\text{hot}}$ and the corresponding opening proof (which will be used for the hot proofs of remembrance). Notice that the cold parties are not involved in the backup creation process.

In the rare case a client C_j initiates recovery (Figure 3), t pairs of hot and cold parties must collaborate to produce a signature on some message m using the backed-up secret sk_j. (Based on the application, m may be a cryptocurrency transaction or some fixed public message; see Appendix A.) We highlight that it is crucial that such recovery requests are accompanied by proper authentication, since the end result will be to output the high-value secret to the requester. Depending on the use case, it may be appropriate to require a client to present themselves in person to the bank or other custodian and present a physical ID, while sometimes a TLS certificate or biometric information sent over a (secure) online channel will suffice. Once ① the custodian has authenticated the request, ② it activates its cold wallet on input m, vk_j. The cold wallet uses the stored decryption key dk_i to compute the per-client secret $sk_{j,i}^{\text{cold}} := \mathcal{H}(vk_j^{dk_i})$ and output a partial signature $\sigma_{j,i}^{\text{cold}} := H(m)^{sk_{j,i}^{\text{cold}}}$. This output must now be passed to the corresponding hot party. In practice, a bank employee may retrieve the custodian's hardware (cold) wallet from a vault and briefly plug it into a bank computer for the duration of a signing procedure, where malware risks are expected to be mitigated by the wallet's secure element [52]. Cold signing could even feasibly be implemented in an airgapped manner by extending existing offline transaction flows [27, 48, 52] (e.g., PSBT [21]). Once $\sigma_{j,i}^{\text{cold}}$ has been transferred to the hot party, ③ the latter uses the client input m and its stored key share $sk_{j,i}^{\text{hot}}$ to compute $\sigma_{j,i}^{\text{hot}} := H(m)^{sk_{j,i}^{\text{hot}}}$. Then it computes a partial BLS signature as $\sigma_{j,i} := \sigma_{j,i}^{\text{hot}} / \sigma_{j,i}^{\text{cold}} = H(m)^{sk_{j,i}}$ and sends it back to the client. Once it receives enough partial signatures, ④ the client can reconstruct the full signature σ_j.

[5]The value of n can vary between clients, as long as it is upper bounded by the total number of registered custodians N. For simplicity, we leave per-client indexing of n implicit.

Finally, cold and hot parties can individually prove remembrance of a client's key shares at a particular point in time using an unpredictable timestamp (Figure 4). A cold party simply proves knowledge of the discrete logarithm of its public key ek_i (ColdProve); a hot party proves knowledge of $sk_{j,i}^{\text{hot}}$ as the ith blinded opening of the public commitment com_j, without revealing the value of $sk_{j,i}^{\text{hot}}$ (HotProve). We discuss the proof systems in detail in Section 4.2.

Proactive refresh. Like most threshold wallets, Throback is compatible with periodic refreshes of the key shares while leaving the underlying client secret sk_j unchanged (Figure 5). Specifically, the modified ElGamal encryption underlying our hot-cold secret sharing maintains (additive) malleability, allowing the share $sk_{j,i}$ to be refreshed even in encrypted form (i.e., hot share refreshes). This provides "hot-only" PCS: an attacker must learn the shares of at least t hot parties within a *single epoch* in order to exfiltrate the key[6]—otherwise, any hot key material obtained is made obsolete by the refresh operation. A client C_j can therefore refresh the shares stored by its hot custodians by ① generating t-out-of-n shares $\delta_{j,1}, \ldots, \delta_{j,n}$ of zero. It also publishes a commitment ucom_j to the polynomial underlying the zero shares. Each hot party receives $\delta_{j,i}$ and the corresponding opening proof $\zeta_{j,i}$. (Here, the client is relied on as a trusted dealer; this can be avoided by using some additional elements, shown in gray, to prove correctness of the zero sharing. We describe this "verifiable" refresh protocol in Appendix I.) Given its update information $\delta_{j,i}, \zeta_{j,i}$, each hot party ② refreshes its current backup share by computing $sk_{j,i}^{\text{hot}} + \delta_{j,i} = \text{Enc}(ek_i, sk_{j,i} + \delta_{j,i})$. To accommodate hot proofs of remembrance, we additionally require the underlying polynomial commitment scheme to be homomorphic (which KZG is) so that the opening proofs $\pi_{j,i}^{\text{hot}}$ and the commitment com_j can be updated to reflect the refreshed values. Again, see Section 4.2 for the full construction of our hot proofs of remembrance.

While hot share refreshes help raise the bar for an attacker seeking to learn sk, they also open up the opportunity for liveness disruptions in case a hot party misses some refresh information (either via benign reasons such as a dropped message, or adversarial interference such as an eclipse attack). In this case, the party would be left with an "out of sync" share. In our setting, this could eventually pose a significant problem, requiring a client to perform the expensive backup creation stage from scratch or risk permanently losing their secret key. Luckily, an advantage of our construction is that hot shares can be refreshed via *standard* proactive secret sharing (PSS) techniques (namely, adding a sharing of zero). Users of Throback can therefore implement hot share refreshes by using any existing PSS of this type [29, 42] *out of the box* and inherit its security properties. For example, to achieve liveness under asynchrony or similar adversarial models, one can plug in any existing asynchronous PSS [41, 75, 78]. For the remainder of the paper, we consider only the basic, synchronous hot refresh protocol described above to show the feasibility of PCS. Our security analysis (Appendix H) also focuses primarily on safety (i.e., keeping sk hidden), since this is the main contribution of our new construction and the hot/cold split. For a discussion of more advanced PSS techniques and an analysis of their liveness guarantees, we refer readers to the above works, and stress again that these apply directly to our construction. However, as the refreshes are only applied to the hot shares, in any case we will achieve a "hot-only" version of the corresponding PCS guarantee; we explain this in more detail in Appendix D. In the same section, we also discuss how to do cold proactive refreshes and lay out the associated design tradeoffs.

[6]Key exfiltration additionally requires an adversary to learn the keys of the t corresponding cold parties, but this can occur in any number of epochs before, during, and/or after the hot leaks.

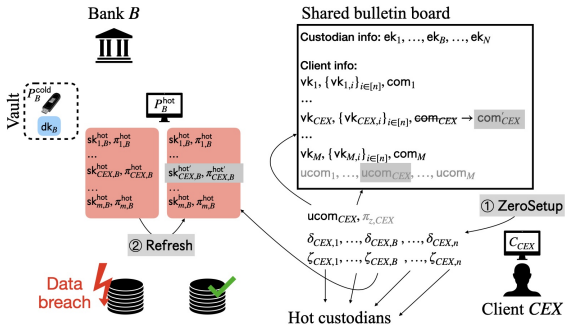


Figure 5: Proactive refresh of hot shares in Throback.

3 Preliminaries

We use $[n]$ to denote the range $\{1, \ldots, n\}$ and λ for the security parameter. We use the uppercase variable X for the free variable of a polynomial, e.g., $f(X)$. Sampling an element x uniformly at random from a set $\mathcal{X}$ is written as $x \leftarrow\!\$\ \mathcal{X}$. We use $:=$ to denote variable assignment, $y \leftarrow \text{Alg}(x)$ to assign to y the output of some algorithm Alg on input x, and $y \leftarrow\!\$\ \text{Alg}(x)$ if the algorithm is randomized. When we wish to be explicit about the randomness r used, we write $y \leftarrow \text{Alg}(x; r)$. For a non-interactive zero-knowledge proof system Π, we write $\pi \leftarrow \Pi.\text{Prove}(x; w)$ to show that the proving algorithm takes as input an instance x and witness w and outputs a proof π. Verification is written as $\Pi.\text{Verify}(x, \pi)$ and outputs a bit b.

We distinguish the key-pairs used in a signature scheme $(vk, sk$ for "verification" and "signing" key, respectively) from those used in an encryption scheme $(ek, dk$ for "encryption" and "decryption" key, respectively). We use the superscripts hot and cold, respectively, to denote the hot and cold components of some value, and a subscript i to mean that the value corresponds to the ith party. For example, the ith party's cold signature is written as σ_i^{cold}. We use the words "user" and "client" interchangeably to refer to the wallet owner.

In the remainder of this section, we present a limited set of preliminaries; the rest is delegated to Appendix B.

3.1 BLS Signatures

Definition 1 (BLS signature scheme [9])**.** Let $\mathbb{G}_1, \mathbb{G}_2$ be elliptic curve groups of order p generated by g_1 and g_2, respectively, and $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$ be a bilinear pairing. Let $H : \{0,1\}^* \to \mathbb{G}_1$ be a hash function.

- $(sk, vk) \leftarrow\!\$\ \text{KGen}(1^\lambda)$: Given a security parameter 1^λ, sample $x \leftarrow\!\$\ \mathbb{Z}_p$ and output a signing key $sk := x$ and verification key $vk := g_2^x$.

- $\sigma \leftarrow \mathsf{Sign}(sk, m)$: Given a signing key $sk \in \mathbb{Z}_p$ and a message $m \in \{0,1\}^*$, compute a signature $\sigma := H(m)^{sk}$.
- $\{0,1\} \leftarrow \mathsf{Verify}(vk, m, \sigma)$: Given a verification key $vk \in \mathbb{G}_2$, message $m \in \{0,1\}^*$, and signature $\sigma \in \mathbb{G}_1$, if $e(\sigma, g_2) = e(H(m), vk)$, output 1. Else output 0.

The security of BLS relies on the gap co-Diffie Hellman assumption (co-GDH) on $(\mathbb{G}_1, \mathbb{G}_2)$, as well as the existence of an efficiently computable homomorphism $\phi : \mathbb{G}_2 \to \mathbb{G}_1$ (i.e., e is a type-2 pairing). In practice, most implementations use type-3 pairings, and we will do the same. In this case, no efficiently computable ϕ exists, and security instead relies on a variant of the co-GDH assumption (see [9, §3.1] and [70, §2.2]); however, these assumptions are equivalent for appropriately chosen generators [17].

3.1.1 Threshold Variant. Sharing a BLS signing key $sk \in \mathbb{Z}_p$ via Shamir secret sharing (recalled in Appendix B.2) leads directly to a robust t-out-of-n threshold signature [7, 9]. More specifically, each party $i \in [n]$ receives a t-out-of-n Shamir secret share sk_i of the key, and the "partial" public keys $vk_i := g_2^{sk_i}$ are published along with vk. A partial signature is computed in the same way as a regular BLS signature as $\sigma_i := H(m)^{sk_i}$, and similarly publicly verifiable by checking if $e(\sigma_i, g_2) = e(H(m), vk_i)$). We will use $\mathsf{PVerify}(vk_i, m, \sigma_i)$ to indicate such partial signature verification. Given t valid partial signatures on a message $m \in \{0,1\}^*$, anyone can recover a full BLS signature by reconstructing sk "in the exponent" to get $H(m)^{sk}$; we refer the reader to [9, §5.3] for details.

3.2 KZG Polynomial Commitments

Because we use asymmetric pairings for the BLS signature, we also use the asymmetric version of KZG commitments, described below.

Definition 2 (KZG polynomial commitments [45])**.** Let $\mathbb{G}_1, \mathbb{G}_2$ be elliptic curve groups of order p with generators g_1, g_2 and $e : \mathbb{G}_1 \times \mathbb{G}_2 \mapsto \mathbb{G}_T$ be a bilinear pairing. The following is a commitment scheme for polynomials in $\mathbb{Z}_p[X]$ with degree at most d.

- $\mathsf{crs} \leftarrow\!\$\ \mathsf{Setup}(d)$: Sample $\tau \leftarrow\!\$\ \mathbb{Z}_p$ and output $\mathsf{crs} := \{g_1, g_1^\tau, g_1^{\tau^2}, \ldots, g_1^{\tau^d}, g_2, g_2^\tau\}$.
- $\mathsf{com}_f \leftarrow \mathsf{Com}(\mathsf{crs}, f(X))$: Let $f(X) = a_0 + a_1 X + \cdots + a_d X^d \in \mathbb{Z}_p[X]$. Use crs to compute and output $\mathsf{com}_f := g_1^{a_0} \cdot (g_1^\tau)^{a_1} \ldots (g_1^{\tau^d})^{a_d} = g_1^{f(\tau)}$.
- $(f(i), \pi_i) \leftarrow \mathsf{Open}(\mathsf{crs}, f(X), i)$: To verifiably open $f(X)$ at i, let $q_i(X) := \frac{f(X)-f(i)}{X-i} \in \mathbb{Z}_p[X]$[7]. Then compute $\mathsf{com}_{q_i} \leftarrow \mathsf{Com}(\mathsf{crs}, q_i(X))$ and output $(f(i), \mathsf{com}_{q_i})$.
- $\{0,1\} \leftarrow \mathsf{Verify}(\mathsf{crs}, \mathsf{com}_f, i, y, \pi_i)$: To confirm $y = f(i)$, it suffices to ensure $q_i(X) = \frac{f(X)-y}{X-i}$ at $X = \tau$ by checking

$$e(\mathsf{com}_f / g_1^y, g_2) \stackrel{?}{=} e(\pi_i, g_2^\tau / g_2^i).$$

The security of KZG relies on the d-Strong Diffie Hellman assumption (d-SDH).

Notice that KZG commitments are linearly homomorphic: given $\mathsf{com}_f, \mathsf{com}_{f'}$ to two distinct polynomials $f(X), f'(X)$, the commitment to $(f + f')(X)$ is $\mathsf{com}_{f+f'} := \mathsf{com}_f \cdot \mathsf{com}_{f'}$. The same holds with the corresponding evaluation proofs $\pi_{f,i}, \pi_{f',i}$ for some (fixed) point i: the evaluation proof for the point $(f + f')(i)$ is $\pi_{f+f',i} := \pi_{f,i} \cdot \pi_{f',i}$. Scaling the polynomial $f(X)$ by a constant $c \in \mathbb{Z}_p$ is also simple: the new commitment is $\mathsf{com}_{cf} := \mathsf{com}_f^c$ and an adjusted evaluation proof is obtained as $\pi_{cf,i} := \pi_{f,i}^c$.

[7] This is a polynomial by Little Bézout's Theorem.

3.3 Leftover Hash Lemma

While there are many versions of the leftover hash lemma (LHL) [43], we use the excellent version from [57]. To define the LHL, we need to start by defining an extractor; we recall the definitions of minimum entropy and universal hash functions in Appendix B.

Definition 3 (cryptographic extractor)**.** A function $Ext : \mathcal{U} \times \mathcal{S} \to \mathcal{V}$ is a (k, ϵ) extractor if for all random variables X with $H_\infty(X) \geq k$, we have: $SD\left[(S, Ext\,(X, S)), (S, V)\right] \leq \epsilon$, where SD is statistical distance, where S is uniformly distributed over $\mathcal{S}$, and V is uniformly distributed over $\mathcal{V}$.

In other words, if an input to an extractor has "enough" entropy, then the output should be distributed statistically close to uniform.

We can now state the leftover hash lemma itself.

Lemma 1 (Leftover Hash Lemma)**.** Every universal hash function is also a (k, ϵ)-extractor for $k \geq \ell + 2log\,(1/\epsilon) - 2$, where $\ell = log|\mathcal{V}|$.

4 Throback: Auditable Hot-Cold Threshold Backups from BLS Signatures

We now formally introduce our construction, Throback. We start by describing the hot/cold key shares and proofs of remembrance in detail, and put the pieces together in Section 4.3.

4.1 Hot and Cold Key Shares

As already outlined in Section 2, our construction uses a set of custodians $I_i = (P_i^{\mathsf{hot}}, P_i^{\mathsf{cold}})$ to store threshold shares of a client C_j's high-value secret sk_j (a BLS signing key), where the ith share is held jointly between the hot and cold parties. To do this, we use a modified ElGamal encryption scheme which we now formally describe below. Like the original construction, this version is malleable (but additively rather than multiplicatively), which allows "shifting" of the message m in a ciphertext by an additive factor. Recall that we will use this to enable proactive refresh of the hot (encrypted) key shares.

Definition 4 (additive ElGamal)**.** Let $\mathbb{G}$ be a DLog-hard group of order p with generator g and $\mathcal{H} : \mathbb{G} \to \mathbb{Z}_p$ be a hash function.

- $(ek, dk) \leftarrow\!\$\ \mathsf{KGen}(1^\lambda)$: Sample $dk \leftarrow\!\$\ \mathbb{Z}_p$ and let $ek := g^{dk}$. Output (ek, dk).
- $ct \leftarrow \mathsf{Enc}(ek, m; r)$: Given an encryption key $ek \in \mathbb{G}$ and a message $m \in \mathbb{Z}_p$, use randomness $r \in \mathbb{Z}_p$ to compute the ciphertext $ct := (g^r, m + \mathcal{H}(ek^r))$.
- $m' \leftarrow \mathsf{Dec}(dk, ct)$: Given a secret key $dk \in \mathbb{Z}_p$ and a ciphertext $ct \in \mathbb{G} \times \mathbb{Z}_p$, parse ct as (ct_0, ct_1) and return $ct_1 - \mathcal{H}(ct_0^{dk})$.

In practice, we instantiate our encryption scheme over the group $\mathbb{G}_2$ used by the BLS signature scheme. Notice that Enc is written with an explicit randomness input r: in the context of Throback, instead of sampling encryption fresh randomness, we set

$sk^{\text{hot}}_{j,i} \leftarrow \text{Enc}(ek_i, sk_{j,i}; sk_j)$. This also means we can leave out the redundant first element of the ciphertext, which now equals the public value $vk_j = g_2^{sk_j}$, so that $sk^{\text{hot}}_{j,i}$ is a single group element. The corresponding decryption key dk_i is kept in the cold storage, and the cold secret $sk^{\text{cold}}_{j,i} = \mathcal{H}(vk_j^{dk_i})$ can be recomputed when necessary from dk_i and vk_j,[8] without requiring the cold party to store any per-client randomness.

As mentioned previously, however, because the input to $\mathcal{H}$ is no longer uniformly random, $\mathcal{H}$ needs to be a hash function where the leftover hash lemma (Lemma 1) holds so that its output will still be indistinguishable from uniform. As we will see, this requires two group elements as input. Therefore, in our construction we will use $ek_i := (g_2^{dk_{i,1}}, g_2^{dk_{i,2}})$ and $ct := m + \mathcal{H}(ek_{i,1}^{sk_j}, ek_{i,2}^{sk_j})$. We will sometimes write operations without expanding the key into its two elements, by which we mean elementwise application of the operation: for example, $ek_i^x = (ek_{i,1}^x, ek_{i,2}^x)$.

Although any function $\mathcal{H} : \mathbb{G}_2^2 \to \mathbb{Z}_p$ which meets the above requirements suffices, it is desirable to find a very efficient construction since $\mathcal{H}$ will have to be computed by the (resource-constrained) cold party at signing time. One such $\mathcal{H}$ is a random subset sum. Given two input elements x_1, x_2 of a group $\mathbb{G}$ of prime order p, the random subset sum requires two (public) uniform vectors $\mathbf{r}_1, \mathbf{r}_2 \in \mathbb{Z}_p^{2\ell}$, where $\mathbf{r}_k = (\mathbf{r}'_{k,0}, \mathbf{r}'_{k,1})$ for $k = 1, 2$ and $\ell = \lceil \log p \rceil$. We will use bracket notation to index into the vector, e.g., $\mathbf{r}'_{1,0}[i]$ is the ith element of $\mathbf{r}'_{1,0}$. Then $\mathcal{H}$ is defined as the randomized function

$$\mathcal{H}(x_1, x_2; \mathbf{r}_1, \mathbf{r}_2) := S(\mathbf{r}_1, \mathbf{x}_1) + S(\mathbf{r}_2, \mathbf{x}_2),$$

where $\mathbf{x}_1, \mathbf{x}_2 \in \{0,1\}^\ell$ are canonical ℓ-bit representations of the inputs $x_1, x_2 \in \mathbb{G}$ and $S : \mathbb{Z}_p^{2\ell} \times \{0,1\}^\ell \to \mathbb{Z}_p$ is the subset sum function $S((\mathbf{r}'_0, \mathbf{r}'_1), \mathbf{x}) := \sum_{b_i \in \mathbf{x}} \mathbf{r}'_{b_i}[i]$. To simplify notation going forward, we leave the randomness used in $\mathcal{H}$ implicit and write $\mathcal{H}(x_1, x_2)$. We prove in Lemma 2 that, by the leftover hash lemma, the output of $\mathcal{H}$ on uniform group elements is statistically close to uniform. It follows that (KGen, Enc, Dec) instantiated with $\mathcal{H}$ is a secure encryption scheme.

4.2 Proofs of Remembrance

To realize hot and cold proofs of remembrance, we use two non-replayable zero-knowledge proofs of knowledge (ZKPoKs); we refer the reader to [71] for the definition of a ZKPoK. Both ZKPoKs are Sigma protocols, made non-interactive via the Fiat-Shamir transform [33].

As mentioned in Section 2, to prove knowledge of the cold share, it suffices to prove knowledge the decryption key dk_i with a proof system Π_{DL} for the language $\{y \in \mathbb{G}_2 : \exists w \in \mathbb{Z}_p \text{ s.t. } y = g_2^w\}$. We use the classic Schnorr protocol [68].

The proof of knowledge for the hot parties is more complicated. At setup time, each hot party must receive, along with its encrypted share sk_i^{hot}, a proof of knowledge for the share's well-formedness. This should prove that $sk_i^{\text{hot}} = \mathcal{H}(ek_{i,1}^x, ek_{i,2}^x) + x_i$ for some secret value x such that $g_2^x = vk_j$ and all the x_i are outputs of the same t-out-of-n Shamir sharing of x. As mentioned before, we will use a polynomial commitment scheme to commit to the underlying polynomial of the shares; each hot party receives an opening as its share. Recall that the commitments must be homomorphic (to accommodate proactive refreshes) and have blind openings (to preserve secrecy of the hot shares). Our construction uses KZG commitments (see Section 3.2), which offer both of these properties.

We begin with a strawman example where the hot shares are unencrypted and then show how to adapt the construction to encrypted shares. To simplify notation, in the remainder of this subsection we will drop the subscript j on the client and backed up secret. In our description, the client acts as a trusted dealer to generate both the initial hot shares and the refresh information; we discuss this assumption further in Section 4.3 and give a protocol for verifiable proactive refreshes in Appendix I.

Since the cold parties are not involved in backup creation or hot proofs of remembrance, they are ommitted from this description.

4.2.1 PoK of (Unencrypted) Key Share. Let crs be a degree-$(t-1)$ KZG CRS.[9] At backup creation ($\mathtt{t} = 0$), the client C acts as the dealer and computes a t-out-of-n Shamir sharing of sk. Let $f_0(X) \in \mathbb{Z}_p$ be the degree-$(t-1)$ polynomial used for the sharing. C publishes a (randomized) commitment $\text{com}_0 \leftarrow\!\!\$\ \text{KZG.Com}(\text{crs}, f_0(X))$. Then it sends each hot party P_i^{hot} the opening of $f_0(X)$ at $X = i$, which consists of the key share sk_i and an evaluation proof π_i.

Whenever C wants to initiate a refresh of the backup shares and thus transition to a new epoch (incrementing $\mathtt{t}$ by 1), it computes a t-out-of-n Shamir sharing of 0. Let $z(X) \in \mathbb{Z}_p[X]$ be the degree-$(t-1)$ polynomial used for the zero sharing. C publishes the update commitment ucom to $z(X)$ and also homomorphically updates com to $\text{com}' := \text{com} \cdot \text{ucom}$. This is now a commitment to the polynomial underlying the refreshed shares, namely $f'(X) := (f+z)(X)$. Note that the secret $f'(0) = f(0) + z(0) = sk + 0$ remains unchanged. Next, C sends each P_i^{hot} its share $z(i) = \delta_i$ and the corresponding opening proof ζ_i, i.e., the verifiable opening of $z(X)$ at $X = i$.

Upon receiving its update information (δ_i, ζ_i), each P_i^{hot} verifies the opening before updating its key share sk_i and proof π_i, respectively, to $sk'_i := sk_i + \delta_i$ and $\pi'_i := \pi_i \cdot \zeta_i$. By the homomorphic nature of the KZG commitment scheme, (sk'_i, π'_i) is a verifiable opening of the current polynomial $f'(X)$ at $X = i$.

We are not done yet: given the verifiable opening (sk_i, π_i) of the current commitment com, how can P_i^{hot} prove knowledge of sk_i to a third party? It cannot use π_i directly, since it can only be verified in conjunction with the opening value sk_i, which as the high-value key share must remain secret. Instead, we will prove knowledge of sk_i in *zero-knowledge* via a blinded KZG evaluation proof. This is a special case of a protocol in previous work [76, §6.1], which makes use of Pedersen commitments and a ZKPoK of Pedersen openings Π_{ped} (recalled in Appendix B.3). The idea is to hide the polynomial evaluation (i.e., the share) within a Pedersen commitment and blind the evaluation proof with another random value. Then the prover outputs the masked values, a ZKPoK of the Pedersen opening, and a "shift" term which depends on the randomnesses used for the masking.

In more detail, P_i^{hot} commits to sk_i as $\text{com}_{\text{ped}} := g_1^{sk_i} h_1^r$ and proves knowledge of the opening as $\pi_{\text{ped}} \leftarrow \Pi_{\text{ped}}.\text{Prove}(\text{com}_{\text{ped}};$

[8] Essentially, we are splitting Dec into (i) the computation of $\mathcal{H}(ct_0^{dk})$ and (ii) the unmasking of ct_1 with the resulting value. During recovery, the former is performed by P_i^{cold} and the latter is done in the exponent by P_i^{hot}.

[9] For this exposition, we assume that the threshold t is known a priori and used to sample the CRS at during an earlier setup.

(sk_i, r)). It also samples $s \leftarrow\$ \mathbb{Z}_p$ and blinds its evaluation proof as $\overline{\pi_i} := \pi_i h_1^s$. The final hot proof of remembrance is defined as $\pi_i^{\text{hot}} := (\text{com}_{\text{ped}}, \pi_{\text{ped}}, \overline{\pi_i}, g^{s_i(\tau)})$, where $s_i(X) := -r - s(X - i)$. Upon receiving π_i^{hot}, the client accepts if and only if

$$e(\text{com}/\text{com}_{\text{ped}}, g_2) = e(\overline{\pi_i}, g_2^{\tau}/g_2^{i}) \cdot e(h_1, g_2^{s_i(\tau)})$$

(where the highlighted parts are changes to the original KZG verification check due to the blinding).

4.2.2 PoK of Encrypted Key Share. Next, we modify the previous construction to accommodate the *encrypted* key shares $sk_i^{\text{hot}} := \mathcal{H}(ek_{i,1}^{sk}, ek_{i,2}^{sk}) + sk_i$ used in the final construction. To back up its secret sk, the client C computes the sk_i's as above and then encrypts each share. Because the encrypted shares are no longer guaranteed to lie on a degree-$(t-1)$ polynomial, C instead interpolates a degree-$(n-1)$ polynomial $\tilde{f}_0(X)$ where $\tilde{f}_0(i) = sk_i^{\text{hot}}$ for $i \in [n]$. (To accommodate the higher degree, the setup must instead generate a CRS of degree $n-1$.[10]) Similar to the unencrypted case, C publishes $\text{com}_0 \leftarrow\$ \text{KZG.Com}(\text{crs}, \tilde{f}_0(X))$ and distributes to each hot party the opening of $\tilde{f}_0(X)$ at $X = i$, that is, the values $(sk_i^{\text{hot}}, \pi_i)$.

Recall that our goal is not to prove the correctness of the encryption or the sharing of sk, or even of the refresh information, which we assume hold since C is a trusted dealer. Instead, our aim is to enable the hot parties to prove knowledge of a (potentially refreshed) value $sk_i^{\text{hot}} = \tilde{f}(i)$, where $\tilde{f}(X)$ is fixed for all parties.

Share refreshes proceed as before with C publishing a commitment ucom to a degree-$(t-1)$ polynomial $z(X)$ used to compute the zero shares. Parties receive their update information (δ_i, ζ_i) and can update their (in this case encrypted) key share and opening proof like before. This still works because our encryption scheme allows additive shifts of the plaintext via addition to the ciphertext.

Finally, using its (potentially refreshed) share sk_i^{hot}, each hot party can prove remembrance in zero-knowledge as before, namely by committing to the share and blinding its (potentially updated) evaluation proof π_i as described in the previous section. A formal description of the final proof of remembrance, called Π_{EKS}, is shown in Protocol 3 (Appendix C), and soundness and zero-knowledge follow the security proof in [76].

4.3 Full Construction

In this section, we describe our main construction, the auditable hot-cold threshold backup system Throback.

We summarize our public parameters (the same ones we have used so far) in Table 2. We also assume the existence of the following ideal functionalities, which we will use as building blocks for our protocol. We outline them briefly here and leave the formal definitions to Appendix G.

(Encrypted) secret share generation $\mathcal{F}_{\text{ESS}}$: $\mathcal{F}_{\text{ESS}}$ interacts with a client C and a set of custodians $\mathcal{P} = P_1, \ldots, P_n$ to offer the encrypted secret sharing functionality described above. The SSSetup interface is called by a client C, who provides a set of custodians $\mathcal{P}$ and their encryption keys, as well as a threshold $t \le |\mathcal{P}|$. $\mathcal{F}_{\text{ESS}}$ samples a secret x and generates encrypted secret shares as described in Section 4.2.2, outputting an encrypted share-proof pair $(\tilde{x}_i, \pi_i)$ to each P_i and publishing values com and $y := g_2^x$. The ZeroSetup generates share refresh information by sampling shares of zero. It outputs a pair (δ_i, ζ_i) to each P_i and publishes a commitment com_0 to the zero polynomial used. $\mathcal{F}_{\text{ESS}}$ also maintains the canonical, up-to-date values of com, y, which can be accessed at any point via the AuxRecover interface.

Public key infrastructure $\mathcal{F}_{\text{PK}}$: The PKSetup interface allows a party P to sample a uniform secret (decryption) key $dk \in \mathbb{Z}_p^2$ and publish the corresponding public (encryption) key $ek := g_2^{dk} \in \mathbb{G}_2^2$. The public key ek of any party can be retrieved reliably via the PKRecover interface.

Unpredictable timestamps $\mathcal{F}_{\text{time}}^{\ell}$: Recall that for non-replayability of our proofs of remembrance, we require an unpredictable timestamp as input. To avoid restricting ourselves to a specific primitive, we simply assume an ideal functionality $\mathcal{F}_{\text{time}}^{\ell}$, parameterized by an integer $\ell \in \mathbb{N}$, which regularly samples a fresh, uniformly random ℓ-bit string. The interface GetTimestamp outputs the most recent random value `tstamp` and can check the age of any timestamp via CheckTimestamp. Our definition is inspired by the clock functionality in [13, Fig. 2] and the randomness beacon functionality in [10].

In practice, $\mathcal{F}_{\text{ESS}}$ is easily implemented by a trusted C as described previously. While this suffices for a number of applications, we model it as an abstract ideal functionality as we want our protocol to generalize to settings where security is desired even if C (i) does not have access to a source of true randomness, or (ii) is (or, may in future be) corrupted. In Appendix I, we show how to use an additional proof system Π_{Ref} and a public bulletin-board with limited programmability to realize ZeroSetup and AuxRecover without a trusted third party. We leave a fully decentralized key generation (i.e., SSSetup) to future work.

Our full construction is shown in Protocol 1. We mark calls to $\mathcal{F}_{\text{PK}}$ with underlines calls to $\mathcal{F}_{\text{ESS}}$ with highlights. For simplicity and generality, we describe backup creation as not only creating a threshold hot/cold sharing of the secret sk, but actually sampling sk (as in $\mathcal{F}_{\text{ESS}}$); the construction is easily tweaked to back up existing secrets sk by using a variation $\mathcal{F}_{\text{ESS}}$.SSSetup which takes the secret as an input instead of sampling a fresh x.

Custodians join the system by registering their cold storage via the ColdRegister protocol, which generates (ek_i, dk_i). To back up a secret, a client runs the ClientRegister protocol with a set custodians and a threshold t. This generates encrypted shares as described in Section 4.1, publishes vk, $\{vk_i\}_{i \in [n]}$, com, and sends $(sk_i^{\text{hot}}, \pi_i)$ to each hot party.

Recovering the secret is equivalent to signing a message m with sk, which C can do via TSign by sending a threshold signing request to (the online component of) each custodian,[11] which forwards the signing requests to its corresponding cold storage.[12] P_i^{hot} and P_i^{cold} individually decide whether to honor the request by producing a

[10] In practice, because different clients in the system may chose different values of n, we generate a larger degree-$(d-1)$ KZG CRS during system setup, where d is a system parameter set to the maximum allowed value of n.

[11] Remember that this request is sent to the hot parties via an authenticated channel.

Protocol 1: THROBACK (THReshold Online/offline BACKups)

Public parameters: crs, H, $\mathcal{H}$ as defined in Table 2.

/ On input $(\texttt{sid}, \text{ColdRegister}, P_i^{\text{cold}})$

Custodian Registration: To join the system, an institutional custodian $I_i = (P_i^{\text{hot}}, P_i^{\text{cold}})$ registers its cold storage as follows:

(1) P_i^{cold} samples $dk_i \leftarrow\!\$\ \mathbb{Z}_p^2$ and sets $ek_i := g_2^{dk_i}$. It saves dk_i and registers ek_i with the PKI. (If it already has a registered keypair, it skips this whole process.)
(2) Return ek_i. / Output $(\texttt{sid}, \text{ColdRegistered}, P_i^{\text{cold}}, ek_i)$

/ On input $(\texttt{sid}, \text{ClientRegister}, C, (t, \mathcal{I}))$

Backup Creation: When a client C wishes to create a new backup with a set of institutional entities $\mathcal{I} = \{(P_i^{\text{hot}}, P_i^{\text{cold}})\}_{i\in[n]}$ and a signing threshold $t \le n$:

(1) C retrieves $ek_1, \ldots, ek_n$ from the PKI. If any key is not available, it aborts.
(2) Next, C samples $sk \leftarrow\!\$\ \mathbb{Z}_p \setminus \{0\}$ and sets $vk := g_2^{sk}$. (Alternatively, if C has a preexisting secret sk to back up, it uses it instead of sampling a fresh value.) It generates t-of-n Shamir shares $sk_1, \ldots, sk_n$ of sk and computes the partial verification keys $vk_1, \ldots, vk_n$ such that $vk_i := g_2^{sk_i}\ \forall i \in [n]$. Then, it interpolates the degree-n polynomial $\tilde{f}$ such that $\tilde{f}(i) := \mathcal{H}(ek_i^{sk}) + sk_i$ for all $i \in [n]$, sets $\text{com} \leftarrow\!\$\ \text{KZG.Com}(\text{crs}, \tilde{f})$ and computes $(sk_i^{\text{hot}}, \pi_i) \leftarrow \text{KZG.Open}(\text{crs}, \tilde{f}, i)\ \forall i \in [n]$. Then it sends $(i, sk_i^{\text{hot}}, \pi_i, vk_i)$ to each P_i^{hot}. It locally stores the parameters $(vk, \{vk_i\}_{i\in[n]}, \text{com}, \mathcal{I}, t)$ in a list D. Finally, C returns vk. / Output $(\texttt{sid}, \text{ClientRegistered}, C, vk)$
(3) Each P_i^{hot} stores the values it received and the identity of the client as a tuple $(C, sk_i^{\text{hot}}, \pi_i)$ in a list L_i. It returns 1 to indicate success. / Output $(\texttt{sid}, \text{ClientRegistered}, P_i^{\text{hot}}, (C, b_i = 1))$

/ On input $(\texttt{sid}, \text{TSign}, C, (\mathcal{I}, m))$

Recovery (threshold signing): To recover a backed-up secret, C asks its custodian set $\mathcal{I}$ to threshold-sign a message m:[a]

(1) C sends a signature request to all the hot and cold parties in its custodian set.
(2) For all $i \in [n]$, where $n = |\mathcal{I}|$:
  (a) P_i^{hot} ignores any requests that do not come from some C in its list L_i. For legitimate requests, it requests vk from C and also forwards the cold signing request containing (vk, m) from C to P_i^{cold}.
  (b) If P_i^{cold} decides to honor the request, it computes $sk_i^{\text{cold}} := \mathcal{H}(vk^{dk_i})$ and sends $\sigma_i^{\text{cold}} := H(m)^{sk_i^{\text{cold}}}$ to P_i^{hot}.
  (c) If P_i^{hot} decides to honor the request, it retrieves $(C, sk_i^{\text{hot}}, *, *) \in L_i$ and computes $\sigma_i^{\text{hot}} := H(m)^{sk_i^{\text{hot}}}$. Once it receives σ_i^{cold}, P_i^{hot} computes $\sigma_i := \sigma_i^{\text{hot}}/\sigma_i^{\text{cold}}$, which it sends to C.
(3) Once it receives all n partial signatures,[b] C runs standard BLS reconstruction to compute the signature σ on m, which it outputs. / Output $(\texttt{sid}, \text{TSignDone}, C, (\mathcal{I}, m, \sigma))$

/ On input $(\texttt{sid}, \text{CProof}, C, (P_i^{\text{cold}}))$

Cold Proof of Remembrance: A client C can verify that a cold party P_i^{cold} still retains its key material (namely dk_i) as follows:

(1) C sends a proof request message to P_i^{cold}.
(2) P_i^{cold} retrieves its stored keypair $ek_i = (ek_{i,1}, ek_{i,2}), dk_i = (dk_{i,1}, dk_{i,2})$ and also gets the latest unpredictable timestamp $\texttt{tstamp}$ (via $\mathcal{F}_{\text{time}}^{\ell}$.GetTimestamp). It uses these values to compute non-replayable proofs $\pi_{i,k}^{\text{cold}} \leftarrow \Pi_{\text{DL}}.\text{Prove}_{\texttt{tstamp}}(ek_{i,k}; dk_{i,k})$ for $k = 1, 2$. It sets $\pi_i^{\text{cold}} := (\pi_{i,1}^{\text{cold}}, \pi_{i,2}^{\text{cold}})$ and sends the timestamped proof $(\texttt{tstamp}, \pi_i^{\text{cold}})$ to C.
(3) To verify, C retrieves P_i^{cold}'s public key $ek_i = (ek_{i,1}, ek_{i,2})$ from the PKI and computes $b_k \leftarrow \Pi_{\text{DL}}.\text{Verify}_{\texttt{tstamp}}(ek_{i,k}, \pi_{i,k}^{\text{cold}})$ for $k = 1, 2$ (this includes checking that $\texttt{tstamp}$ is sufficiently recent via $\mathcal{F}_{\text{time}}^{\ell}$). It accepts if $b_1 \wedge b_2 = 1$ and rejects otherwise. / Output $(\texttt{sid}, \text{CProofResult}, C, (P_i^{\text{cold}}, b_1 \wedge b_2))$

/ On input $(\texttt{sid}, \text{HProof}, C, (P_i^{\text{hot}}))$

Hot Proof of Remembrance: A client C can verify that a hot party P_i^{hot} still retains its key material (namely sk_i^{hot}) as follows:

(1) C sends a proof request message to P_i^{hot}.
(2) P_i^{hot} retrieves its hot share material $(C, sk_i^{\text{hot}}, \pi_i, *) \in L_i$; it also requests the latest com from C and retrieves the latest unpredictable timestamp $\texttt{tstamp}$ (via $\mathcal{F}_{\text{time}}^{\ell}$.GetTimestamp). Then it computes a non-replayable proof $\pi_i^{\text{hot}} \leftarrow \Pi_{\text{EKS}}.\text{Prove}_{\texttt{tstamp}}((\text{crs}, \text{com}, i); (sk_i^{\text{hot}}, \pi_i))$; it sends the timestamped proof $(\texttt{tstamp}, \pi_i^{\text{hot}})$ to C.

Parameter	Domain(s)	Description	Reference
$\mathbb{G}_1, \mathbb{G}_2$	–	Groups of prime order p	–
g_1, h_1	$\mathbb{G}_1$	Generators of $\mathbb{G}_1$	–
g_2	$\mathbb{G}_2$	Generator of $\mathbb{G}_2$	–
d	$\mathbb{N}$	Maximum number of custodians per backup	–
crs	$(\mathbb{G}_1)^{d-1} \times (\mathbb{G}_2)^2$	KZG common reference string	Definition 2
com	$\mathbb{G}_1$	KZG commitment to $\tilde{f}$ (the hot shares)	Definition 2 and section 4.2.2
Signature parameters			
vk	$\mathbb{G}_2$	Client verification key	–
e	$\mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$	Efficiently computable asymmetric (type-3) pairing	Section 3.1
H	$\{0,1\}^* \to \mathbb{G}_1$	BLS hash function	Section 3.1
Encryption parameters			
ek_i	$(\mathbb{G}_2)^2$	Encryption key	Section 4.1
$\mathcal{H}$	$(\mathbb{G}_2)^2 \to \mathbb{Z}_p$	Random subset sum hash	Section 4.1

Table 2: Public parameters of Throback.

(3) To verify, C retrieves com from its list D and computes $b \leftarrow \Pi_{\mathsf{EKS}}.\mathsf{Verify}_{\mathtt{tstamp}}((\mathsf{crs}, \mathsf{com}, i), \pi_i^{\mathsf{hot}})$ (this includes checking that tstamp is sufficiently recent via $\mathcal{F}_{\mathsf{time}}^{\ell}$). It accepts if $b = 1$ and rejects otherwise. / Output $(\mathtt{sid}, \mathsf{HProofResult}, C, (P_i^{\mathsf{hot}}, b))$

/ On input $(\mathtt{sid}, \mathsf{ShareRefresh}, C)$

(Hot) Proactive Refresh: A client C can trigger a refresh of its hot key shares at any point:

(1) C retrieves its backup parameters $(vk, \{vk_i\}_{i\in[n]}, \mathsf{com}, \mathcal{I}, t) \in D$, where $\mathcal{I} = \{(P_i^{\mathsf{hot}}, P_i^{\mathsf{cold}})\}_{i\in[n]}$.

(2) Next, C generates t-of-n Shamir shares $\delta_1, \ldots, \delta_n$ of zero (let z be the polynomial used), computes $\mathsf{ucom} \leftarrow \mathsf{KZG.Com}(\mathsf{crs}, z)$ and $(\delta_i, \zeta_i) \leftarrow \mathsf{KZG.Open}(\mathsf{crs}, z, i)\ \forall i \in [n]$, and sends (δ_i, ζ_i) to each P_i^{hot}. It also locally updates com to $\mathsf{com}' := \mathsf{com} \cdot \mathsf{ucom}$ and vk_i to $vk_i' := vk_i g_2^{\delta_i}\ \forall i \in [n]$. Finally, C returns $b := 1$ to indicate success (and 0 otherwise). / Output $(\mathtt{sid}, \mathsf{ShareRefreshResult}, C, (b))$

(3) Each P_i^{hot} uses (δ_i, ζ_i) to update $(C, sk_i^{\mathsf{hot}}, \pi_i, vk_i) \in L_i$ to $(C, sk_i^{\mathsf{hot}} + \delta_i, \pi_i \cdot \zeta_i, vk_i g_2^{\delta_i})$ and outputs $b_i := 1$ to indicate success. / Output $(\mathtt{sid}, \mathsf{ShareRefreshResult}, P_i^{\mathsf{hot}}, (C, b_i))$

[a] The value of m is determined by the application; see discussion in the body of the paper.

[b] We wait for all n messages for simplicity and without loss of generality; in practice, C may begin reconstruction as soon as it has received t valid partial signatures.

partial signature σ_i^{hot} or σ_i^{cold}, then P_i^{hot} combines them to get a partial signature σ_i, which is verified before being sent to C.

CProof and HProof compute and verify proofs of remembrance as described in Section 4.2. Non-replayability is enforced by including the most recent output tstamp from $\mathcal{F}_{\mathsf{time}}^{\ell}$ as an input to the Fiat-Shamir hash; we make this explicit by including tstamp as a subscript to the Prove and Verify algorithms of Π_{EKS} and Π_{DL}. We write that P_i^{cold} sends its proof directly to C, but in practice this communication would again be sent via P_i^{hot}.

Finally, the hot key shares can be proactively refreshed via ShareRefresh, which updates the public values com, $\{vk_i\}_{i\in[n]}$ and outputs refresh information (δ_i, ζ_i) to each P_i^{hot}.

Correctness. Correctness requires that at any point in time, if there exists some set of at least t custodians who produce valid proofs of remembrance, then the output of an honest execution of TSign should reconstruct to a valid BLS signature on the input m. More formally, let $\{ek_i\}_{i\in[n]}, vk, \{vk_i\}_{i\in[n]}, \mathsf{com}$ be the public values of some client C's backup after any number of refreshes and $\pi_i^{\mathsf{hot}}, \pi_i^{\mathsf{cold}}$ for $i \in [n]$ be proofs of remembrance provided by each party. Suppose there exists a set $S \subseteq [n]$ with $|S| \geq t$ such that $\Pi_{\mathsf{EKS}}.\mathsf{Verify}(\mathsf{crs}, \mathsf{com}, \pi_i^{\mathsf{hot}}) = 1$ and $\Pi_{\mathsf{DL}}.\mathsf{Verify}((ek_i, g_2), \pi_i^{\mathsf{cold}}) = 1$ for all $i \in S$. The values σ_i computed during an honest execution of TSign equal $\sigma_i^{\mathsf{hot}}/\sigma_i^{\mathsf{cold}} = H(m)^{sk_i^{\mathsf{hot}}}/H(m)^{\mathcal{H}(vk^{dk_i})}$ by construction. If $i \in S$, then by soundness of Π_{EKS} and Π_{DL}, it holds that $sk_i^{\mathsf{hot}} = sk_i + \mathcal{H}(ek_i^{sk})$ and $ek_i = g_2^{dk_i}$ and thus $\sigma_i = H(m)^{sk_i}$. By the correctness of BLS signatures, $\mathsf{PVerify}(vk_i, m, \sigma_i) = 1$ and TSign outputs σ_i instead of $\perp$. Since $|S| \geq t$, TSign outputs t such non-$\perp$ values σ_i, and by correctness of threshold BLS, $\mathsf{BLS.Verify}(vk, m, \mathsf{BLS.Reconstruct}(\{\sigma_i\}_{i\in S})) = 1$.

Security. To prove security of Throback in the UC model, we show that the protocol UC-realizes a "hot-cold" threshold BLS functionality $\mathcal{F}_{\mathsf{HC}}$, and then argue that this functionality implements regular threshold BLS signatures. $\mathcal{F}_{\mathsf{HC}}$ is similar to our real-world

[12] In most implementations, requests would be passed to each cold party via the corresponding hot party. To prevent a malicious hot party from sending spurious requests to its cold party, one could instantiate an authenticated channel over the hot parties between the client and each cold party.

protocol, but makes adversarial actions (hot/cold leaks, tampering, or corruption) and their influence on the hot/cold signatures explicit. Intuitively, security of recovery is maintained by security and uniqueness of threshold BLS signatures, which guarantee that any tampering with the hot/cold partial signatures will result in an incorrect threshold signature σ_i unless the relationship $sk_i^{\text{hot}} - \mathcal{H}(vk^{dk_i}) = sk_i$ is maintained. The security of the proofs of remembrance is reduced to the knowledge soundness of Π_{EKS} and Π_{DL}.

This second part shows, in a sense, the "correctness" of our $\mathcal{F}_{\mathsf{HC}}$ by showing that it is a reasonable extension of threshold BLS signatures, i.e., that the hot/cold split does not introduce any security issues or leak any secrets. At its core, we do this by simulating the individual hot and cold shares using only a standard threshold BLS functionality, and in particular without knowledge of the pair's complete share sk_i unless *both parties* will be corrupted. We also simulate the hot and cold signatures given only σ_i. This shows that the the hot and cold parties *individually* have no information about the complete share sk_i; that is, the hot/cold split is secure. Due to limited space, we delegate the full proof to Appendix H.

Efficiency and compactness of cold parties. Our construction optimizes storage-, computation-, and communication-efficiency for the cold parties. Each P_i^{cold} only stores a single decryption key $dk_i \in \mathbb{G}_2^2$, regardless of the number of client secrets stored. To produce a cold partial signature, it computes two $\mathbb{G}_2$ exponentiations, one addition in $\mathbb{Z}_p$, and a single evaluation of $\mathcal{H}$, which is highly efficient since it is a simple subset sum (see Section 4.1). Computing a proof of remembrance requires 2 $\mathbb{G}_2$ exponentiations, 1 hash function evaluation (for Fiat-Shamir), and 2 additions and multiplications each in $\mathbb{Z}_p$. Finally, in terms of communication, a cold party only needs to send a single $\mathbb{G}_1$ element per signing operation, and a cold proof of remembrance consists of 2 $\mathbb{G}_2$ and 2 $\mathbb{Z}_p$ elements.

5 Implementation and Evaluation

We implemented our construction in Rust using the BLS12-381 elliptic curve.[13] Each element in $\mathbb{G}_1$ is 48 bytes in compressed form, $\mathbb{G}_2$ is 96 bytes, and $\mathbb{Z}_p$ is 32 bytes. Thus the size of vk is 96 bytes. Each ek_i is 192 bytes (2 $\mathbb{G}_2$ elements). The hot shares $\tilde{x}_i$ and share refresh information δ_i are 32 bytes, and their opening proofs (π_i and ζ_i, respectively) are 48 bytes. The cold shares dk_i are 64 bytes. The (partial) signatures $\sigma_i^{\text{hot}}, \sigma_i^{\text{cold}}$, and σ_i and the commitment com_t are 48 bytes.

We report runtimes of each subprotocol for small $(t, n) = (3, 5)$, medium $(5, 20)$, and large $(67, 100)$ parameter settings. The times for cold registration and proofs, threshold signing, and processing hot share refreshes (which includes running Π_{Ref}.HVerify to check ζ_i) are independent of (t, n) and are shown in Table 3. Similarly, proof sizes are independent of (t, n) and are given in Table 3. Runtimes for generating a backup (ClientRegister), hot share refreshes, and hot proofs depend on the specific values of (t, n) and are shown in Table 4. The benchmarks are an average over 100 iterations using a laptop with 8-core Apple M1 chip with 16GB RAM.

Because Throback is incomparable to existing secret storage solutions (see Section 1.2), there is no alternative approach to which we could make a direct efficiency comparison. A useful metric instead is to just compare to a regular threshold BLS signature. According to the IETF draft on BLS signatures [8], *plain* BLS signatures with 128 bits of security take approximately 370 μs and 2700 μs to sign and verify, respectively, on a good laptop. Thus, for a modest performance overhead even compared to "vanilla" BLS signatures, Throback provides targeted auditability, hardened security without sacrificing easy backup creation, and remains extremely practical. We also emphasize that using even the most efficient general-purpose SNARKs [40] to build a similar solution (or even just a backup solution, as in Filecoin [35]) would be significantly less efficient than what we construct with Throback.

[13] https://github.com/LF-Decentralized-Trust-labs/agora-key-share-proofs/

	Time (ms)
ColdRegister	0.521
TSign	1.577
ShareRefresh (P_i^{hot})	0.266
Cold Prove	0.537
Cold Verify	1.072
Hot Prove (Π_{EKS})	1.579
Hot Verify (Π_{EKS})	2.491

Table 3: Throback benchmarks independent of threshold

	Time (s)		
$(t, n) =$	$(3, 5)$	$(5, 20)$	$(67, 100)$
ClientRegister	0.026	0.699	10.247
ShareRefresh (C)	0.008	0.029	9.682

Table 4: Throback benchmarks for each setting of (t, n).

	$\lvert\pi\rvert$ (B)
Cold proof (Π_{DL})	256
Hot proof (Π_{EKS})	304
Refresh proof (ζ_i)	48

Table 5: Sizes for our various proof systems regardless of threshold

Acknowledgments

Sanjam Garg is supported in part by the AFOSR Award FA9550-24-1-0156, and research grants from the Bakar Fund, Peder Sather Funds, the Stellar Development Foundation, Supra Inc., Category Labs, Input Output Global, and Byzantine Research Inc. Noemi Glaeser was partially supported by the European Research Council (ERC) under the European Union's Horizon Europe research and innovation program (grant CRYPTOLAYER-101044770) and by the National Science Foundation Graduate Research Fellowship Program under Grant No. DGE 1840340. Abhishek Jain was supported in part by NSF CAREER 1942789, JP Morgan Faculty Award and research gifts from Ethereum Foundation, Stellar Development

Foundation, and Cisco. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

References

[1] Damiano Abram, Ariel Nof, Claudio Orlandi, Peter Scholl, and Omer Shlomovits. 2022. Low-Bandwidth Threshold ECDSA via Pseudorandom Correlation Generators. In *2022 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, San Francisco, CA, USA, 2554–2572. https://doi.org/10.1109/SP46214.2022.9833559

[2] Brian Armstrong. 2016. How Coinbase Builds Secure Infrastructure To Store Bitcoin In The Cloud. https://medium.com/the-coinbase-blog/how-coinbase-builds-secure-infrastructure-to-store-bitcoin-in-the-cloud-30a6504e40ba.

[3] Jean-Philippe Aumasson, Adrian Hamelink, and Omer Shlomovits. 2020. A Survey of ECDSA Threshold Signing. Cryptology ePrint Archive, Report 2020/1390. https://eprint.iacr.org/2020/1390

[4] Michele Battagliola, Riccardo Longo, and Alessio Meneghetti. 2022. Extensible Decentralized Secret Sharing and Application to Schnorr Signatures. Cryptology ePrint Archive, Report 2022/1551. https://eprint.iacr.org/2022/1551

[5] Mihir Bellare, Elizabeth C. Crites, Chelsea Komlo, Mary Maller, Stefano Tessaro, and Chenzhi Zhu. 2022. Better than Advertised Security for Non-interactive Threshold Signatures. In *Advances in Cryptology – CRYPTO 2022, Part IV (Lecture Notes in Computer Science, Vol. 13510)*, Yevgeniy Dodis and Thomas Shrimpton (Eds.). Springer, Cham, Switzerland, Santa Barbara, CA, USA, 517–550. https://doi.org/10.1007/978-3-031-15985-5_18

[6] Constantin Blokh, Nikolaos Makriyannis, and Udi Peled. 2022. Efficient Asymmetric Threshold ECDSA for MPC-based Cold Storage. Cryptology ePrint Archive, Report 2022/1296. https://eprint.iacr.org/2022/1296

[7] Alexandra Boldyreva. 2003. Threshold Signatures, Multisignatures and Blind Signatures Based on the Gap-Diffie-Hellman-Group Signature Scheme. In *PKC 2003: 6th International Workshop on Theory and Practice in Public Key Cryptography (Lecture Notes in Computer Science, Vol. 2567)*, Yvo Desmedt (Ed.). Springer Berlin Heidelberg, Germany, Miami, FL, USA, 31–46. https://doi.org/10.1007/3-540-36288-6_3

[8] Dan Boneh, Sergey Gorbunov, Riad Wahby, Hoeteck Wee, Christopher Wood, and Zhenfei Zhang. 2022. BLS Signatures. https://www.ietf.org/archive/id/draft-irtf-cfrg-bls-signature-05.html.

[9] Dan Boneh, Ben Lynn, and Hovav Shacham. 2001. Short Signatures from the Weil Pairing. In *Advances in Cryptology – ASIACRYPT 2001 (Lecture Notes in Computer Science, Vol. 2248)*, Colin Boyd (Ed.). Springer Berlin Heidelberg, Germany, Gold Coast, Australia, 514–532. https://doi.org/10.1007/3-540-45682-1_30

[10] Joseph Bonneau, Benedikt Bünz, Miranda Christ, and Yuval Efron. 2024. How Much Public Randomness Do Modern Consensus Protocols Need? Cryptology ePrint Archive, Report 2024/1794. https://eprint.iacr.org/2024/1794

[11] Ran Canetti. 2001. Universally Composable Security: A New Paradigm for Cryptographic Protocols. In *42nd Annual Symposium on Foundations of Computer Science*. IEEE Computer Society Press, Las Vegas, NV, USA, 136–145. https://doi.org/10.1109/SFCS.2001.959888

[12] Ran Canetti, Rosario Gennaro, Steven Goldfeder, Nikolaos Makriyannis, and Udi Peled. 2020. UC Non-Interactive, Proactive, Threshold ECDSA with Identifiable Aborts. In *ACM CCS 2020: 27th Conference on Computer and Communications Security*, Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna (Eds.). ACM Press, Virtual Event, USA, 1769–1787. https://doi.org/10.1145/3372297.3423367

[13] Ran Canetti, Kyle Hogan, Aanchal Malhotra, and Mayank Varia. 2017. A Universally Composable Treatment of Network Time. In *CSF 2017: IEEE 30th Computer Security Foundations Symposium*, Boris Köpf and Steve Chong (Eds.). IEEE Computer Society Press, Santa Barbara, CA, USA, 360–375. https://doi.org/10.1109/CSF.2017.38

[14] Ran Canetti, Yehuda Lindell, Rafail Ostrovsky, and Amit Sahai. 2002. Universally Composable Two-Party and Multi-Party Secure Computation. Cryptology ePrint Archive, Report 2002/140. https://eprint.iacr.org/2002/140

[15] Guilhem Castagnos, Dario Catalano, Fabien Laguillaumie, Federico Savasta, and Ida Tucker. 2020. Bandwidth-Efficient Threshold EC-DSA. In *PKC 2020: 23rd International Conference on Theory and Practice of Public Key Cryptography, Part II (Lecture Notes in Computer Science, Vol. 12111)*, Aggelos Kiayias, Markulf Kohlweiss, Petros Wallden, and Vassilis Zikas (Eds.). Springer, Cham, Switzerland, Edinburgh, UK, 266–296. https://doi.org/10.1007/978-3-030-45388-6_10

[16] Guilhem Castagnos, Dario Catalano, Fabien Laguillaumie, Federico Savasta, and Ida Tucker. 2021. Bandwidth-efficient threshold EC-DSA revisited: Online/Offline Extensions, Identifiable Aborts, Proactivity and Adaptive Security. Cryptology ePrint Archive, Report 2021/291. https://eprint.iacr.org/2021/291

[17] Sanjit Chatterjee, Darrel Hankerson, Edward Knapp, and Alfred Menezes. 2010. Comparing two pairing-based aggregate signature schemes. *Designs, Codes and Cryptography* 55, 2-3 (2010), 141–167. https://doi.org/10.1007/s10623-009-9334-7

[18] Chia Network. 2024. BLS Keys. https://docs.chia.net/bls-keys/.

[19] Alessandro Chiesa, Yuncong Hu, Mary Maller, Pratyush Mishra, Psi Vesely, and Nicholas P. Ward. 2020. Marlin: Preprocessing zkSNARKs with Universal and Updatable SRS. In *Advances in Cryptology – EUROCRYPT 2020, Part I (Lecture Notes in Computer Science, Vol. 12105)*, Anne Canteaut and Yuval Ishai (Eds.). Springer, Cham, Switzerland, Zagreb, Croatia, 738–768. https://doi.org/10.1007/978-3-030-45721-1_26

[20] Kevin Choi, Aathira Manoj, and Joseph Bonneau. 2023. SoK: Distributed Randomness Beacons. In *2023 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, San Francisco, CA, USA, 75–92. https://doi.org/10.1109/SP46215.2023.10179419

[21] Ava Chow. 2017. BIP174: Partially Signed Bitcoin Transaction Format. https://en.bitcoin.it/wiki/BIP_0174.

[22] CNBC. 2010. Colonel"s Secret Recipe Gets Bodyguards. https://www.cnbc.com/2008/09/09/colonels-secret-recipe-gets-bodyguards.html.

[23] Coinbase. [n. d.]. What does Coinbase do with my digital assets? https://help.coinbase.com/en/coinbase/other-topics/legal-policies/what-does-coinbase-do-with-my-digital-assets.

[24] Coldcard. [n. d.]. Coldcard Documentation. https://coldcard.com/docs/.

[25] Ivan Damgård, Thomas Pelle Jakobsen, Jesper Buus Nielsen, Jakob Illeborg Pagter, and Michael Bæksvang Østergård. 2020. Fast Threshold ECDSA with Honest Majority. Cryptology ePrint Archive, Report 2020/501. https://eprint.iacr.org/2020/501

[26] Thibault de Lachèze-Murel and Christopher Grilhault des Fontaines. 2025. The Bybit/Safe Hack. https://www.dfns.co/article/the-bybit-safe-hack.

[27] Feather Wallet Documentation. [n. d.]. Offline transaction signing. https://docs.featherwallet.org/guides/offline-tx-signing.

[28] Jack Doerner, Yashvanth Kondi, Eysa Lee, and abhi shelat. 2018. Secure Two-party Threshold ECDSA from ECDSA Assumptions. In *2018 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, San Francisco, CA, USA, 980–997. https://doi.org/10.1109/SP.2018.00036

[29] Shlomi Dolev, Karim El Defrawy, Joshua Lampkins, Rafail Ostrovsky, and Moti Yung. 2016. Proactive Secret Sharing with a Dishonest Majority. In *SCN 16: 10th International Conference on Security in Communication Networks (Lecture Notes in Computer Science, Vol. 9841)*, Vassilis Zikas and Roberto De Prisco (Eds.). Springer, Cham, Switzerland, Amalfi, Italy, 529–548. https://doi.org/10.1007/978-3-319-44618-9_28

[30] Ben Edgington. 2023. *Upgrading Ethereum: A technical handbook on Ethereum's move to proof of stake and beyond*. https://eth2book.info/latest/.

[31] Taher ElGamal. 1984. A Public Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms. In *Advances in Cryptology – CRYPTO'84 (Lecture Notes in Computer Science, Vol. 196)*, G. R. Blakley and David Chaum (Eds.). Springer Berlin Heidelberg, Germany, Santa Barbara, CA, USA, 10–18. https://doi.org/10.1007/3-540-39568-7_2

[32] Ethereum. 2024. Account abstraction. https://ethereum.org/en/roadmap/account-abstraction/.

[33] Amos Fiat and Adi Shamir. 1987. How to Prove Yourself: Practical Solutions to Identification and Signature Problems. In *Advances in Cryptology – CRYPTO'86 (Lecture Notes in Computer Science, Vol. 263)*, Andrew M. Odlyzko (Ed.). Springer Berlin Heidelberg, Germany, Santa Barbara, CA, USA, 186–194. https://doi.org/10.1007/3-540-47721-7_12

[34] Filecoin. [n. d.]. Filecoin Spec. https://spec.filecoin.io/algorithms/crypto/signatures/.

[35] Filecoin. [n. d.]. Zero Knowledge and the Filecoin Network. https://filecoin.io/blog/posts/zero-knowledge-and-the-filecoin-network/.

[36] Fireblocks. [n. d.]. Wallets-as-a-Service. https://www.fireblocks.com/platforms/wallets-as-a-service/.

[37] Adam Gągol, Jędrzej Kula, Damian Straszak, and Michał Świętek. 2020. Threshold ECDSA for Decentralized Asset Custody. Cryptology ePrint Archive, Report 2020/498. https://eprint.iacr.org/2020/498

[38] Rosario Gennaro and Steven Goldfeder. 2018. Fast Multiparty Threshold ECDSA with Fast Trustless Setup, See [54], 1179–1194. https://doi.org/10.1145/3243734.3243859

[39] Rosario Gennaro, Steven Goldfeder, and Arvind Narayanan. 2016. Threshold-optimal DSA/ECDSA signatures and an application to Bitcoin wallet security. Cryptology ePrint Archive, Report 2016/013. https://eprint.iacr.org/2016/013

[40] Jens Groth. 2016. On the Size of Pairing-Based Non-interactive Arguments. In *Advances in Cryptology – EUROCRYPT 2016, Part II (Lecture Notes in Computer Science, Vol. 9666)*, Marc Fischlin and Jean-Sébastien Coron (Eds.). Springer Berlin Heidelberg, Germany, Vienna, Austria, 305–326. https://doi.org/10.1007/978-3-662-49896-5_11

[41] Christoph U. Günther, Sourav Das, and Lefteris Kokoris-Kogias. 2022. Practical Asynchronous Proactive Secret Sharing and Key Refresh. Cryptology ePrint Archive, Report 2022/1586. https://eprint.iacr.org/2022/1586

[42] Amir Herzberg, Stanislaw Jarecki, Hugo Krawczyk, and Moti Yung. 1995. Proactive Secret Sharing Or: How to Cope With Perpetual Leakage. In *Advances in Cryptology – CRYPTO'95 (Lecture Notes in Computer Science, Vol. 963)*, Don Coppersmith (Ed.). Springer Berlin Heidelberg, Germany, Santa Barbara, CA, USA,

339–352. https://doi.org/10.1007/3-540-44750-4_27
[43] Russell Impagliazzo and David Zuckerman. 1989. How to Recycle Random Bits. In *30th Annual Symposium on Foundations of Computer Science*. IEEE Computer Society Press, Research Triangle Park, NC, USA, 248–253. https://doi.org/10.1109/SFCS.1989.63486
[44] Hannah Jin. 2023. Building user-focused web3 wallets at Coinbase. https://www.coinbase.com/blog/building-user-focused-web3-wallets-at-coinbase.
[45] Aniket Kate, Gregory M. Zaverucha, and Ian Goldberg. 2010. Constant-Size Commitments to Polynomials and Their Applications. In *Advances in Cryptology – ASIACRYPT 2010 (Lecture Notes in Computer Science, Vol. 6477)*, Masayuki Abe (Ed.). Springer Berlin Heidelberg, Germany, Singapore, 177–194. https://doi.org/10.1007/978-3-642-17373-8_11
[46] Jonathan Katz. 2024. Round-Optimal, Fully Secure Distributed Key Generation. In *Advances in Cryptology – CRYPTO 2024, Part VII (Lecture Notes in Computer Science, Vol. 14926)*, Leonid Reyzin and Douglas Stebila (Eds.). Springer, Cham, Switzerland, Santa Barbara, CA, USA, 285–316. https://doi.org/10.1007/978-3-031-68394-7_10
[47] Jonathan Katz and Antoine Urban. 2024. Honest-Majority Threshold ECDSA with Batch Generation of Key-Independent Presignatures. Cryptology ePrint Archive, Paper 2024/2011. https://eprint.iacr.org/2024/2011
[48] Keystone. 2025. How Offline Private Keys Sign Online Transactions: A Complete Guide. https://blog.keyst.one/how-offline-private-keys-sign-online-transactions-a-complete-guide.
[49] Chelsea Komlo and Ian Goldberg. 2020. FROST: Flexible Round-Optimized Schnorr Threshold Signatures. In *SAC 2020: 27th Annual International Workshop on Selected Areas in Cryptography (Lecture Notes in Computer Science, Vol. 12804)*, Orr Dunkelman, Michael J. Jacobson, Jr., and Colin O'Flynn (Eds.). Springer, Cham, Switzerland, Halifax, NS, Canada (Virtual Event), 34–65. https://doi.org/10.1007/978-3-030-81652-0_2
[50] Kraken. [n. d.]. Cryptocurrency Security Protocols. https://www.kraken.com/features/security.
[51] Dan Lamothe. 2014. Air Force Swears: Our Nuke Launch Code Was Never "00000000". https://foreignpolicy.com/2014/01/21/air-force-swears-our-nuke-launch-code-was-never-00000000/.
[52] Ledger. 2024. How Can You Sign Online Transactions When Your Private Key is Offline? https://www.ledger.com/academy/how-can-you-sign-online-transactions-when-your-private-key-is-offline.
[53] Ledger SAS. [n. d.]. Web3Auth MPC Architecture. https://github.com/LedgerHQ/recover-whitepaper/blob/main/LedgerRecoverTechnicalWhitePaper.pdf.
[54] David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang (Eds.). 2018. *ACM CCS 2018: 25th Conference on Computer and Communications Security*. ACM Press, Toronto, ON, Canada.
[55] Yehuda Lindell and Ariel Nof. 2018. Fast Secure Multiparty ECDSA with Practical Distributed Key Generation and Applications to Cryptocurrency Custody, See [54], 1837–1854. https://doi.org/10.1145/3243734.3243788
[56] Lit Protocol. 2024. Lit Protocol Whitepaper v1. https://github.com/LIT-Protocol/whitepaper.
[57] Biswaroop Maiti and Daniel Wichs. 2015. Lecture 4: Leftover Hash Lemma and One Way Functions. https://www.khoury.northeastern.edu/home/wichs/class/crypto-fall15/lecture4.pdf.
[58] National Institute of Standards and Technology. 2011. BitLocker Drive Encryption Security Policy. https://csrc.nist.gov/csrc/media/projects/cryptographic-module-validation-program/documents/security-policies/140sp1054.pdf.
[59] Tatsuaki Okamoto. 1993. Provably Secure and Practical Identification Schemes and Corresponding Signature Schemes. In *Advances in Cryptology – CRYPTO'92 (Lecture Notes in Computer Science, Vol. 740)*, Ernest F. Brickell (Ed.). Springer Berlin Heidelberg, Germany, Santa Barbara, CA, USA, 31–53. https://doi.org/10.1007/3-540-48071-4_3
[60] Rafail Ostrovsky and Moti Yung. 1991. How to Withstand Mobile Virus Attacks (Extended Abstract). In *10th ACM Symposium Annual on Principles of Distributed Computing*, Luigi Logrippo (Ed.). Association for Computing Machinery, Montreal, QC, Canada, 51–59. https://doi.org/10.1145/112600.112605
[61] Torben P. Pedersen. 1992. Non-Interactive and Information-Theoretic Secure Verifiable Secret Sharing. In *Advances in Cryptology – CRYPTO'91 (Lecture Notes in Computer Science, Vol. 576)*, Joan Feigenbaum (Ed.). Springer Berlin Heidelberg, Germany, Santa Barbara, CA, USA, 129–140. https://doi.org/10.1007/3-540-46766-1_9
[62] M. Pendergrast. 1993. *For God, Country, and Coca-Cola: The Unauthorized History of the Great American Soft Drink and the Company that Makes it.* Scribner's. https://books.google.com/books?id=Gc6zAAAAIAAJ
[63] Michaella Pettit. 2021. Efficient Threshold-Optimal ECDSA. In *CANS 21: 20th International Conference on Cryptology and Network Security (Lecture Notes in Computer Science, Vol. 13099)*, Mauro Conti, Marc Stevens, and Stephan Krenn (Eds.). Springer, Cham, Switzerland, Vienna, Austria, 116–135. https://doi.org/10.1007/978-3-030-92548-2_7
[64] Privacy & Scaling Explorations. 2022. BLS Wallet: Bundling up data. https://medium.com/privacy-scaling-explorations/bls-wallet-bundling-up-data-fb5424d3bdd3.
[65] Michael O. Rabin. 1983. Transaction protection by beacons. *J. Comput. System Sci.* 27, 2 (1983), 256–267. https://doi.org/10.1016/0022-0000(83)90042-9
[66] James Royal. 2024. Are your lost bitcoins gone forever? Here's how you might be able to recover them. https://www.bankrate.com/investing/how-to-recover-lost-bitcoins-and-other-crypto/.
[67] Satoshi Labs. [n. d.]. Hardware Wallets 101. https://blog.trezor.io/hardware-wallets-101/home.
[68] Claus-Peter Schnorr. 1990. Efficient Identification and Signatures for Smart Cards. In *Advances in Cryptology – CRYPTO'89 (Lecture Notes in Computer Science, Vol. 435)*, Gilles Brassard (Ed.). Springer, New York, USA, Santa Barbara, CA, USA, 239–252. https://doi.org/10.1007/0-387-34805-0_22
[69] Adi Shamir. 1979. How to Share a Secret. *Commun. ACM* 22, 11 (nov 1979), 612–613. https://doi.org/10.1145/359168.359176
[70] Nigel Smart and Frederik Vercauteren. 2005. On Computable Isomorphisms in Efficient Asymmetric Pairing Based Systems. Cryptology ePrint Archive, Report 2005/116. https://eprint.iacr.org/2005/116
[71] Justin Thaler. 2023. Proofs, arguments, and zero-knowledge. https://people.cs.georgetown.edu/jthaler/ProofsArgsAndZK.pdf.
[72] Joe Tidy. 2025. North Korean hackers cash out hundreds of millions from $1.5bn ByBit hack. *BBC* (10 Mar 2025). https://www.bbc.com/news/articles/c2kgndwwd7lo.
[73] Web3Auth. [n. d.]. Web3Auth MPC Architecture. https://web3auth.io/docs/infrastructure/mpc-architecture.
[74] Neil Woodfine. 2025. Multisig 2-of-3 vs. 3-of-5—are more keys more secure for your bitcoin? *Unchained Blog* (1 Jan 2025). https://www.unchained.com/blog/bitcoin-multisig-2-of-3-vs-3-of-5.
[75] Yunzhou Yan, Yu Xia, and Srinivas Devadas. 2022. Shanrang: Fully Asynchronous Proactive Secret Sharing with Dynamic Committees. Cryptology ePrint Archive, Report 2022/164. https://eprint.iacr.org/2022/164
[76] Arantxa Zapico, Vitalik Buterin, Dmitry Khovratovich, Mary Maller, Anca Nitulescu, and Mark Simkin. 2022. Caulk: Lookup Arguments in Sublinear Time. In *ACM CCS 2022: 29th Conference on Computer and Communications Security*, Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi (Eds.). ACM Press, Los Angeles, CA, USA, 3121–3134. https://doi.org/10.1145/3548606.3560646
[77] Zengo. [n. d.]. What is an MPC Wallet? https://zengo.com/mpc-wallet/.
[78] Lidong Zhou, Fred B Schneider, and Robbert Van Renesse. 2005. APSS: Proactive secret sharing in asynchronous systems. *ACM transactions on information and system security (TISSEC)* 8, 3 (2005), 259–286.

Open Science

All the artifacts used to generate the results of this paper have been made publicly available and have been judged as Available, Functional, and Reproducible by the CCS 2026 Artifact Evaluation Committee. The artifact details are described in Appendix J.

Ethical Considerations

Our work does not involve human subjects or real-world vulnerabilities and therefore avoids direct ethical concerns. However, this work has implications for user privacy and security since it constructs a back-up mechanism for high-value private keys. To mitigate potential risks from security flaws in our construction and implementation, we have provided a formal security proof in the paper and included a disclaimer in our source code to specify that the implementation has not been formally audited for security and should only be deployed at the user's own risk.

A Example Applications

In this section, we describe several scenarios in which users or institutions have high-value secret keys they rarely access and may wish to store them in a reliable and secure backup system.

Cryptocurrency exchanges. Consider a cryptocurrency exchange which is frequently used by individuals to store cryptocurrency long-term, in a similar way to a bank [2, 23, 50]. The net inflows and outflows of cryptocurrency from the exchange might be roughly equal over short- and mid-term periods, meaning that the exchange

rarely needs to access the bulk of its cryptocurrency deposits. However, because the value of these deposits could be extremely large, the key (or keys) safeguarding these rarely-accessed deposits need to be securely backed up in a way that reflects their value.

An individual with a high cryptocurrency net worth would be in a very similar situation: they might secure the bulk of their cryptocurrency with a key that is very rarely used and perhaps send enough funds once a year to a "hot" wallet or similar system for their yearly expenses.

Threshold wallets. Interestingly, this issue also arises in the threshold wallet setting. Many systems may remove threshold signing parties when they fail to meet some predefined criteria, e.g., exceed maximal response latency or produce invalid signature shares. However, to maintain high security, threshold wallets should maintain a minimum number of nodes. If the signing set becomes too small, parties can no longer be removed, but may still exhibit behavior that would warrant removal under normal circumstances. In such cases, a backup recovery process is needed to regenerate the signing key shares.

For example, Lit Protocol, a cryptographic key management provider offering a threshold signing network, specifies a backup recovery mechanism to restore the system to an operational state in such cases [56]. However, the system currently functions only as a snapshot and not an ideal backup system. In particular, it does not support refreshes of the key shares, reducing the security of the overall system, and has no way of checking backup integrity.

Backing up arbitrary secrets. We note that there are many applications where a user needs to remember a rarely used secret that can come from an arbitrary user-defined (but typically high-entropy) distribution. For these kinds of applications, the user can simply define the secret to be a *unique* signature on a particular message (by choosing the password to be a signature on a publicly known message, for instance), reducing the overall problem back to backing up a high-value secret signing key. Note that our construction is based on BLS signatures, which are unique.

For instance, many encrypted file and data storage systems, such as BitLocker [58], enable users to set a backup or recovery password. In some cases, the data secured by these systems is extremely valuable; losing the recovery password to the system at a time when it is needed could be disastrous. These backup passwords would (hopefully) be used very infrequently, but the system owner(s) might want to ensure they are readily available if necessary. Looking ahead, we prove our construction secure in the UC framework, which means the security guarantees of our protocol combine seamlessly with those of any existing backup storage systems to which we could apply our construction.

In addition, there are many other real-world "arbitrary" secrets that need strong protection. For instance, nuclear launch codes are (essentially) arbitrary secrets that need maximum security and are hopefully never used, but governments would certainly want extremely high assurances that they were always available; it's worth noting that the U.S. government has (probably) struggled with this in the past [51].

On non-arbitrary secrets. There are many famous secrets which have been protected by spending significant time and resources. For instance, Coca-Cola [62] and KFC [22] have famously secret recipes for their soda and eleven herbs and spices, respectively. Their secret recipe protections include heavy physical security, as well as no single person in the production process knowing all of the secret ingredients. The full recipes themselves are extremely rarely (if ever) accessed but of course extremely valuable to the respective companies.

Our framework and construction only directly back up secret keys and not arbitrary secrets, and thus cannot immediately be extended to backing up non-arbitrary secrets. However, there are some simple solutions that work for basic security models: for instance, given a random oracle RO, a secret encoded as a string $s = s_1||...||s_n$, and a public message $m = m_1||...||m_n$, one could provide all parties an "encryption" $ct = s_1 \oplus RO(\sigma_1)||...||s_n \oplus RO(\sigma_n)$, where σ_i is a *unique* signature on m_i, reducing the overall problem again to backing up a secret signing key. However, we believe this kind of problem needs considerably more study, particularly in modeling security, and thus we (mostly) leave it to future work.

B Extended Preliminaries

Definition 5 (minimum entropy)**.** For a random variable X we define the *minimum entropy* $H_\infty(x) := -log(max_x Pr[X = x])$.

Informally speaking, the minimum entropy is a measure of the likelihood of the *most likely* outcome of the random variable X.

Definition 6 (universal hash function)**.** A function $H : \mathcal{U} \times \mathcal{S} \to \mathcal{V}$ is a *universal hash function* if $\forall x \neq x' \in \mathcal{U}$, we have

$$Pr\left[H(x, S) = H(x', S)\right] = \frac{1}{|\mathcal{V}|}.$$

In words, a universal hash function is a function family where, for all pairs of inputs, the probability of a collision is uniform.

B.1 Bilinear Pairings

Definition 7 (bilinear pairing)**.** A bilinear pairing is a map $e : \mathbb{G}_1 \times \mathbb{G}_2 \mapsto \mathbb{G}_T$ where $\mathbb{G}_1, \mathbb{G}_2$, and $\mathbb{G}_T$ are cyclic groups of prime order p. (Often, $\mathbb{G}_1, \mathbb{G}_2$ are written in additive notation and $\mathbb{G}_T$ in multiplicative notation.) Let $g_1, h_1 \in \mathbb{G}_1$ and $g_2, h_2 \in \mathbb{G}_2$ (generators of their respective groups). The map e has the following properties:

Bilinearity: For all $a, b \in \mathbb{Z}_p^*$, the following hold:

$$e(g_1^a, g_2) = e(g_1, g_2^a) = e(g_1, g_2)^a$$
$$e(g_1 h_1, g_2) = e(g_1, g_2) \cdot e(h_1, g_2)$$
$$e(g_1, g_2 h_2) = e(g_1, g_2) \cdot e(g_1, h_2)$$

Non-degeneracy: $e(g_1, g_2) \neq 1$.
Computability: There is an efficient algorithm for computing e.

Pairings are divided into types based on the (in)equality of the utilized groups $\mathbb{G}_1, \mathbb{G}_2$ and whether there is an efficiently computable homomorphism between the groups. For our purposes, we will use a type-3 pairing: asymmetric ($\mathbb{G}_1 \neq \mathbb{G}_2$) and with no such efficiently computable homomorphism. In this case, the Decisional Diffie-Hellman (DDH) assumption is believed to hold in both $\mathbb{G}_1$ and $\mathbb{G}_2$; this is referred to as the *symmetric external Diffie-Hellman (SXDH) assumption.*

B.2 Shamir Secret Sharing

Shamir [69] introduced a scheme to share a secret among n parties such that any t parties can work together to recover the secret, but with any fewer parties the secret remains information-theoretically hidden.

Construction 1 (Shamir secret sharing [69])**.** Let p be a prime.

- $\{s_1, \ldots, s_n\} \leftarrow \mathsf{Share}(s, t, n)$: Given a secret $s \in \mathbb{Z}_p$ and $t \leq n \in \mathbb{N}$, compute a t-out-of-n sharing of s by choosing a random degree-$(t-1)$ polynomial $f(X) \in \mathbb{Z}_p[X]$ such that $f(0) = s$. For $i \in [n]$, compute $s_i := (i, f(i))$.[14]
- $\{s', \perp\} \leftarrow \mathsf{Rec}(S, t, n)$: Given some set of shares S, check if $|S| < t$. If so, output $\perp$. Otherwise, without loss of generality, let $(x_1, y_1), \ldots, (x_t, y_t)$ be the first t entries of S and output their Lagrange interpolation s' at 0:

$$s' := \sum_{i=1}^{t} y_i \prod_{j=1, j\neq i}^{t} \frac{x_j}{x_j - x_i} \pmod{p}.$$

This secret sharing scheme is *correct*, since for any secret $s \in \mathbb{Z}_p$ and values $t \leq n \in \mathbb{N}$, we have $\mathsf{Rec}(\mathsf{Share}(s, t, n), t, n) = s$.

B.3 Pedersen Commitments

Next, we recall Pedersen commitments [61], a commitment scheme which is unconditionally (information-theroetically) hiding and computationally binding (by the discrete logarithm assumption on $\mathbb{G}$). In our construction we will instantiate the scheme over $\mathbb{G}_1$, so we let $\mathbb{G} = \mathbb{G}_1$ in the description below.

Construction 2 (Pedersen commitment scheme [61])**.** Let $\mathbb{G}_1$ be a group of order p and g_1, h_1 be generators of $\mathbb{G}_1$. The following is a commitment scheme for elements $x \in \mathbb{Z}_p$.

- $(\mathsf{com}, \mathsf{decom}) \leftarrow \mathsf{Com}(x)$: Sample $r \leftarrow\!\!\$\ \mathbb{Z}_p$ and return $\mathsf{com} := g_1^x h_1^r$ and decommitment information (x, r).
- $(x, r) \leftarrow \mathsf{Open}(\mathsf{com}, \mathsf{decom})$: To open com, directly output $\mathsf{decom} = (x, r)$.
- $\{0, 1\} \leftarrow \mathsf{Verify}(\mathsf{com}, x, r)$: To confirm the opening of com to x, it suffices to check that $\mathsf{com} = g_1^x h_1^r$.

One can prove knowledge of the opening of a Pedersen commitment using a Sigma protocol due to Okamoto [59], which we present in Protocol 2 and refer to as Π_{ped}. This can be made non-interactive using the Fiat-Shamir transform [33] by computing the challenge in step 2 of Prove as the hash of the transcript tr until that point, i.e., $c := h(\mathsf{tr})$. Then $\pi_{\mathsf{ped}} := (a, (t_1, t_2))$. In Verify, $c := h(tr)$ is recomputed and used in the validity check.

To achieve a non-replayable variant of the proof system, both algorithms are additionally parameterized by an unpredictable timestamp `tstamp` and the Fiat-Shamir challenge is computed as $c := h(\mathsf{tr}, \mathtt{tstamp})$. Moreover, $\mathsf{Verify}_{\mathtt{tstamp}}$ additionally checks that `tstamp` is recent enough. The definition of "recent" is left up to the deployer; in our case we use the timestamping functionality $\mathcal{F}_{\mathsf{time}}^{\ell}$ and let the client C specify some maximum age M. Then $\mathsf{Verify}_{\mathtt{tstamp}}$ will check that $\mathcal{F}_{\mathsf{time}}^{\ell}$.CheckTimestamp returns a value $\Delta \leq M$.

[14] For efficiency in practice, powers of a root of unity are used as the evaluation points.

Protocol 2: PoK of Pedersen opening (Π_{ped}) [59]

Public parameters: Group $\mathbb{G}_1$ of order p with generators g_1, h_1.

$\mathsf{Prove}(\mathsf{com}_{\mathsf{ped}}; (v, r)) \rightarrow \pi_{\mathsf{ped}}$: Given a Pedersen commitment $\mathsf{com}_{\mathsf{ped}} = g_1^v h_1^r$, a party can prove knowledge of the opening (v, r) as follows:

(1) The prover samples $s_1, s_2 \leftarrow\!\!\$\ \mathbb{Z}_p$ and sends $a := g_1^{s_1} h_1^{s_2}$ to the verifier.
(2) The verifier sends back a uniform challenge $c \leftarrow\!\!\$\ \mathbb{Z}_p$.
(3) The prover computes $t_1 := s_1 + vc$ and $t_2 := s_2 + rc$ and sends both values to the verifier.

The proof is defined as $\pi_{\mathsf{ped}} := (a, c, (t_1, t_2))$.

$\mathsf{Verify}(\mathsf{com}_{\mathsf{ped}}, \pi_{\mathsf{ped}}) \rightarrow \{0, 1\}$: Given the commitment $\mathsf{com}_{\mathsf{ped}}$ and a proof π_{ped}, the verifier parses $\pi_{\mathsf{ped}} := (a, c, (t_1, t_2))$ and outputs 1 iff $a \cdot \mathsf{com}_{\mathsf{ped}}^c = g_1^{t_1} h_1^{t_2}$.

C PoK of Encrypted Key Share

Here we present a formal description of our proof system Π_{EKS} used for the hot proofs of remembrance.

Protocol 3: PoK of Encrypted Key Share (Π_{EKS})

Public parameters: $\mathbb{G}_1, \mathbb{G}_2, g_1, h_1, g_2, \mathsf{crs}$ as defined in Table 2.

$\mathsf{Prove}((\mathsf{crs}, \mathsf{com}, i); (sk_i^{\mathsf{hot}}, \pi_i)) \rightarrow \pi_i^{\mathsf{hot}}$: Given crs, a KZG commitment com to the current polynomial $\tilde{f}(X)$, and its index i, a hot party uses its key share $sk_i^{\mathsf{hot}} = \tilde{f}(i)$ and corresponding opening proof π_i to compute a ZKPoK of sk_i^{hot} as follows:

(1) Sample $r \leftarrow\!\!\$\ \mathbb{Z}_p$, let $\mathsf{com}_{\mathsf{ped}} := g_1^{sk_i^{\mathsf{hot}}} h_1^r$, and compute $\pi_{\mathsf{ped}} \leftarrow \Pi_{\mathsf{ped}}.\mathsf{Prove}(\mathsf{com}_{\mathsf{ped}}; (sk_i^{\mathsf{hot}}, r))$.
(2) Sample $s \leftarrow\!\!\$\ \mathbb{Z}_p$ and let $s_i(X) := -r - s(X - i)$. Compute $\overline{\pi_i} := \pi_i h_1^s$ and $S := g_2^{s_i(\tau)}$ using crs.
(3) Output $\pi_i^{\mathsf{hot}} := (\mathsf{com}_{\mathsf{ped}}, \pi_{\mathsf{ped}}, \overline{\pi_i}, S)$.

$\mathsf{Verify}((\mathsf{crs}, \mathsf{com}, i), \pi_i^{\mathsf{hot}}) \rightarrow \{0, 1\}$: Given crs, a KZG commitment com, and a party index i, verify the hot proof $\pi_i^{\mathsf{hot}} = (\mathsf{com}_{\mathsf{ped}}, \pi_{\mathsf{ped}}, \overline{\pi_i}, S)$ by outputting 1 iff the following hold:

$$\Pi_{\mathsf{ped}}.\mathsf{Verify}(\mathsf{com}_{\mathsf{ped}}, \pi_{\mathsf{ped}}) = 1$$
$$e(\mathsf{com}/\mathsf{com}_{\mathsf{ped}}, g_2) = e(\overline{\pi_i}, g_2^{\tau}/g_2^{i}) \cdot e(h_1, S).$$

In practice, we use a *non-interactive* and *non-replayable* variant of both Π_{EKS} and the cold proof system Π_{DL} via the Fiat-Shamir transform with an unpredictable timestamp; see Appendix B.3.

D Proactive Refresh

The proactive refresh of hot shares once per epoch means that Throback achieves a "hot-only" version of proactive security out of the box. This guarantee is implicit in the ideal functionality $\mathcal{F}_{\mathsf{HC}}$ realized by Throback; see the security proof, which includes proactive refreshes, in Appendix H. Since UC definitions are notoriously inscrutable, however, we give an intuitive argument for our hot-only proactive security guarantee below.

Remember that, as long as the underlying secret sk and the cold decryption keys dk_i are unknown, the hot shares are uniformly distributed due to the encryption masks, i.e., they lie on a degree-$(n-1)$ polynomial. Thus, recovering any number of the hot shares—even all n of them—reveals no useful information. Instead, an attacker must unmask/decrypt a hot share to learn any information linked to the secret (namely, the plaintext share sk_i). By the security of the encryption scheme, this requires the corresponding cold decryption key dk_i (because these keys are never refreshed—more on that below—a leak at any point in time, before or after the corresponding hot leak, is sufficient).

Hence, learning a share sk_i requires corrupting both the cold party (during any epoch) and the hot party in the pair. However, proactive refresh means the hot share, and by consequence the value sk_i, is being updated every epoch. Thus, the t hot shares obtained by the attacker must all come from the *same* epoch, otherwise they will be "out of sync" due to the refreshes. (This is the same security guarantee offered by proactive refreshes in the "regular" wallet setting.)

This hot-only proactive security guarantee is already a very strong one in practice. Compromising one cold party—let alone t, even across multiple epochs—without detection is a very difficult task. Being content with only refreshing the hot shares has the benefit of allowing us to leverage existing wallet solutions: our cold parties essentially just store independently-generated BLS signing keys, allowing Throback custodians to use out-of-box BLS hardware wallets (e.g. a Ledger wallet [53], or any other hardware wallet that supports storing Ethereum validator keys). Such hardware-embedded keys do not typically allow key updates, so it would be impossible and/or very difficult to do a cold wallet update in practice even if a construction supported it.

For completeness, however, we sketch how one could implement cold refreshes in Throback. A cold wallet P_i^{cold} with key dk_i could refresh its key by sampling a uniform value r and updating its key to $dk_i' := dk_i + r$. It must now publish both an updated encryption key and some information to allow its hot party to update its encrypted key share to use the new key. For the former, P_i^{cold} outputs a new key $ek_i' := ek_i^r$. For the latter, it must compute n update terms: one for every client C_j whose information is being stored by the hot party. Due to our construction's use of the hash function $\mathcal{H}$, which can be seen as a random oracle, this dependence of n is unavoidable. Specifically, the jth update value is $\delta_{i,j}^{\text{cold}} := \mathcal{H}(vk_j{}^{dk_i'}) - \mathcal{H}(vk_j{}^{dk_i})$. The hot party can now update each of its encrypted shares $sk_{i,j}^{\text{hot}}$ as $sk_{i,j}^{\text{hot}} + \delta_{i,j}^{\text{cold}} = (sk_{j,i} + \mathcal{H}(vk_j{}^{dk_i})) + (\mathcal{H}(vk_j{}^{dk_i'}) - \mathcal{H}(vk_j{}^{dk_i})) = sk_{j,i} + \mathcal{H}(vk_j{}^{dk_i'}) = \mathsf{Enc}(ek_i', sk_{j,i})$.

To sketch security against corrupt hot parties, note that even given dk_i, ek_i, ek_i', the value of dk_i' is indistinguishable from uniform by the discrete logarithm assumption, and thus so is each update value $\delta_{i,j}^{\text{cold}}$ received by a hot party. We leave more thorough modeling of a cold refresh protocol and the question of how to detect malicious cold refreshes to future work.

E Security Proof of $\mathcal{H}$

Lemma 2. Consider a group $\mathbb{G}$ of prime order p and $\ell = \lceil \log p \rceil$. The output of the function $\mathcal{H} : \mathbb{G}^2 \times \mathbb{Z}_p^{2\ell} \to \mathbb{Z}_p$ defined in Section 4.1 when evaluated on uniformly sampled $x_1, x_2 \in \mathbb{G}$ has statistical distance at most $\frac{1}{2\sqrt{p}}$ from uniform.

Proof. We first argue that $\mathcal{H}$ is a $\left(-log\left(\frac{1}{p^2}\right), \frac{1}{2\sqrt{p}}\right)$ extractor for the parameter set we have chosen. Note that input of our function $\mathcal{H}$ is two uniformly sampled group elements in $\mathbb{G}$, so if we treat this as a random variable X, we have $H_\infty(X)) = -log\left(\frac{1}{p^2}\right)$, since p is the order of our group elements.

In addition, if we represent $\mathcal{H}$ using the language of our extractor definition, we also have $\mathcal{V} = p$ and $\ell = \lceil log(p) \rceil$ because we are outputting a single finite field element. Therefore our hash function is valid extractor if $-log\left(\frac{1}{p^2}\right) \geq \log(p) + 2log\left(\frac{1}{\epsilon}\right) - 2$, or if $log(p) \geq log\left(\frac{1}{4\epsilon^2}\right)$. Setting $\epsilon = \frac{1}{2\sqrt{p}}$ gives us the desired equality.

Furthermore, we know that $\mathcal{H}$ is a universal hash function, because performing a subset sum operation using random field elements is a universal hash function (subset sums over integers modulo primes are universal).

Thus, by the leftover hash lemma, this means our extractor (and thus hash function) has statistical distance $\frac{1}{2\sqrt{p}}$ from uniform. □

F Universal Composability (UC) Framework

In the universal composability (UC) framework [11], the security requirements of a protocol are defined via an *ideal functionality* which is executed by a trusted party. To prove that a protocol *UC-realizes* a given ideal functionality, we show that the execution of this protocol (in the real or hybrid world) can be *emulated* in the ideal world, where in both worlds there is an additional adversary $\mathcal{E}$ (the environment) which models arbitrary concurrent protocol executions. Specifically, we show that for any adversary $\mathcal{A}$ attacking the protocol execution in the real world (by controlling communication channels and corrupting parties involved in the protocol execution), there exists an adversary $\mathcal{S}$ (the simulator) in the ideal world, interacting with the ideal functionality, who can produce a protocol execution which no environment $\mathcal{E}$ can distinguish from the real-world execution.

Below we describe the UC framework as it is presented in [14]. All parties are represented as probabilistic interactive Turing machines (ITMs) with input, output, and ingoing/outgoing communication tapes. For simplicity, we assume that all communication is authenticated, so an adversary can only delay but not forge or modify messages from parties involved in the protocol. Therefore, the order of message delivery is also not guaranteed (asynchronous communication). We consider a PPT malicious, adaptive adversary

who can corrupt or tamper with parties at any point during the protocol execution.

The execution in both worlds consists of a series of sequential party activations. Only one party can be activated at a time (by writing a message on its input tape). In the real world, the execution of a protocol Π occurs among parties $P_1, \ldots, P_n$ with adversary $\mathcal{A}$ and environment $\mathcal{E}$. In the ideal world, interaction takes place between dummy parties $\tilde{P}_1, \ldots, \tilde{P}_n$ communicating with the ideal functionality $\mathcal{F}$, with the adversary (simulator) $\mathcal{S}$ and environment $\mathcal{E}$. Every copy of $\mathcal{F}$ is identified by a unique session identifier `sid`.

In both the real and ideal worlds, the environment is activated first and activates either the adversary ($\mathcal{A}$ resp. $\mathcal{S}$) or an uncorrupted (dummy) party by writing on its input tape. If $\mathcal{A}$ (resp. $\mathcal{S}$) is activated, it can take an action or return control to $\mathcal{E}$. After a (dummy) party (or $\mathcal{F}$) is activated, control returns to $\mathcal{E}$. The protocol execution ends when $\mathcal{E}$ completes an activation without writing on the input tape of another party.

Messages to and from the ideal functionalities consist of two pieces: a *header* and the *contents*. The header normally consists of a session identifier `sid`, a description of the action the functionality should take/has taken, and the sender/recipient. In this paper, we will use the convention of putting the message contents in parentheses so it is clear where the (public) header ends and the (private) contents begin, e.g., $(\texttt{sid}, \text{Action}, P_{\text{sender}}, (\text{contents}))$ or $(\texttt{sid}, \text{ActionDone}, P_{\text{recipient}}, \text{publicvars}, (\text{contents}))$.

We denote with $\text{REAL}_{\Pi,\mathcal{A},\mathcal{E}}(\lambda, x)$ the random variable describing the output of the real-world execution of Π with security parameter λ and input x in the presence of adversary $\mathcal{A}$ and environment $\mathcal{E}$. We write the corresponding distribution ensemble as $\{\text{REAL}_{\Pi,\mathcal{A},\mathcal{E}}(\lambda, x)\}_{\lambda\in\mathbb{N},x\in\{0,1\}^*}$. The output of the ideal-world interaction with ideal functionality $\mathcal{F}$, adversary (simulator) $\mathcal{S}$, and environment $\mathcal{E}$ is represented by the random variable $\text{IDEAL}_{\mathcal{F},\mathcal{S},\mathcal{E}}(\lambda, x)$ and corresponding distribution ensemble $\{\text{IDEAL}_{\mathcal{F},\mathcal{S},\mathcal{E}}(\lambda, x)\}_{\lambda\in\mathbb{N},x\in\{0,1\}^*}$.

The actions each party can take are summarized below:

- Environment $\mathcal{E}$: **read** output tapes of the adversary ($\mathcal{A}$ or $\mathcal{S}$) and any uncorrupted (dummy) parties; then **write** on the input tape of one party (the adversary $\mathcal{A}$ or $\mathcal{S}$ or any uncorrupted (dummy) parties).
- Adversary $\mathcal{A}$: **read** its own tapes and the outgoing communication tapes of all parties; then **deliver** a pending message to party by writing it on the recipient's ingoing communication tape *or* **corrupt** a party (which becomes inactive: its tapes are given to $\mathcal{A}$ and $\mathcal{A}$ controls its actions from this point on, and $\mathcal{E}$ is notified of the corruption).
- Real-world party P_i: only follows its code (potentially writing to its output tape or sending messages via its outgoing communication tape).
- Dummy party $\tilde{P}_i$: acts only as a simple relay with the ideal functionality $\mathcal{F}$, copying inputs from its input tape to its outgoing communication tape (to $\mathcal{F}$) and any messages received on its ingoing communication tape (from $\mathcal{F}$) to its output tape.
- Adversary $\mathcal{S}$: **read** its own input tape and the public headers (see below) of the messages on $\mathcal{F}$'s and dummy parties' outgoing communication tapes; then **deliver** a message to $\mathcal{F}$ from a dummy party or vice versa by copying it from the sender's outgoing communication tape to the recipient's incoming communication tape *or* **send** its own message to $\mathcal{F}$ by writing on the latter's incoming communication tape *or* **corrupt** a dummy party (which becomes inactive: its tapes are given to $\mathcal{S}$ and $\mathcal{S}$ controls its actions from this point on, and $\mathcal{E}$ and $\mathcal{F}$ are notified of the corruption).
- Ideal functionality $\mathcal{F}$: **read** incoming communication tape; then **send** any messages specified by its definition to the dummy parties and/or adversary $\mathcal{S}$ by writing to its outgoing communication tape.

Definition 8. We say a protocol Π *UC-realizes* an ideal functionality $\mathcal{F}$ if for any PPT adversary $\mathcal{A}$, there exists a simulator $\mathcal{S}$ such that for any environment $\mathcal{E}$, the distribution ensembles $\{\text{REAL}_{\Pi,\mathcal{A},\mathcal{E}}(\lambda, x)\}_{\lambda\in\mathbb{N},x\in\{0,1\}^*}$ and $\{\text{IDEAL}_{\mathcal{F},\mathcal{S},\mathcal{E}}(\lambda, x)\}_{\lambda\in\mathbb{N},x\in\{0,1\}^*}$ are computationally indistinguishable.

G Additional ideal functionalities

We give the full ideal functionalities $\mathcal{F}_{\text{ESS}}$, $\mathcal{F}_{\text{PK}}$, and $\mathcal{F}^{\ell}_{\text{time}}$ in Functionalities 1 to 3. The first two functionalities are used by the registration and refresh protocols of Throback for key and (encrypted) secret-sharing, respectively, while the latter is used in the proofs of remembrance. Thus, the corresponding interfaces of $\mathcal{F}_{\text{HC}}$, the functionality realized by Throback, repeat large parts of $\mathcal{F}_{\text{PK}}$ and $\mathcal{F}_{\text{ESS}}$: ColdRegister runs the same code as $\mathcal{F}_{\text{PK}}$.PKSetup; ClientRegister as $\mathcal{F}_{\text{ESS}}$.SSSetup; and ShareRefresh repeats $\mathcal{F}_{\text{ESS}}$'s ZeroSetup. We use underlining (for $\mathcal{F}_{\text{PK}}$) and gray highlights (for $\mathcal{F}_{\text{ESS}}$) to show this repetition in $\mathcal{F}_{\text{HC}}$. Because the protocol (interacting with these functionalities) and $\mathcal{F}_{\text{HC}}$ (repeating their code) are identical in these interfaces, the simulation is very straightforward: we can simply pass messages along between the (dummy) parties and the $\mathcal{F}_{\text{HC}}$ functionality.

The PKSetup interface of $\mathcal{F}_{\text{PK}}$ does not allow re-registrations. This is because it is used in our construction for ColdRegister, which cannot accommodate cold key rotations, i.e., a cold party changing its keypair after a client has already used it to back up some secrets. Throback could easily be extended to implement this naively by asking clients to recover their secrets and create a new backup when a cold party changes its keys; we leave a more elegant realization of cold key rotations without reconstruction to future work.

On the other hand, we do in principle allow re-registrations in SSSetup (resp. ClientRegister), since a single client may have multiple secrets it wishes to back up, potentially even with different sets of custodians. Our functionality stores all of these secrets for the client. To avoid inhibiting readability, however, we have written our functionalities and proof as though each client only registers once; these can easily be generalized to client re-registrations by using the corresponding verification key vk to disambiguate between (potentially multiple) hot key shares for the same client.[15]

H Security Analysis of Throback

We prove our scheme secure in the universal composability framework[16], which we summarize in Appendix F. As in some previous

[15]This includes sending (C, vk) instead of only C in calls to the recovery signing and proof of remembrance interfaces of $\mathcal{F}_{\text{HC}}$ in order to identify the secret to use.
[16]Specifically, we use the version presented in [14].

Functionality 1: $\mathcal{F}_{\mathsf{PK}}$ (Extended PKI)

Public parameters: $(\mathbb{G}_2, g_2, p)$ as defined in Table 2.
Internal variables: $L = (P, ek, dk)$ / parties and their keypairs

On input (**sid**, PKSetup, P)**:**
(1) If $\exists(P, *, *) \in L$, let $ek, dk := \perp$.
(2) Otherwise, sample $dk := (dk_1, dk_2) \leftarrow\$ \mathbb{Z}_p^2$, set $ek := (g_2^{dk_1}, g_2^{dk_2})$, and add (P, ek, dk) to L.
(3) Output $(\mathtt{sid}, \mathsf{PKSetupResult}, P, (ek, dk))$.

On input (**sid**, PKRecover, P, (Q))**:**
(1) Retrieve $(Q, ek, *, *, *) \in L$; if there is no such entry, let $ek := \perp$.
(2) Output $(\mathtt{sid}, \mathsf{PK}, P, (Q, ek))$.

Functionality 2: $\mathcal{F}_{\mathsf{time}}^{\ell}$ (Unpredictable timestamps)

Internal variables: $D = (\mathtt{tstamp}_1, \mathtt{tstamp}_2, \dots)$ / an ordered list of issued timestamps, initialized with a single entry

On input (**sid**, IncrementTime, $\mathcal{E}$)**:**
(1) Sample $\mathtt{tstamp} \leftarrow\$ \{0,1\}^{\ell}$ and add it to the list D. Ignore requests sent by any party other than the environment.
(2) Output $(\mathtt{sid}, \mathsf{IncrementOk}, \mathcal{E})$.

On input (**sid**, GetTimestamp, P)**:**
(1) Retrieve the latest value $\mathtt{tstamp}_n$ in D.
(2) Output $(\mathtt{sid}, \mathsf{Timestamp}, P, \mathtt{tstamp})$.

On input (**sid**, CheckTimestamp, P, (**tstamp**))**:**
(1) Find the (highest) index i of tstamp in D and let $\Delta := |D| - i$.
(2) Output $(\mathtt{sid}, \mathsf{TimestampAge}, P, (\Delta))$.

work [46, 55], our functionalities contain some cryptographic operations specific to our construction. While this formulation is less general, it renders the analysis more straightforward and suffices for our purposes. In particular, our ideal functionality captures the cryptographic operations of threshold BLS signatures. This allows us to precisely state the security requirements we aim to achieve and reduce the UC-security of protocols to the security of the BLS threshold signature scheme. A significant benefit of the same is that we do not need to reprove the security of threshold BLS signatures. Finally, another benefit of proving security in the UC framework is that we get security for protocols in the concurrent setting where multiple messages are being signed concurrently.

This approach is similar to some previous work on discrete-log-based signature schemes like ECDSA. For example, Lindell and Nof [55] give a threshold ECDSA protocol and prove its security by showing it realizes a (non-threshold) ECDSA functionality. In another work [46], Katz defines and realizes a distributed key generation functionality that outputs threshold keys to the participants. The advantage of this approach is that none of these proofs require reproving the security of the standard signature scheme (i.e., ECDSA or threshold BLS).

We now present our BLS auditable hot-cold threshold backup functionality $\mathcal{F}_{\mathsf{HC}}$. Because Throback (Protocol 1) consists of four different sub-protocols (two types of registration, signing, two proofs of remembrance, and proactive refresh), our UC functionality has six interfaces corresponding to these sub-protocols. It also has three additional types of interfaces (Leak, Corrupt, and Delete) which capture the adversarial model. We also explicitly disallow corruption/leakage of more than t hot-cold *pairs* simultaneously (i.e., within the same epoch).

Observe that the functionality offered is close to that of standard threshold BLS signatures. In particular, the functionality has no concept of hot and cold shares (or partial signatures). Specifically, it makes no distinction between hot and cold parties: from the point of view of the functionality, clients register (i.e., run key generation) with a set of n *monolithic* custodians $I_1, \dots, I_n$; each threshold signature is generated by a single party I_i; and proactive refreshes act on the key material sk_i stored by these P_i. Since our protocol realizes this functionality, it means that the changes we made to accommodate a hot/cold split do not leak any additional information about the secrets nor open up any new attacks. Therefore, the security of a protocol implementing $\mathcal{F}_{\mathsf{HC}}$ follows from security of threshold BLS.

Adversarial model. We assume that the real-execution adversary $\mathcal{A}$ can corrupt/leak secrets from from hot parties (adaptively) and from cold parties (selectively[17]). The adversary can also direct a hot/cold party to delete its individual hot or cold secrets.

Theorem 1 (Throback security)**.** Throback (Protocol 1) UC-realizes $\mathcal{F}_{\mathsf{HC}}$ in the $\mathcal{F}_{\mathsf{ESS}}$,$\mathcal{F}_{\mathsf{PK}}$,$\mathcal{F}_{\mathsf{time}}^{\ell}$-hybrid model, assuming security of KZG commitments, and knowledge soundness and zero-knowledge of Π_{DL} and Π_{EKS}.

Before giving the proof, we give an overview of our approach. At a high level, we need to simulate the real protocol execution, including the outputs and any information leaked to a real-world adversary $\mathcal{A}$. The crux of the proof is the simulation of the threshold signing and the proof of remembrance interfaces. For signing, the simulator must simulate the partial (hot/cold) signatures to its internal adversary, and also provide values to the functionality that ensure it outputs a consistent threshold signature σ_i. We rely on the homomorphism of BLS signatures and our share encryption for this. Similarly, for proofs of remembrance, the simulator must decide whether a corrupted party should respond to a proof request honestly. Here we rely on knowledge soundness of Π_{DL} and Π_{EKS}.

Proof of Theorem 1. Let $\mathcal{A}$ be the adversary interacting with the parties (namely, $P_1^{\mathsf{hot}}, P_1^{\mathsf{cold}}, \dots, P_n^{\mathsf{hot}}, P_n^{\mathsf{cold}}, C$) running Full Construction. We will construct a simulator $\mathcal{S}$ running in the ideal world against $\mathcal{F}_{\mathsf{HC}}$ so that no environment $\mathcal{E}$ can distinguish an execution of the ideal-world interaction from the real protocol. $\mathcal{S}$

[17] by this we mean that $\mathcal{A}$ must commit to a set C_{cold} of size $\leq t$ of cold parties it can corrupt before the protocol execution begins

Functionality 3: $\mathcal{F}_{\mathsf{ESS}}$ (Encrypted secret sharing)

Public parameters: $(\mathbb{G}_2, g_2, p)$, crs, $\mathcal{H}$ as defined in Table 2.
Internal variables: $D = (C, \mathcal{P}, t, y, \{y_i\}_{i\in[n]}, \mathsf{com}, \mathsf{epoch})$ / list of clients

On input (**sid**, SSSetup, C, $(t, \mathcal{P}, \{ek_i\}_{i\in[n]})$)**:**
/ $\mathcal{P} = \{P_1, \ldots, P_n\}$ is a set of parties, $ek_i \in \mathbb{G}_2^2$ their encryption keys, and $t \le n = |\mathcal{P}|$
(1) Sample $x \leftarrow\!\$\ \mathbb{Z}_p \setminus \{0\}$. Let $y := g_2^x$.
(2) Generate t-of-n Shamir shares of x as $x_1, \ldots, x_n$. Let $y_i := g_2^{x_i} \forall i \in [n]$.
(3) Interpolate the degree-n polynomial $\tilde{f}$ such that $\tilde{f}(i) = \mathcal{H}(ek_i^x) + x_i \forall i \in [n]$. Compute com $\leftarrow\!\$\ \mathsf{KZG.Com}(\mathsf{crs}, \tilde{f})$.
(4) For each $i \in [n]$, compute $(\tilde{x}_i, \pi_i) \leftarrow \mathsf{KZG.Open}(\mathsf{crs}, \tilde{f}, i)$ and output $(\mathtt{sid}, \mathsf{SecretShare}, P_i, (C, i, \tilde{x}_i, \pi_i, y_i))$.
(5) Finally, add $(C, \mathcal{P}, t, y, \{y_i\}_{i\in[n]}, \mathsf{com}, \mathsf{epoch} := 0)$ to D. Output $(\mathtt{sid}, \mathsf{SSSetupDone}, C, y)$.

On input (**sid**, ShareRefresh, C)**:**
(1) Retrieve $(C, \mathcal{P}, t, y, \{y_i\}_{i\in[n]}, \mathsf{com}, \mathsf{epoch}) \in D$ for the maximum value of epoch.
(2) Generate t-of-n Shamir shares of 0 as $x_1, \ldots x_n \in \mathbb{Z}_p$ using polynomial z. Let $y_i' := y_i g_2^{x_i} \forall i \in [n]$.
(3) Compute $\mathsf{com}_0 \leftarrow \mathsf{KZG.Com}(\mathsf{crs}, z)$.
(4) For each $i \in [n]$, if C is corrupt, send $(\mathtt{sid}, \mathsf{ZeroShareRequest}, \mathcal{A}, (P_i))$ and wait for response b_i^*; else, set $b_i^* := 1$. If $b_i^* = 1$, compute $(\delta_i, \zeta_i) \leftarrow \mathsf{KZG.Open}(\mathsf{crs}, z, i)$; otherwise set $(\delta_i, \zeta_i) := (\perp, \perp)$. Output $(\mathtt{sid}, \mathsf{ShareRefreshResult}, P_i, (C, \delta_i, \zeta_i))$.
(5) Finally, if C is corrupt, send $(\mathtt{sid}, \mathsf{ZeroSetupRequest}, \mathcal{A})$ and await response b^*; else, set $b^* := 1$. If $b^* = 1$, add $(C, \mathcal{P}, t, y, \{y_i'\}_{i\in[n]}, \mathsf{com} \cdot \mathsf{com}_0, \mathsf{epoch}{+}{+})$ to D; otherwise, set $\mathsf{com}_0 = \perp$ and $y_i' = \perp \forall i \in [n]$. Output $(\mathtt{sid}, \mathsf{ShareRefreshResult}, C, \{y_i'\}_{i\in[n]}, \mathsf{com}_0)$.

On input (**sid**, AuxRecover, P_i, (C))**:**
– Retrieve $(C, *, *, y, *, \mathsf{com}, \mathsf{epoch}) \in D$ for the maximum epoch and output $(\mathtt{sid}, \mathsf{AuxInfo}, P_i, (C, y, \mathsf{com}))$.

will interact with $\mathcal{F}_{\mathsf{HC}}$, $\mathcal{E}$, and invoke a copy of $\mathcal{A}$ to run a simulated interaction of the protocol (we call this simulated interaction between $\mathcal{A}$, $\mathcal{E}$, and the parties the *internal interaction* to distinguish it from the *external interaction* between $\mathcal{S}$, $\mathcal{E}$, and $\mathcal{F}_{\mathsf{HC}}$).

Each protocol/algorithm begins with a party receiving some input. In the ideal world, this input is received by some dummy party, who immediately copies it to its outgoing communication tape where $\mathcal{S}$ can read it (public header only) and potentially deliver it to the ideal functionality $\mathcal{F}_{\mathsf{HC}}$. (Recall that for simplicity we assume authenticated communication, so $\mathcal{S}$ cannot modify the messages it delivers to/from $\mathcal{F}_{\mathsf{HC}}$.) To complete the protocol, $\mathcal{S}$ will deliver the response of $\mathcal{F}_{\mathsf{HC}}$ to the dummy party, who copies it to its output tape (which is visible to $\mathcal{E}$).

Simulator $\mathcal{S}$: At system setup, $\mathcal{S}$ generates system parameters for the internal execution, including the KZG $\widetilde{\mathsf{crs}}$. These are generated honestly, except $\mathcal{S}$ keeps the trapdoor τ of $\widetilde{\mathsf{crs}}$, which it will use to simulate hot opening proofs later.

Message delivery. $\mathcal{S}$ waits to deliver any messages until $\mathcal{A}$ delivers the corresponding message in the internal interaction.

Custodian (Cold Party) Registration. $\mathcal{S}$ acts as $\mathcal{F}_{\mathsf{PK}}$, sampling a cold keypair $\widetilde{dk_i} \leftarrow\!\$\ \mathbb{Z}_p^2$, $\widetilde{ek_i} := g_2^{\widetilde{dk_i}}$. It saves both $\widetilde{ek_i}, \widetilde{dk_i}$ in a list entry corresponding to P_i^{cold} and outputs $(\mathtt{sid}, \mathsf{ColdRegistered}, P_i^{\mathsf{cold}}, \widetilde{ek_i})$. If the cold party is corrupt, $\mathcal{S}$ instead first sends $\widetilde{ek_i}$ to $\mathcal{A}$ in the internal execution, and uses as its output the value of ek_i^* returned by $\mathcal{A}$ (controlling P_i^{cold}) in the internal execution.

Backup Creation. $\mathcal{S}$ plays the role of $\mathcal{F}_{\mathsf{PK}}$ and $\mathcal{F}_{\mathsf{ESS}}$ in the internal execution. In the simulation of the latter, $\mathcal{S}$ does not compute any of the values which depend on sk; instead, it uses $vk, \{vk_i\}_{i\in[n]}$ output by $\mathcal{F}_{\mathsf{HC}}$ in the ideal execution and sample $\widetilde{\mathsf{com}} \leftarrow\!\$\ \mathbb{G}_1$. It stores these values for future use.

If C and/or any P_i^{hot} is corrupt, $\mathcal{S}$ outputs on their behalf the same value (vk^* or b_i^*) that $\mathcal{A}$ outputs in the internal execution.

Recovery (Signing). For each custodian, if the hot party is corrupt, $\mathcal{S}$ has already fixed a value $\widetilde{sk_i^{\mathsf{hot}}}$. In this case, the functionality will ask $\mathcal{S}$ for a value σ_i^*, to which it responds using the same value output by $\mathcal{A}$ on behalf of P_i^{hot} in the internal execution. If the cold party is honest, the functionality's request includes $\sigma_i = H(m)^{sk_i}$, and in this case $\mathcal{S}$ must also send a cold signature $\widetilde{\sigma_i^{\mathsf{cold}}}$ to $\mathcal{A}$ in the internal execution. It can compute this as $H(m)^{\mathcal{H}(vk_i^{\widetilde{dk_i}})}$ if $i \in \mathcal{C}_{\mathsf{cold}}$ and $H(m)^{\widetilde{sk_i^{\mathsf{hot}}}}/\sigma_i$ for $(x_i, r) \in \mathcal{L}_{\mathcal{H}}$ otherwise. If only the cold party is corrupt, $\mathcal{S}$ has only fixed a value $\widetilde{sk_i^{\mathsf{cold}}}$, and will use this along with the cold signature sent by $\mathcal{A}$ in the internal execution to send $R^* := H(m)^{\widetilde{sk_i^{\mathsf{cold}}}}/\sigma_i^{\mathsf{cold}*}$ to the functionality when prompted. If neither party in the pair is corrupt, $\mathcal{S}$ does not exchange any messages with $\mathcal{F}_{\mathsf{HC}}$.

Proofs of Remembrance. If the hot (resp. cold) party in question is corrupted, $\mathcal{S}$ must provide a bit b^* to $\mathcal{F}_{\mathsf{HC}}$ in the ideal execution. To do so, it verifies the timestamp and proof sent by $\mathcal{A}$ on behalf

Functionality 4: $\mathcal{F}_{\mathsf{HC}}$ (Hot-cold BLS)

Public parameters:
$(\mathbb{G}_2, g_2, p)$, crs, $H, \mathcal{H}$ as defined in Table 2.
Internal variables:
$D = \{(C, \mathcal{I}, t, vk, \{vk_i\}_{i\in[n]}, \mathsf{epoch})\}$ / registered clients (users)
$L = \{(I_i, C_j, sk_i, \mathsf{epoch}, \mathtt{leaked}, \mathtt{del}_{\mathsf{hot}}, \mathtt{del}_{\mathsf{cold}})\}$ / custodian secrets (per-client)
$S = \{(I_i, \mathtt{corrupted}_{\mathsf{hot}}, \mathtt{corrupted}_{\mathsf{cold}})\}$ / registered custodians & their states (corruptions)

On input $(\mathtt{sid}, \mathsf{ClientRegister}, C, (t, \mathcal{I}))$: / $\mathcal{I} = \{(P_i^{\mathsf{hot}}, P_i^{\mathsf{cold}})\}_{i\in[n]}$ is a set of institutions, $t \le |\mathcal{I}|$

(1) For each $i \in [n]$, check that $\exists (I_i, *, *) \in S$. If so, send $(\mathtt{sid}, \mathsf{PK}, C, (I_i, \top))$ to C, otherwise send $(\mathtt{sid}, \mathsf{PK}, C, (I_i, \bot))$. Wait for C's confirmation before proceeding. If any institution was missing, abort and output $(\mathtt{sid}, \mathsf{ClientRegistered}, C, \bot)$.
(2) Upon receiving $(\mathtt{sid}, \mathsf{SSSetup}, C)$, continue by sampling $sk \leftarrow\!\$\ \mathbb{Z}_p \setminus \{0\}$ and letting $vk := g_2^{sk}$.
(3) Generate t-of-n Shamir shares of sk as $sk_1, \ldots sk_n$. Let $vk_i := g_2^{sk_i}\ \forall i \in [n]$.
(4) Add $(C, \mathcal{I}, t, vk, \{vk_i\}_{i\in[n]}, \mathsf{epoch} := 0)$ to D and, for each $i \in [n]$, add $(I_i, C, sk_i, \mathsf{epoch} := 0, 0, 0, 0)$ to L.
(5) Output $(\mathtt{sid}, \mathsf{ClientRegistered}, C, vk)$ and $(\mathtt{sid}, \mathsf{ClientRegistered}, I_i, (C, b_i = 1))\ \forall i \in [n]$.

On input $(\mathtt{sid}, \mathsf{TSign}, C, (\mathcal{I}, m))$: / Recovery (threshold Signing)

(1) Retrieve $(C, \mathcal{I}, t, vk, \{vk_i\}_{i\in[n]}, *, \mathsf{epoch}) \in D$. If there is no such entry, abort.
(2) For each $I_i \in \mathcal{I}$, retrieve $(I_i, \mathtt{corrupted}_{\mathsf{hot}}, \mathtt{corrupted}_{\mathsf{cold}}) \in S$ and $(I_i, C, sk_i, \cdot, \cdot) \in L$ and set σ_i as follows:
   - If $\mathtt{corrupted}_{\mathsf{hot}} \wedge \mathtt{corrupted}_{\mathsf{cold}}$, send $(\mathtt{sid}, \mathsf{TSignRequest}, \mathcal{A}, (C, I_i, m))$ to $\mathcal{A}$, and wait for response σ_i^*. Let $\sigma_i := \sigma_i^*$.
   - If $\mathtt{corrupted}_{\mathsf{hot}} \wedge \neg\mathtt{corrupted}_{\mathsf{cold}}$, send $(\mathtt{sid}, \mathsf{TSignRequest}, \mathcal{A}, (C, I_i, m, H(m)^{sk_i}))$ to $\mathcal{A}$, and wait for response σ_i^*. Let $\sigma_i := \sigma_i^*$.
   - Instead, if $\neg\mathtt{corrupted}_{\mathsf{hot}} \wedge \mathtt{corrupted}_{\mathsf{cold}}$, send $(\mathtt{sid}, \mathsf{CSignRequest}, \mathcal{A}, (P_i^{\mathsf{cold}}, C, vk, m))$ to $\mathcal{A}$, and wait for response R^*. Let $\sigma_i := H(m)^{sk_i} R^*$.
   - Otherwise ($\neg\mathtt{corrupted}_{\mathsf{cold}} \wedge \neg\mathtt{corrupted}_{\mathsf{hot}}$), set $\sigma_i := H(m)^{sk_i}$.
(3) Run BLS reconstruction on $\{\sigma_i\}_{i\in[n]}$ to get σ. Output $(\mathtt{sid}, \mathsf{TSignDone}, C, (\mathcal{I}, m, \sigma))$.

On input $(\mathtt{sid}, \mathsf{ShareRefresh}, C)$: / Proactive refresh

(1) Output $\bot$ if a prior share refresh from C is still pending.[a]
(2) Retrieve $(C, \mathcal{I}, t, vk, \{vk_i\}_{i\in[n]}, \mathsf{epoch}) \in D$ for the maximum value of epoch, where $n = |\mathcal{I}|$.
(3) Generate t-of-n Shamir shares of 0 as $\delta_1, \ldots, \delta_n \in \mathbb{Z}_p$. Let $vk_i' := vk_i g_2^{\delta_i} \forall i \in [n]$.
(4) For each $i \in [n]$:
   (a) If C is corrupt, send δ_i to $\mathcal{A}$ and wait for a bit b_i^* in response; if $b_i^* = 0$, set $\delta_i, vk_i' := \bot$.
   (b) If $\delta_i \neq \bot$, retrieve $(I_i, C, sk_i, \mathsf{epoch}, \mathtt{leaked}, \mathtt{del}_{\mathsf{hot}}, \mathtt{del}_{\mathsf{cold}}) \in L$ for the maximum value of epoch and $(I_i, \mathtt{corrupted}_{\mathsf{hot}}, \mathtt{corrupted}_{\mathsf{cold}}) \in S$. Let $\mathtt{leaked}' := 0 \vee (\mathtt{corrupted}_{\mathsf{hot}} \wedge \mathtt{corrupted}_{\mathsf{cold}})$ and add $(I_i, C, sk_i + \delta_i, \mathsf{epoch}{+}{+}, \mathtt{leaked}', \mathtt{del}_{\mathsf{hot}}, \mathtt{del}_{\mathsf{cold}})$ to L.
   (c) Let $b_i := (\delta_i \neq \bot)$ and output $(\mathtt{sid}, \mathsf{ShareRefreshResult}, I_i, (C, b_i))$.
(5) If C is corrupt, send $\{vk_i'\}_{i\in[n]}$ to $\mathcal{A}$ and wait for a bit b^* in response; otherwise set $b^* := 1$.
(6) Finally, if $b^* = 1$, add $(C, \mathcal{I}, t, vk, \{vk_i'\}_{i\in[n]}, \mathsf{epoch}{+}{+})$ to D. Output $(\mathtt{sid}, \mathsf{ShareRefreshResult}, C, (b^*))$.

/ Adversarial Interfaces

On input $(\mathtt{sid}, \mathsf{Leak}, \mathcal{A}, (I_i))$

- For every distinct C_j such that $(I_i, C_j, *, *, *, *, *) \in L$:
  - Retrieve $(C_j, *, t, *, *, *) \in D$. If $|\{(*, C_j, *, 1, *, *) \in L\}| \ge t$, let $\ell := \bot$.
  - Otherwise, select the entry $(I_i, C_j, sk_i, \mathsf{epoch}, \mathtt{leaked}, *, *)$ in L with the maximum value of epoch, set leaked to 1, and let $\ell := sk_i$.
  - Output $(\mathtt{sid}, \mathsf{LeakResult}, \mathcal{A}, (I_i, C_j, \ell))$.

On input $(\mathtt{sid}, \mathsf{Corrupt}, \mathcal{A}, (I_i))$

- If, for any entry $(C, \mathcal{I}, t, *, *, *) \in D$, $|\{I \in \mathcal{I} : (I, 1, 1) \in S\}| \ge t$, return $\bot$. Else, retrieve $(I_i, \mathtt{corrupted}) \in D$ and set $\mathtt{corrupted} = 1$. ($\mathcal{A}$ receives I_i's tapes and $\mathcal{E}$ is notified.)

/ Additional interfaces due to hot/cold split

On input $(\mathtt{sid}, \mathsf{ColdRegister}, I_i)$**:**

(1) If $\exists(I_i, \cdot, \cdot) \in S$, let $ek_i := \bot$. Otherwise, add $(I_i, 0, 0)$ to S and sample $ek_i \leftarrow\!\!\$\ (\mathbb{G}_2)^2$.
(2) Output $(\mathtt{sid}, \mathsf{ColdRegistered}, I_i, ek_i)$.

On input $(\mathtt{sid}, \mathsf{CProof}, C, (P_i^{\mathsf{cold}}))$**:**

(1) Retrieve $(I_i, C, \cdot, \cdot, \cdot, \cdot, \mathtt{del}_{\mathsf{cold}}) \in L$ for max epoch and $(I_i, \cdot, \mathtt{corrupted}_{\mathsf{cold}}) \in S$ and set b as follows:
   – If $\mathtt{corrupted}_{\mathsf{cold}} = 1$, send $(\mathtt{sid}, \mathsf{CProofRequest}, \mathcal{A}, (C, P_i^{\mathsf{cold}}))$ to the adversary, who will send back a bit b^* to represent whether the query should be responded to honestly. Set $b := b^* \wedge \neg\mathtt{del}_{\mathsf{cold}}$.
   – Otherwise, set $b := \neg\mathtt{del}_{\mathsf{cold}}$.
(2) Output $(\mathtt{sid}, \mathsf{CProofResult}, C, (P_i^{\mathsf{cold}}, b))$.

On input $(\mathtt{sid}, \mathsf{HProof}, C, (P_i^{\mathsf{hot}}))$**:**

(1) Retrieve $(I_i, C, \cdot, \cdot, \cdot, \mathtt{del}_{\mathsf{hot}}, \cdot) \in L$ for max epoch and $(I_i, \mathtt{corrupted}_{\mathsf{hot}}, \cdot) \in S$ and set b as follows:
   – If $\mathtt{corrupted}_{\mathsf{hot}} = 1$, send $(\mathtt{sid}, \mathsf{HProofRequest}, \mathcal{A}, (C, P_i^{\mathsf{hot}}))$ to the adversary, who will send back a bit b^* to represent whether the query should be responded to honestly. Set $b := b^* \wedge \neg\mathtt{del}_{\mathsf{hot}}$.
   – Otherwise, set $b := \neg\mathtt{del}_{\mathsf{hot}}$.
(2) Output $(\mathtt{sid}, \mathsf{HProofResult}, C, (P_i^{\mathsf{hot}}, b))$.

/ Adversarial Interfaces

On input $(\mathtt{sid}, \mathsf{HDelete}, \mathcal{A}, (I_i, C_j))$

– Retrieve $(I_i, C_j, sk_i, \mathsf{epoch}, *, \mathtt{del}_{\mathsf{hot}}, *)$ in L with the maximum value of epoch. Set $sk_i = \bot$ and $\mathtt{del}_{\mathsf{hot}} = 1$.
– Output $(\mathtt{sid}, \mathsf{HDeleteDone}, \mathcal{A}, (I_i, C_j))$.

On input $(\mathtt{sid}, \mathsf{CDelete}, \mathcal{A}, (I_i, C_j))$

– Retrieve $(I_i, C_j, sk_i, \mathsf{epoch}, *, *, \mathtt{del}_{\mathsf{cold}})$ in L with the maximum value of epoch. Set $sk_i = \bot$ and $\mathtt{del}_{\mathsf{cold}} = 1$.
– Output $(\mathtt{sid}, \mathsf{CDeleteDone}, \mathcal{A}, (I_i, C_j))$.

[a] We assume share refreshes are sequential and delivery to all P_i precedes any new refreshes.

of the corrupted party in the internal execution, using the simulated public values/keys: $b^* \leftarrow \Pi_{\mathsf{DL}}.\mathsf{Verify}_{\mathtt{tstamp}^*}(\widetilde{ek}_i, \pi_i^{\mathsf{cold}^*})$ resp. $b^* \leftarrow \Pi_{\mathsf{EKS}}.\mathsf{Verify}((\widetilde{\mathsf{crs}}, \widetilde{\mathsf{com}}, i), \pi_i^{\mathsf{hot}^*})$.

Proactive Refresh. Again, if all parties are honest, $\mathcal{S}$ can simply pass messages along in the ideal world when they are delivered in the internal execution. If C is corrupted, however, it needs to provide bits $b_i^*\ \forall i \in [n]$ and b^* to $\mathcal{F}_{\mathsf{HC}}$. It lets each $b_i^* := 1$ iff $(\delta_i, \zeta_i) \neq (\bot, \bot)$ in the internal execution, and similarly $b^* := 1$ iff ucom $\neq \bot$. If $(\widetilde{sk_i^{\mathsf{hot}}}, \tilde{\pi}_i)$ have already been set, $\mathcal{S}$ also updates them to $(\widetilde{sk_i^{\mathsf{hot}}} + \delta_i, \tilde{\pi}_i \cdot \zeta_i)$.

Deletion. Whenever $\mathcal{A}$ deletes a hot/cold party's secret in the internal execution, $\mathcal{S}$ activates the corresponding adversarial interface (HDelete or CDelete) of $\mathcal{F}_{\mathsf{HC}}$.

Leakage. Recall that we allow adaptive corruptions/leakage of the hot parties but only *non-adaptive* corruption/leakage of the cold parties. Thus, $\mathcal{A}$ announces the set of cold parties C_{cold} it will corrupt upfront. When $\mathcal{A}$ corrupts a hot party P_i^{hot}, $\mathcal{S}$ first checks if $i \in C_{\mathsf{cold}}$. If so, it activates the Leak interface of $\mathcal{F}_{\mathsf{HC}}$ to get sk_i and sets $\widetilde{sk_i^{\mathsf{hot}}} := sk_i + \mathcal{H}(\widetilde{vk_i^{dk_i}})$. Otherwise, if $i \notin C_{\mathsf{cold}}$, $\mathcal{S}$ samples $\widetilde{sk_i^{\mathsf{hot}}} \leftarrow\!\!\$\ \mathbb{Z}_p$. It sends $\widetilde{sk_i^{\mathsf{hot}}}$ to $\mathcal{A}$ along with a simulated opening proof $\tilde{\pi}_i$ (constructed using the KZG trapdoor τ) that the current value of $\widetilde{\mathsf{com}}$ opens to $\widetilde{sk_i^{\mathsf{hot}}}$.

When $\mathcal{A}$ corrupts a cold party P_i^{cold} (this means $i \in C_{\mathsf{cold}}$ must hold), $\mathcal{S}$ sets $\widetilde{sk_i^{\mathsf{cold}}} := \mathcal{H}(\widetilde{vk_i^{dk_i}})$ and leaks $\widetilde{dk_i}$ to $\mathcal{A}$.

Corruptions. When the first party of a pair is corrupted, $\mathcal{S}$ activates the Corrupt interface of $\mathcal{F}_{\mathsf{HC}}$ to mark the party fully corrupt. It also changes its behavior in the other interfaces according to the values sent by the internal $\mathcal{A}$ on behalf of the corrupt hot/cold party (see above) and fixes the value of the partial share ($\widetilde{sk_i^{\mathsf{hot}}}$ or $\widetilde{sk_i^{\mathsf{cold}}}$) as described in the leakage simulation above.

We will now show via a series of hybrids that $\mathcal{E}$ cannot distinguish between the ideal execution (with $\mathcal{S}$ and $\mathcal{F}_{\mathsf{HC}}$) and a real protocol execution (with $\mathcal{A}$ and parties running the real protocol).

Hybrid H_0: This is the real protocol execution (Protocol 1).

Hybrid H_1: During system setup, do not delete the KZG trapdoor τ. In ClientRegister, instead of interpolating $\tilde{f}$ to compute com, replace it with $\widetilde{\mathsf{com}} \leftarrow\!\!\$\ \mathbb{G}_1$, and simulate the opening proofs $\{\tilde{\pi}_i\}_{i\in[n]}$ for the hot shares using τ.

Hybrid H_2: In TSign, when both parties in a pair are honest, let $\sigma_i := H(m)^{sk_i}$.

Hybrid H_3: In backup creation and recovery, when evaluating $\mathcal{H}$ on input $ek_i^{sk} = vk^{dk_i}$, if $i \notin C_{\mathsf{cold}}$ instead use input $x_i \leftarrow\!\!\$\ (\mathbb{G}_2)^2$. Let $\mathcal{X} = \{x_i\}_{i\notin C_{\mathsf{cold}}}$.

Hybrid H_4: During setup, initialize an empty list $\mathcal{L}_{\mathcal{H}}$. Whenever $\mathcal{H}$ is evaluated on an input $x \in \mathcal{X}$, instead of computing the hash

function, set the output to r where $(x, r) \in \mathcal{L}_{\mathcal{H}}$. If $\nexists(x, \cdot) \in \mathcal{L}_{\mathcal{H}}$, let $r \leftarrow\!\!\$\ \mathbb{Z}_p$ and store (x, r) in $\mathcal{L}_{\mathcal{H}}$.

Claim 1. For all PPT distinguishers $\mathcal{E}$,

$$\mathsf{EXEC}_{H_0,\mathcal{A},\mathcal{E}} \equiv \mathsf{EXEC}_{H_1,\mathcal{A},\mathcal{E}}$$

Proof. Failing to delete the trapdoor does not affect the protocol execution itself so this change is perfectly indistinguishable. Futhermore, the value com is a commitment to the polynomial which interpolates the uniformly random values sk_i^{hot} and an additional randomizer ρ. Therefore, even if all the openings sk_i^{hot} are eventually revealed through hot-party compromises, com is a uniform value just like $\widetilde{\text{com}}$ and thus identically distributed. Lastly, simulated KZG opening proofs are computed as $(\widetilde{\text{com}} \cdot g_1^{-sk_i^{\text{hot}}})^{1/(\tau-i)} = g_1^{(\tilde{g}(\tau)-sk_i^{\text{hot}})/(\tau-i)}$, where $\tilde{g}$ denotes the polynomial committed to by $\widetilde{\text{com}}$, which is identical to a real opening proof with respect to $\widetilde{\text{com}}$. This means that, overall, the distributions are perfectly indistinguishable. □

Claim 2. For all PPT distinguishers $\mathcal{E}$,

$$\mathsf{EXEC}_{H_1,\mathcal{A},\mathcal{E}} \equiv \mathsf{EXEC}_{H_2,\mathcal{A},\mathcal{E}}$$

Proof. The values of σ are identical by construction, since in the honest case we have $\sigma_i = \sigma_i^{\text{hot}}/\sigma_i^{\text{cold}} = H(m)^{sk_i^{\text{hot}} - sk_i^{\text{cold}}}$, which by construction equals $H(m)^{sk_i} = \tilde{\sigma}_i$. □

Claim 3. For all PPT distinguishers $\mathcal{E}$,

$$\mathsf{EXEC}_{H_2,\mathcal{A},\mathcal{E}} \approx_c \mathsf{EXEC}_{H_3,\mathcal{A},\mathcal{E}}$$

Proof. Since sk and dk_i (due to $i \notin C_{\text{cold}}$) are uniformly distributed, we have that (ek_i, vk, ek_i^{sk}) and (ek_i, vk, x_i) are indistinguishable by the DDH assumption.[18] Therefore these distributions are computationally indistinguishable. □

Claim 4. For all PPT distinguishers $\mathcal{E}$,

$$\mathsf{EXEC}_{H_3,\mathcal{A},\mathcal{E}} \approx_s \mathsf{EXEC}_{H_4,\mathcal{A},\mathcal{E}}$$

Proof. These distributions are statistically indistinguishable by the leftover hash lemma applied to $\mathcal{H}$ (see Lemma 2) since the inputs x_i are uniformly distributed. □

Claim 5. For all PPT distinguishers $\mathcal{E}$,

$$\mathsf{EXEC}_{H_4,\mathcal{A},\mathcal{E}} \approx_c \mathsf{EXEC}_{\mathcal{F}_{\mathsf{HC}},\mathcal{S},\mathcal{E}}$$

Proof. The only simulated elements which are not identical to H_4 are the partial signatures $\sigma_i, \sigma_i^{\text{hot}}, \sigma_i^{\text{cold}}$ in the case in which only one of the hot/cold parties is corrupted, the proofs of remembrance, and the hot shares $\widetilde{sk_i^{\text{hot}}}$ for $i \notin C_{\text{cold}}$. We now argue their indistinguishability one by one.

The proof of remembrance bits b^* are indistinguishable from the real execution by the soundness of the proof systems Π_{DL} and Π_{EKS}. Indistinguishability of the hot shares is also simple to see: in H_4, $\widetilde{sk_i^{\text{hot}}}$ for $i \notin C_{\text{cold}}$ is computed as $sk_i + r$ where $(x, r) \in \mathcal{L}_{\mathcal{H}}$, while in the ideal execution, the value is directly sampled from $\mathbb{Z}_p$. Since r is uniform, these elements are statistically indistinguishable. Notice that the simulation is consistent even with proactive refreshes, since the simulator updates $\widetilde{sk_i^{\text{hot}}}$ with every refresh and the other values depending on it remain consistent with the current sk_i refreshed by $\mathcal{F}_{\mathsf{HC}}$.

Finally, we argue (perfect) indistinguishability of the partial signatures in the partial corruption case. During recovery, if only the hot party is corrupt, the cold party in H_4 nevertheless sends it a correctly computed partial signature $\sigma_i^{\text{cold}} = H(m)^r$, where $r = \mathcal{H}(vk^{dk_i})$ if $i \in C_{\text{cold}}$ and $(x_i, r) \in \mathcal{L}_{\mathcal{H}}$ otherwise. In the former case, this is identical to the simulated cold signature. In the latter case, this is also equal by construction: the simulated value is $H(m)^{\widetilde{sk_i^{\text{hot}}}}/\sigma_i = H(m)^{\widetilde{sk_i^{\text{hot}}} - sk_i}$, and since $\widetilde{sk_i^{\text{hot}}}$ is identically distributed to $sk_i^{\text{hot}} = sk_i + r$ this is indistinguishable from the value $H(m)^r$ sent in H_4. This also implies that σ_i is indistinguishable in the two executions, since in both cases it is chosen by $\mathcal{A}$.

On the other hand, when only the cold party is corrupt. In this case, in H_4 we have $\sigma_i = H(m)^{sk_i^{\text{hot}}}/\sigma_i^{\text{cold}^*}$, while in the ideal world it is $H(m)^{sk_i} R^* = H(m)^{sk_i + \mathcal{H}(\widetilde{vk_i^{dk_i}})}/\sigma_i^{\text{cold}^*}$. Since in this case $sk_i^{\text{hot}} = sk_i + \mathcal{H}(vk_i^{dk_i})$ in H_4, and dk_i and $\widetilde{dk_i}$ are identically distributed, these values are indistinguishable.

Overall, the two executions are computationally indistinguishable. □

In conclusion, the real world execution of Throback is indistinguishable from an ideal execution with the ideal functionality $\mathcal{F}_{\mathsf{HC}}$. As we argued above, this implements threshold BLS signatures with the addition of proofs of remembrance. Thus, Throback is UC-secure. □

I Untrusted Hot Refreshes Using a Bulletin-Board

In Section 4.2.2, we implemented our proactive refreshes by using the client C as a trusted dealer. Specifically, the correctness of our proactive refresh protocols assumed two properties of the update polynomial $z(X)$: that it is of degree $t-1$, and that $z(0) = 0$. In this section, we show how to remove this assumption so that share refreshes can be distributed by any untrusted third party (e.g., a designated custodian or other coordinator) without "poisoning" the sharing (i.e., destroying the backed-up secret with a malformed share refreshes).

To enforce the degree requirement, we use a folklore technique [19, §2.5]: the client publishes an additional public "degree commitment" $\text{dcom} := g_1^{\tau^{d-t} \cdot z(\tau)}$, which is verified by checking

$$e(\text{dcom}, g_2) = e(\text{ucom}, g_2^{\tau^{d-t}}),$$

where $d-1$ is the degree of the KZG CRS. This ensures that the polynomial $z(X)$ committed to by ucom is of degree at most $t-1$. (This requires expanding the CRS with additional elements in $\mathbb{G}_2$.) To enforce the evaluation at zero, C can simply provide an opening proof for ucom at $X = 0$, which we refer to as ζ_0. The system (e.g., the parties collectively, or the bulletin board itself) can check these

[18] Strictly speaking, $sk \in \mathbb{Z}_p$ but $dk_i \in (\mathbb{Z}_p)^2$ and therefore $ek_i, x_i \in (\mathbb{G}_2)^2$, so we have 5-tuples rather than 3-tuples, namely $(g_2^{dk_{i,1}}, g_2^{dk_{i,2}}, g_2^{sk}, g_2^{dk_{i,1}sk}, g_2^{dk_{i,2}sk})$ in the first distribution and $(g_2^{dk_{i,1}}, g_2^{dk_{i,2}}, g_2^{sk}, x_{i,1}, x_{i,2})$ in the second. However, these are still indistinguishable by standard DDH via a hybrid argument over the second index $j = 1, 2$. Similarly, indistinguishability for all $i \notin C_{\text{cold}}$ holds via a hybrid argument over i.

Protocol 4: Proof of correct share refresh (Π_{Ref})

Public parameters: $g_1, g_2, \mathsf{crs}, e$ as defined in Table 2.

$\mathsf{Prove}((\mathsf{crs}, \mathsf{ucom}, t-1, \{\delta_i\}_{i\in[n]}); z(X)) \to (\{\zeta_i\}_{i\in[n]}, \pi_z)$: Given a KZG commitment ucom to update polynomial $z(X)$, the latter's claimed degree $t-1$, and each party's share refresh information δ_i, use $z(X)$ to compute the following:

1. For each $i \in [n]$, prove that $\delta_i = z(i)$ by computing $(\delta_i, \zeta_i) \leftarrow \mathsf{KZG.Open}(\mathsf{crs}, z(X), i)$.
2. Prove that $z(0) = 0$ and $z(X)$ has degree $(t-1) \leq (d-1)$ by computing $(0, \zeta_0) \leftarrow \mathsf{KZG.Open}(\mathsf{crs}, z(X), 0)$ and $\mathsf{dcom} := g_1^{\tau^{d-t} \cdot z(\tau)}$ using crs. Let $\pi_{\mathsf{ucom}} := (\zeta_0, \mathsf{dcom})$.
3. Output $(\{\zeta_i\}_{i\in[n]}, \pi_{\mathsf{ucom}})$.

$\mathsf{HVerify}((\mathsf{crs}, \mathsf{ucom}, i, \delta_i), \zeta_i) \to \{0,1\}$: Given crs, a KZG commitment ucom, party index i, and share refresh information δ_i, output $\mathsf{KZG.Verify}(\mathsf{crs}, \mathsf{ucom}, i, \delta_i, \zeta_i)$.

$\mathsf{UCVerify}((\mathsf{crs}, \mathsf{ucom}, t-1), \pi_{\mathsf{ucom}}) \to \{0,1\}$: Given crs, a KZG commitment ucom, and its supposed degree $t-1$, verify the proof $\pi_{\mathsf{ucom}} = (\zeta_0, \mathsf{dcom})$ by outputting 1 iff the following hold:

$$\mathsf{KZG.Verify}(\mathsf{crs}, \mathsf{ucom}, 0, 0, \zeta_0) = 1$$
$$e(\mathsf{dcom}, g_2) = e(\mathsf{ucom}, g_2^{\tau^{d-t}})$$

additional proofs, and only updates com to $\mathsf{com}' := \mathsf{com} \cdot \mathsf{ucom}$ if the checks pass.

Additionally, the hot parties should also be sure that the untrusted dealer is using the *same* polynomial $z(X)$ for all of them when computing their share updates δ_i. This is easily done, since our construction already provides each P_i^{hot} with an evaluation proof ζ_i for δ_i. In the case of a trusted dealer, these were assumed to be computed correctly so P_i^{hot} only used them to blindly update the its key share and opening proof $(sk_i^{\mathsf{hot}}, \pi_i)$. Now, with an untrusted dealer, parties will instead first check the correctness of the update by ensuring $\mathsf{KZG.Verify}(\mathsf{crs}, \mathsf{ucom}, i, \delta_i, \zeta_i) = 1$, and only update their key share and opening proof if verification passes.

We summarize the proofs of correct share refresh in Protocol 4 and refer to the protocol as Π_{Ref}. Verification is split between verifying the well-formedness of the update polynomial $z(X)$ committed to by ucom (i.e., degree and zero opening) via UCVerify and of each hot party's refresh information δ_i via HVerify (executed by each party individually for its own update information).

Given Π_{Ref}, it is fairly simple to realize $\mathcal{F}_{\mathsf{ESS}}$ in a manner that is secure against a malicious dealer C except for the setup phase. To do this, we assume a public bulletin board functionality $\mathcal{F}_{\mathsf{BB}}$ with limited programmability. This functionality will store the most up-to-date commitment com to the encrypted shares $sk_1^{\mathsf{hot}}, \ldots, sk_n^{\mathsf{hot}}$. Whenever the shares are refreshed, it will also check the correctness of the update ucom before making the new commitment $\mathsf{com}' = \mathsf{com} \cdot \mathsf{ucom}$ available.

Concretely, instead of only running the steps of $\mathcal{F}_{\mathsf{ESS}}$ locally, C will compute some additional values using Π_{Ref} to let $\mathcal{F}_{\mathsf{BB}}$ and each P_i^{hot} check the correctness of the share and update commitments. The proof for the update commitment ucom will be checked by $\mathcal{F}_{\mathsf{BB}}$ before updating the stored commitment. Additionally, C will let $\mathcal{F}_{\mathsf{BB}}$ store and distribute ucom instead. We describe the full protocol realizing $\mathcal{F}_{\mathsf{ESS}}$ in the $\mathcal{F}_{\mathsf{BB}}$-hybrid model in Protocol 5. Changes with respect to a trusted third party locally executing $\mathcal{F}_{\mathsf{ESS}}$ are highlighted.

Theorem 2. The encrypted secret sharing protocol in Protocol 5 UC-realizes $\mathcal{F}_{\mathsf{ESS}}$ in the $\mathcal{F}_{\mathsf{BB}}$-hybrid model, assuming binding and hiding of KZG commitments.

Proof. Let $\mathcal{A}$ be the adversary interacting with the parties $P_1, \ldots, P_n, C$ running Protocol 5. We will construct a simulator $\mathcal{S}$ running in the ideal world so that no environment $\mathcal{E}$ can distinguish an execution of the ideal-world interaction from the real protocol. $\mathcal{S}$ will interact with $\mathcal{F}_{\mathsf{ESS}}$, $\mathcal{E}$, and invoke a copy of $\mathcal{A}$ to run a simulated interaction of the protocol (which we again refer to as the internal interaction). The dummy parties, communication tapes, and message delivery are handled in the same way as in the proof of Theorem 1.

Corruptions. Whenever $\mathcal{A}$ corrupts a party P_i in the internal execution, $\mathcal{S}$ corrupts the corresponding dummy party.

Setup and auxiliary information. Since all the computation in setup is done by C and we do not allow it to be corrupted in this phase, $\mathcal{S}$ only passes messages to and from $\mathcal{F}_{\mathsf{ESS}}$. It also keeps track of client-threshold pairs in a list D'. Similarly, retrieving auxiliary information involves no computation, so $\mathcal{S}$ again only passes along messages.

Generating share refreshes. If C is corrupt, $\mathcal{S}$ retrieves $(C, t) \in D'$ and observes what values ucom, π_{ucom} the adversary $\mathcal{A}$ sends to $\mathcal{F}_{\mathsf{BB}}$ in the internal interaction. It computes $b^* \leftarrow \Pi_{\mathsf{Ref}}.\mathsf{UCVerify}((\mathsf{crs}, \mathsf{ucom}, t), \pi_{\mathsf{ucom}})$. When $\mathcal{F}_{\mathsf{ESS}}$ sends $\mathcal{S}$ a ZeroSetupRequest message, it responds with b^*.

$\mathcal{S}$ also keeps track of the values (x_i, ζ_i) sent by $\mathcal{A}$ to each P_i in the internal interaction and computes $b_i^* \leftarrow \Pi_{\mathsf{Ref}}.\mathsf{HVerify}((\mathsf{crs}, \mathsf{ucom}, i, x_i), \zeta_i)$. When $\mathcal{F}_{\mathsf{ESS}}$ sends $\mathcal{S}$ a ZeroShareRequest message, it responds with b_i^*. □

I.1 Evaluation

For completeness, we also benchmark our untrusted refresh protocol. We compare these runtimes with trusted refresh in Table 6. Compared to the trusted version, untrusted refresh generation (by C or some other untrusted coordinator) additionally includes the time to generate the opening at zero and the degree commitment dcom. We also include the (global) verification of these two elements in this benchmark. Untrusted update of the hot shares (by each hot party) additionally includes the time required to verify the KZG opening proof ζ_i.

Protocol 5: Encrypted secret sharing

Public parameters:

$\mathbb{G}_1, \mathbb{G}_2, g_1, g_2, d$ as defined in Table 2 and an extended KZG common reference string $\text{crs} = \{g_1, g_1^{\tau}, \ldots, g_1^{\tau^{d-1}}, g_2, g_2^{\tau}, g_2^{\tau^2}, \ldots, g_2^{\tau^{d-1}}\}$.

Setup: On input $(\texttt{sid}, \text{SSSetup}, C, (t, \mathcal{P}, \{ek_i\}_{i\in[n]}))$, where $t, n \in \mathbb{N}$ s.t. $n = |\mathcal{P}|$ and $t \le n \le d$, $\mathcal{P} = \{P_1 \ldots P_n\}$ a set of parties, and $\{ek_i\}_{i\in[n]}$ a set of public (encryption) keys ($\forall i \in [n], ek_i \in (\mathbb{G}_2)^2$), C proceeds as follows:

(1) Sample $x \leftarrow\!\!\$\ \mathbb{Z}_p \setminus \{0\}$. Let $y := g_2^x$.
(2) Generate t-out-of-n Shamir Shares of x as $x_1, \ldots x_n \in \mathbb{Z}_p$. Let $y_i := g_2^{x_i}\ \forall i \in [n]$.
(3) Interpolate the degree-n polynomial $\tilde{f}$ such that $\tilde{f}(i) = \mathcal{H}(ek_i^x) + x_i\ \forall i \in [n]$. Compute $\text{com} \leftarrow\!\!\$\ \text{KZG.Com}(\text{crs}, \tilde{f})$ and send $(\texttt{sid}, \text{Setup}, C, (t, y, \text{com}))$ to $\mathcal{F}_{\text{BB}}$.
(4) Delete any existing entries $(C, *, *) \in D$ and add $(C, \mathcal{P}, t)$ to D.
(5) For each $i \in [n]$, compute $(\tilde{x}_i, \pi_i) \leftarrow \text{KZG.Open}(\text{crs}, \tilde{f}, i)$ and output $(\texttt{sid}, \text{SecretShare}, P_i, (C, i, \tilde{x}_i, \pi_i))$.
(6) Finally, output $(\texttt{sid}, \text{SSSetupDone}, C, y, \{y_i\}_{i\in[n]}, \text{com})$.

Generating share refreshes: On input $(\texttt{sid}, \text{ShareRefresh}, C, (t, \mathcal{P}))$, where $t, n \in \mathbb{N}$ s.t. $n = |\mathcal{P}|$ and $t \le n \le d$, and $\mathcal{P} = \{P_1 \ldots P_n\}$ a set of parties, C will proceed as follows:

(1) Generate t-out-of-n Shamir shares of 0 as $x_1, \ldots x_n \in \mathbb{Z}_p$; let $z(X)$ be the polynomial used.
(2) Compute $\text{com}_0 \leftarrow \text{KZG.Com}(\text{crs}, z(X))$ and $(\{\zeta_i\}_{i\in[n]}, \pi_z) \leftarrow \Pi_{\text{Ref}}.\text{Prove}((\text{crs}, \text{com}_0, t-1, \{x_i\}_{i\in[n]}); z(X))$. Send $(\texttt{sid}, \text{ComUpdate}, C, (\text{com}_0, \pi_z))$ to $\mathcal{F}_{\text{BB}}$, which outputs $(\texttt{sid}, \text{ComUpdateResult}, C, b)$.
(3) Send (x_i, ζ_i) to P_i for all $i \in [n]$, then output $(\texttt{sid}, \text{ZeroSetupDone}, C, \text{ucom})$.
(4) Each party P_i for $i \in [n]$ will send $(\texttt{sid}, \text{UComRecover}, P_i, (C))$ to $\mathcal{F}_{\text{BB}}$ and receive $(\texttt{sid}, \text{UCom}, P_i, (C, \text{ucom}))$ in return. It checks that $\Pi_{\text{Ref}}.\text{HVerify}((\text{crs}, \text{ucom}, i, x_i), \zeta_i) = 1$. If not, set $x_i, \zeta_i = \perp$. Output $(\texttt{sid}, \text{ShareRefreshResult}, P_i, (C, x_i, \zeta_i))$.

Providing auxiliary information: On input $(\texttt{sid}, \text{AuxRecover}, P_i, (C))$ for some client C, P_i sends $(\texttt{sid}, \text{ClientInfoRecover}, P_i, (C))$ to $\mathcal{F}_{\text{BB}}$, which outputs $(\texttt{sid}, \text{AuxInfo}, P_i, (C, y, \text{com}))$. P_i outputs $(\texttt{sid}, \text{AuxInfo}, P, (C, y, \text{com}))$.

J Artifact Appendix

J.1 Abstract

We provide a Rust crate implementing Throback, our auditable hot-cold threshold backup based on BLS signatures. We provide functions for each of our protocol interfaces, which will be executed by a hot or cold custodian or by a client depending on the function. In particular, we implement custodian registration, backup creation, recovery via threshold signing, creation and verification of proofs of remembrance, and hot share refreshes. We also provide a benchmarking script which measures the execution time of each interface, averaged over a user-specified number of iterations, for various settings of (t, n). Our code is made available under the Apache 2.0 license.

	Time			
	Dealer			P_i^{hot}
$(t, n) =$	$(3, 5)$	$(5, 20)$	$(67, 100)$	any
Trusted ($\mathcal{F}_{\text{ESS}}$)	8ms	29ms	9.682s	0.266ms
Verifiable (Protocol 5)	15ms	36ms	9.737s	1.997ms

Table 6: Comparison of trusted (i.e., C locally executing $\mathcal{F}_{\text{ESS}}$) and untrusted (using Protocol 5) hot share refreshes. The times for the former are identical to Tables 3 and 4 and repeated here for ease of comparison.

J.2 Description & Requirements

We provide a `README` in the root of the repository. This also includes the configuration file `Cargo.toml` for Rust's build system Cargo, which allows easy code-building and dependency-handling. The source code which actually implements Throback is located in the `src/` directory; it contains multiple files which implement the building blocks used in our construction, and `client.rs` and `custodian.rs` which implement the high-level algorithms run, respectively, by the client and by the hot/cold portions of each custodian. Finally, the `examples/` directory contains a benchmarking script which can be used to recreate the benchmarks in our paper.

J.2.1 Security, privacy, and ethical concerns. Our code is completely self-contained and only runs in its source directory. It does not access any other files on the host computer, but only creates a new file to record the benchmarks. It does not attempt to delete any data. The only foreseeable risk would be executing the benchmarking script with parameters that are too large, crashing the user's machine. We recommend not exceeding the parameters listed in our reported benchmarks, which an average commercial laptop released in the last 5-10 years should be able to handle. As a reference, our own benchmarks were taken on a 2020 MacBook Pro with an 8-core M1 chip and 16GB of RAM.

J.2.2 How to access. Our source code is permanently available on Zenodo with the following DOI: https://doi.org/10.5281/zenodo.20633750

It is also available on GitHub at the following URL: https://github.com/LF-Decentralized-Trust-labs/agora-key-share-proofs.

Functionality 5: $\mathcal{F}_{\mathsf{BB}}$ (Programmable bulletin board)

Public parameters: $\mathbb{G}_1$, crs as defined in Table 2.
Internal variables:
$D = (C, t, y, \mathsf{com}, \mathsf{epoch})$ / client info
$U = (C, \mathsf{ucom}, \mathsf{epoch})$ / clients' past update commitments

On input (**sid**, Setup, C, (t, y, com))**:**
(1) Delete any existing entries $(C, *, *, *, *) \in D$ and $(C, *, *) \in U$. Add $(C, vk, t, \mathsf{com}, \mathsf{epoch} := 1)$ to D and output $(\mathtt{sid}, \mathsf{SetupResult}, C, (1))$.

On input (**sid**, ComUpdate, C, $(\mathsf{ucom}, \pi_{\mathsf{ucom}})$)**:**
(1) Retrieve $(C, vk, t, \mathsf{com}, \mathsf{epoch}) \in D$ for the maximum value of epoch.
(2) If $\Pi_{\mathsf{Ref}}.\mathsf{UCVerify}((\mathsf{crs}, \mathsf{ucom}, t), \pi_{\mathsf{ucom}}) = 1$, set $b := 1$ and add $(C, t, y, \mathsf{com} \cdot \mathsf{ucom}, \mathsf{epoch}{+}{+})$ to D and $(C, \mathsf{ucom}, \mathsf{epoch}{+}{+})$ to U. Otherwise set $b := 0$.
(3) Output $(\mathtt{sid}, \mathsf{ComUpdateResult}, C, (b))$.

On input (**sid**, ClientInfoRecover, P, (C))**:**
- Retrieve $(C, vk, *, \mathsf{com}, \mathsf{epoch}) \in D$ for the maximum value of epoch and output $(\mathtt{sid}, \mathsf{AuxInfo}, P, (C, y, \mathsf{com}))$.

On input (**sid**, UComRecover, P, (C))**:**
- Retrieve $(C, \mathsf{ucom}, \mathsf{epoch}) \in U$ for the maximum value of epoch and output $(\mathtt{sid}, \mathsf{UCom}, P, (C, \mathsf{ucom}))$.

J.2.3 Hardware dependencies. None.

J.2.4 Software dependencies. Our implementation is written in Rust and uses the Cargo build system. Instructions for installing Rust and Cargo on any operating system can be found in the Rust Book: https://doc.rust-lang.org/stable/book/ch01-01-installation.html. Once Cargo has been installed, the additional dependencies (listed in `Cargo.toml`) are handled directly by the `cargo build` command (see the next section).

J.2.5 Benchmarks. None.

J.3 Set Up

J.3.1 Installation. Download the artifact from Zenodo and unzip the archive, or clone the GitHub repository using `git clone https://github.com/LF-Decentralized-Trust-labs/agora-key-share-proofs.git`. In the root of the respository, run `cargo build` to compile the code and dependencies.

J.3.2 Basic test. As a simple functionality test, run `cargo test`. This will run all the tests in the source code; if 0 tests fail, the code is working as expected.

J.4 Evaluation Workflow

J.4.1 Major claims.

(C1): Throback achieves efficiency on the order of milliseconds for most of its interfaces, which are independent of the threshold t and number of parties n. Generating a new backup (`ClientRegister`) and generating share refresh information (`ShareRefresh`), both executed by the client, instead scale with the setting of (t, n) and take several seconds for the largest configuration. This is shown by experiment (E1) described in Section 5, whose results are reported in Tables 3 and 4.

J.4.2 Experiments.

(E1): *Benchmarks [5 person-minutes + 35 compute-minutes]:* Measures the average runtime of each subprotocol of Throback for a small threshold setup with $(t, n) = (3, 5)$, medium setup with $(5, 20)$, and large setup with $(67, 100)$. The average is taken over 100 iterations and the results are written to a file `bench.txt` in the root directory of the project.
Preparation: Follow the setup instructions given in Appendix J.3. Make sure to run the benchmarks from the root directory of the project.
Execution: Run `cargo run -r –example bench [t] [n] [iters]` on the command line, replacing the bracketed values with the appropriate parameters. More explicitly, use the following three commands to recreate the benchmarking settings used for the paper:

```
# small setting
cargo run -r --example bench 3 5 100
# medium
cargo run -r --example bench 5 20 100
# large
cargo run -r --example bench 67 100 100
```

The code will print some text to stdout to indicate its progress, and will create a file `bench.txt` in the root directory of the project if it does not exist yet. Once the benchmarks are completed, it simply exits.
Results: The results of each command are appended as a table to `bench.txt` in the root directory of the project. The row labels correspond, in order, to the subprotocols of Protocol 1 (pages 11-12 of the paper), with the exception of the "refresh" times, which refer to the final subprotocol ("(Hot) Proactive Refresh"). Each header in the figure is labeled above with the formal name of its interface, e.g., ColdRegister for the client registration. These interface names are used in Tables 3 and 4 where the results are reported.
The "refresh" times are split into two rows. The "client" portion refers to steps (1) and (2) of this subprotocol, which are executed by the client C, and the "hot" portion refers to step (3), namely the computation done by each individual hot custodian P_i^{hot} (where $i \in [n]$). Finally, each refresh entry also has a second column labeled "untrusted". This refers to the untrusted hot refresh extension detailed in Appendix I of the paper, whose benchmarks are reported in Table 6 of that section.
The results should match approximately with the numbers reported in Tables 3, 4, and 6 of the paper, with some variation to be expected due to running on different hardware.

J.5 Notes on Reusability

Our code encourages reusability by being hosted in a Linux Foundation lab under an Apache 2.0 license. Anyone can feel free to reuse the code safely with the same degree of protection and assurance that they would expect from other Linux Foundation projects.

J.6 Version

Based on the LaTeX template for Artifact Evaluation V20220926.

Received 14 January 2026; revised 20 May 2026; accepted 5 June 2026; finalized 7 August 2026