Anonymous Outsourced Statekeeping with Reduced Server Storage

Dana Dachman-Soled¹, Esha Ghosh², Mingyu Liang¹, Ian Miers¹, and Michael Rosenberg¹

¹ University of Maryland, College Park {danadach,mliang,imiers,micro}@umd.edu ² Microsoft Research esha.ghosh@microsoft.com

Abstract. Strike-lists are a common technique for rollback and replay prevention in protocols that require that clients remain anonymous or that their current position in a state machine remain confidential. Strikelists are heavily used in anonymous credentials, e-cash schemes, and trusted execution environments, and are widely deployed on the web in the form of Privacy Pass (PoPETS '18) and Google Private State Tokens. In such protocols, clients submit pseudorandom tokens associated with each action (e.g., a page view in Privacy Pass) or state transition, and the token is added to a server-side list to prevent reuse.

Unfortunately, the size of a strike-list, and hence the storage required by the server, is proportional to the total number of issued tokens, $N \cdot t$, where N is the number of clients and t is the maximum number of tickets per client. In this work, we ask whether it is possible to realize a strikelist-like functionality, which we call the *anonymous tickets functionality*, with storage requirements proportional to $N \log(t)$.

For the anonymous tickets functionality we construct a secure protocol from standard assumptions that achieves server storage of O(N) ciphertexts, where each ciphertext encrypts a message of length $O(\log(t))$. We also consider an extension of the strike-list functionality where the server stores an arbitrary state for each client and clients advance their state with some function $s_i \leftarrow f(s_{i-1}, \texttt{auxinput})$, which we call the anonymous outsourced state-keeping functionality. In this setting, malicious clients are prevented from rolling back their state, while honest clients are guaranteed anonymity and confidentiality against a malicious server. We achieve analogous results in this setting for two different classes of functions.

Our results rely on a new technique to preserve client anonymity in the face of selective failure attacks by a malicious server. Specifically, our protocol guarantees that misbehavior of the server either (1) does not prevent the honest client from redeeming a ticket or (2) provides the honest client with an *escape hatch* that can be used to simulate a redeem in a way that is indistinguishable to the server.

1 Introduction

Strike-lists are a common technique in e-cash protocols, TEEs³, anonymous credentials, and larger protocols like stateful access control [18] and proof carrying data systems. [6] As a concrete example, consider privacy-pass [19] and its derivatives, which are in widespread use on the web [38] for fraud and abuse prevention. In privacy-pass, clients complete a verification step (e.g., completing a CAPTCHA or signing into a verified account), and are issued t one time use tokens. Clients then use these tokens to access resources while remaining anonymous. I.e., they do not link their access request together or to token issuance. Instead, clients reveal only that they hold a token (typically the tokens is the output of a verifiable oblivious pseudo-random function). To enforce access limits and prevent fraud, clients cannot reuse tokens indefinitely. To prevent this, a server maintains a strike-list of used tokens. For each authentication attempt, the server checks that the presented token is not already in the strike-list and then appends it to the list. More elaborate protocols, such as Google's trust tokens [16], extend these techniques to support more sophisticated and multi step anti fraud systems, but retain the core requirement for a strike-list of used tokens.

Originally developed for double spend prevention in e-cash schemes [14], strike-lists form a general technique for rollback prevention that arises from a need to either (1) hide the details of a client's particular state in some state machine or (2) hide the identity of a particular client in a multi client setting (e.g., anonymous credentials and e-cash).

While conceptually simple, a drawback of strike-lists is their rapid growth. The size of the strike-list is proportional to the total number of issued tickets, $\Omega(N \cdot t)$, where N is the number of clients and t is the number of tickets issued per client. Beyond the storage requirements, servers managing stikelists also incur the computational overhead of maintaining the data structures to support fast non-membership checks.⁴

In this work, we consider the following problem:

Is it possible to achieve the functionality of strike-lists with server storage of $O(N \log(t))$?

Specifically, can we develop protocols achieving the functionality, privacy, and anonymity of strike-lists, while (1) requiring the server to store only an $O(\log t)$ size state per client, (2) preserving the star-shaped interaction graph of strikelists (each client communicates only with the central node corresponding to the server), and (3) preserving to the extent possible the minimal round complexity

³ Intel SGX, for example, does not support trusted counters on server class processors and is no longer supported on consumer processors.

⁴ At scale, the costs of maintaining index data structures poses challenges, as shown by experiences with key transparency systems [39].

of strike-lists.⁵ We refer to such protocols as achieving the **anonymous tickets** functionality.

The above functionality is not limited to simple strike-lists for one time tokens. It can be extended to capture more complex protocols requiring rollback prevention. The anonymous tickets functionality outsources storage of state to a server, but the state is limited to a counter. From this perspective, doublespending corresponds to a client rolling back its state (e.g., resetting its counter), but we are not limited to such incremental state transition functions.

In the **anonymous outsourced state keeping** malicious clients are forced to to apply some state transition function $s_i \leftarrow f(s_{-1}, \operatorname{auxinput})$ to the current state the server holds (i.e., without rollback), and honest clients are guaranteed anonymity and privacy against a malicious server. Here we allow the server storage to be $O(N \cdot s)$, where s is the size of the state stored per client. This is clearly also a lower bound, since the states of the clients may be random (or pseudorandom) strings that cannot be compressed. In this setting, the same questions for optimal storage, communication topology, and round complexity exist. And, looking ahead, we will see an additional question: which functions are safe to run given the achievable security guarantees.

The challenge for anonymous state keeping. An immediate solution for removing the strike-list for single use tickets (which we can extend to arbitrary state) is for the server to store an encrypted per client counter initially set to the number of issued tickets, and decremented on each client action until it reaches zero. But this poses several problems. First, the access pattern of counter updates (and in the extension the state update itself) must be hidden.

Second, because clients must prove something about the server state (i.e., that they are not out of tickets), we are now subject to the server equivocating and sending different global state (i.e. the state it stores for all clients) to each client. Forcing consistency of the server's messages across clients seemingly requires byzantine broadcast or an out of band gossip protocol. But resorting to those approaches would violate our star-shaped interaction graph requirement and, in practice, may offer worse tradeoffs than an $\Omega(N \cdot t)$ -sized strike-list.

Third, and more perniciously, we must guard against a malicious server who aborts, rolls back the state of a target client (or all other clients), or selectively injects faults into the state of specific clients, as such strategies can be used to deanonymize clients, even if all clients receive the same state. The high level reason for deanonymization is that only the client whose state was tampered will detect that tampering occurred and potentially behave differently, while all other clients will behave as though the tampering did not occur. Thus, the server can link a party who behaves differently (e.g. aborts) with the tampered account.

As a concrete example, consider a forum operator who accepts anonymous posts given a ticket. The operator can decide to either rollback that specific client state dynamically, e.g., based on the post, leading to the next subsequent

⁵ Here we refer the round complexity of the scheme itself. Many applications, and our definitions, assume an anonymous communication layer such as a VPN or Tor.

post and ticket being linked by a repeated ticket. They could maul the target client's state, leading to a perhaps detectable abort by that client in the future. This is particularly a problem for more complex protocols with extended state. And it is not one that is stopped by non-equivocation: even if all clients receive the same state vector, only one clients entry is malformed.

The last challenge is that strike-lists are incredibly simple from a round complexity and interaction graph perspective. There is a single server who maintains a list and clients send one message to the server. Approaches that solve rollback and aborts using some gossip protocol, a transparency log, or even interactions with multiple other parties are far from ideal. We want to keep the single round of communication, single server model, and the star-shaped interaction graph in our solution. However looking forward, we seemingly must require one round-trip of communication between a client and a server. I.e., the client first requests the server's state and then sends an update.

1.1 Our Results

We now describe our results for the anonymous tickets and anonymous outsourced state-keeping functionalities. For the formal definitions of the ideal functionalities and formal theorem statements, see Fig. 1, Fig. 13, Theorem 3, and Theorem 4.

We obtain a protocol achieving the anonymous tickets ideal functionality with our desired properties of (1) server storage of O(N) ciphertexts, each encrypting a message of length $O(\log(t))$, where N is the number of parties and t is the maximum number of tickets issued to a single user, (2) preserving the starshaped interaction graph where each client communicates only with the server, and (3) achieving a 3-message redeem protocol in which the client sends a request to redeem, downloads state from the server, and uploads state back to the server.

Our protocol is constructed in a *hybrid* model, meaning the protocol has access to other ideal functionalities. Specifically, it relies on a non-interactive zero-knowledge proof (NIZK) functionality $\mathcal{F}_{nizk}^{\mathcal{R}}$, and an anonymous messaging functionality \mathcal{F}_{am} (further, instantiations of the former must themselves be in the common reference string (CRS)-hybrid model). \mathcal{F}_{am} captures our assumption of an anonymous communication layer, which is implicitly required in most real-world strike-list schemes to hide the IP address of the ticket redeemer. Finally, we make an additional setup assumption that the server's public key is well known.

Theorem 1 (Informal). Assuming setup of the server's public key, as well as the existence of an additively homomorphic encryption scheme, a MAC scheme, a digital signature scheme, and a commitment scheme, there exists a protocol that realizes the anonymous tickets ideal functionality in the $\mathcal{F}_{nizk}^{\mathcal{R}}, \mathcal{F}_{am}$ -hybrid model with server storage and per-redeem communication of O(N) ciphertexts, each encrypting a message of length $O(\log(t))$, where N is the number of users, and t is the number of tickets issued per user. We note that in the strike-list setting, the server storage would consist of $Nt\lambda$ tickets, where t is the maximum number of tickets issued per user and each ticket has length λ . The storage for our scheme is therefore smaller than the strike-list when $t > L/\lambda$, where L is the length of a ciphertext of the additively homomorphic encryption (AHE) scheme encrypting a MAC of length κ and a message of length $\log(t) < \kappa$.

We briefly outline some well known ways to instantiate the assumptions for the above theorem. NIZKs in the CRS model can be instantiated from various standard assumptions such as DLIN, quadratic residuosity, and LWE. MACs and digital signature schemes can be constructed from any one-way function. AHE schemes can be constructed from DDH, DCRA (decisional composite residuosity), and LWE. We note that when all our underlying primitives are instantiated from LWE we achieve plausible post-quantum security.

Under somewhat stronger assumptions, we are able to realize the more general functionality of anonymous outsourced state keeping. In particular, with knowledge-of-exponent assumptions or Diffie-Hellman-type assumptions in bilinear groups, we can build recursive succinct non-interactive arguments of knowledge (recursive SNARKs) with constant proof size. [4, 35, 15, 7, 9]

As for the anonymous tickets functionality, our protocol for the anonymous outsourced state-keeping functionality achieves the desired properties of (1) server storage of O(N) ciphertexts, each encrypting a message of length O(s), where s is the size of the client's state, (2) a star-shaped interaction graph, and (3) a 3-message redeem protocol.

Theorem 2 (Informal). Assuming setup of the server's public key, as well as the existence of an additively homomorphic encryption scheme, a MAC scheme, a digital signature scheme, a commitment scheme, and a recursive SNARKs scheme, there exists a protocol that realizes the anonymous outsourced state keeping functionality in the $\mathcal{F}_{nizk}^{\mathcal{R}}, \mathcal{F}_{am}$ -hybrid model with server storage and perredeem communication of O(N) ciphertexts, each encrypting a message of length O(s), where N is the number of users, and s is the size of the client's state.

Our protocols are given in the anonymous messaging \mathcal{F}_{am} -hybrid model. We choose to present our protocols in this way since our primary objective, like that of previous work that utilizes blind signatures [19], is to achieve "application layer" anonymity, in which a malicious server cannot deanonymize the interaction with the clients based solely on the communication content. Therefore, achieving "network layer" anonymity is an orthogonal topic and we do not consider it in our paper. In practice, the threat posed by a network adversary can be mitigated through the use of anonymous routing such as Tor.

In our protocol, the client sends and receives N ciphertexts during each redeem, so the client's bandwidth is proportional to N. In Section 1.3 we discuss an approach to reduce the client's bandwidth to depend polynomially on security parameter κ . but to be independent of N (we refer to this as constant bandwidth). To achieve this, we require sophisticated primitives such as authenticated private information retrieval (PIR) [17] and non-interactive anonymous

κ	Security parameter
N	Number of users
t	User's initial quantity of tickets
n_{idx}	Number of tickets encoded by ciphertext at index idx
ер	Current redemption epoch
T	State table held by server
m	Message signed by server on successful redemption

Table 1. Notation for this paper

routing [36]. We defer a fully formal treatment of the constant client bandwidth case to future work.

1.2 Technical Overview

Before moving to formal definitions and proofs, we present an overview of our ideal construction and its concrete instantiation. We provide intuition for the decisions and discuss the many difficulties that come with defining and building such a protocol. As an aide, we include a legend for notation in Table 1.

Ideal functionality One of our main contributions is presenting a formal definition of an ideal functionality for the anonymous tickets. We briefly overview here key features and design decisions we made in defining the functionality. For the formal definition, see Fig. 1.

To describe the ideal functionality of our protocol, we first must enumerate the parties involved. There are the *users*, or *clients*, i.e., the individuals who have counters associated with their actions. There is the server, which obliviously manages the clients' state, ensuring the consistency of each counter update. And there are the *service providers*, i.e., the web hosts who wish to use user counters for rate limiting. We now describe the procedures exposed by our ideal functionality.

- **Initialize** is called by the environment to bring up the functionality. In order to ensure service providers can be convinced that a user's counter is nonzero, we give the server a keypair $(sk_{ServID}, pk_{ServID})$, and publicize the public key to all parties. The server will sign redemptions with this key so that service providers can verify it noninteractively.
- **Register** is called by a user who wishes to sign up with the server. On receiving a registration request, the functionality extends its table to contain the user's counter, which it initializes to a default value t.
- **EndRegister** is called by the server to close registration. Once this is done, redemptions may start.
- **Redeem** is called by a user who wishes to decrement their counter. They receive a signature of an arbitrary message of their choosing m under $\mathsf{pk}_{\mathsf{ServID}}$. This m may be a challenge value provided by a service provider. For reasons

described in the next subsection, redemptions happen in *epochs*. A user may only redeem once per epoch.

UpdateEpoch is called by the server to start a new epoch, and thus permit users to redeem if they've already done so in the current epoch.

We describe some properties of our ideal functionality. The functionality is correct—an honest user who registers may redeem and receive up to t signatures on messages of their choice. Further, the functionality is robust to malicious clients—clients are not able to redeem more than t times, nor can they redeem more than once per epoch.

Clients also remain anonymous in this functionality, once they've registered. We capture this by having the Redeem procedure send only the user's message m and perceived epoch ep' to the adversary (we will see in the next subsection why sending ep' is necessary). Importantly, the specific value of ep' may leak some information about a user if, for example, different users had different notions of how much time had passed since the last epoch. We intentionally model this leakage in the ideal functionality, and propose the mitigation in the next subsection that every honest user have approximately synchronized clocks.

Finally, the functionality is robust to malicious servers. The server can always refuse to respond to a client or deny their redemption request while continuing to serve other clients. But, by the client anonymity property, the server cannot do this (beyond the leakage amount) on the basis of the identity of the client.

We note that these properties are achieved in a setting where the only value that must be public and consistent between users is pk_{ServID} . This is in contrast to authenticated dictionary systems such as SEEMless or Parakeet [12, 30], where auditors must continuously query the server and gossip the answers they receive or rely on a third party ledger.

Building up to our protocol Recall that, in our ideal functionality, a malicious server should learn nothing about the states stored by the clients (privacy) nor should it learn which client requested a state update in a given interaction (anonymity). These guarantees should hold for all honest clients, even if a subset of malicious clients collude with the server.

To achieve such a functionality, it is clear that we would need the outsourced client state to be encrypted to achieve privacy. Moreover, the clients should be able to retrieve their state (a simplified state is just a counter) obliviously to achieve anonymity. Perhaps, the first thought that comes to mind, when we think of oblivious access at the server, is Oblivious RAM (ORAM). However, in our model, there are multiple clients. They do not interact with each other and do not have any shared state. So, an ORAM solution does not immediately work for us.

To work up to our final solution, let us start with a naive first attempt for state consisting only of a counter.

Let each client have a counter associated with their account. At the time of registration, a client generates an encryption key pair (sk, pk) and sends pk

to the server. The server then sets the count, denoted as n, to initial value t, encrypts it $\mathsf{ct} \leftarrow \mathsf{Enc}_{\mathsf{pk}}(n)$, and stores $(\mathsf{pk}, \mathsf{ct})$ in its table T of client information. If a client with index idx^* wants to redeem a ticket, they download T and send a fresh table to to the server,

$$T' = [(\mathsf{pk}_1, \mathsf{ct}_1'), \dots, (\mathsf{pk}_N, \mathsf{ct}_N')]$$

where the client's own count has been decremented, $\mathsf{ct}'_{\mathsf{idx}^*} = \mathsf{Enc}_{\mathsf{pk}_{\mathsf{idx}^*}}(n-1)$, and the remaining $\mathsf{ct}'_{\mathsf{idx}} = \mathsf{ct}_{\mathsf{idx}}$. The server then updates T to T'.

Obviously, there are several problems with this proposal.

Adding anonymity. The above scheme reveals which client is updating, since only the idx^{*}-th ciphertext differs between T and T'. We fix this by using a rerandomizable public-key encryption scheme, such as ElGamal or Pallier, and letting each $ct'_{idx} = \operatorname{Rerand}_{pk_{idx}}(ct_{idx})$ for idx \neq idx^{*}.

Adding concurrency. This amended scheme does not permit two users to make an update concurrently since it is forced to pick which T' to accept as the new table. We note, though, that updates from separate users ought to be commutative: if user A decremented and then user B decremented, this is semantically the same as if user B decremented and then user A decremented (as long as we ensure the accounts are distinct). To leverage this observation, we use an additively homomorphic encryption scheme for $\mathsf{Enc}_{\mathsf{pk}}$. In the new scheme, to redeem, a user with index idx^* computes a vector $\Delta = (0, \ldots, 0, -1, 0, \ldots, 0)$ whose -1 appears at index idx^* . The user encrypts Δ , using an additively homomorphic encryption scheme, and gets a list of ciphertexts $\hat{\mathsf{ct}} = [\hat{\mathsf{ct}}_{\mathsf{idx}}]_{\mathsf{idx}\in[N]}$ where $\hat{\mathsf{ct}}_{\mathsf{idx}} \leftarrow \mathsf{Enc}_{\mathsf{pk}_{\mathsf{idx}}}(\Delta_{\mathsf{idx}})$. The user sends $\hat{\mathsf{ct}}$ to the server, and the server lets its new ciphertexts $\mathsf{ct}'_{\mathsf{idx}} = \mathsf{ct}_{\mathsf{idx}} + \hat{\mathsf{ct}}_{\mathsf{idx}}$. Now multiple users can produce $\hat{\mathsf{ct}}$ values concurrently, and the server can apply them in any order.

Adding security against malicious users. Currently, there is nothing stopping a user from letting $\Delta_{idx^*} = 0$ and thus avoiding decrementing its ticket counter. Similarly, there is nothing stopping a user from picking nonzero plaintexts for the other users, thus modifying their counters. To prevent this, we force users to compute a zero-knowledge proof (ZKP) that n > 0 and their \hat{ct} is well-formed, i.e., that each \hat{ct}_{idx} is an encryption of 0 under pk_{idx} , except for at one index, where it is the encryption of -1 under pk_{idx^*} , whose secret key sk_{idx^*} is known. This forces the user to only decrement its own counter in each redemption.

Adding security against maliciously concurrent users. There remains one attack against the soundness of the system that a malicious user can perform. Suppose a user's count n is 1. They can still redeem twice by performing two valid redemptions concurrently, both with respect to the same server state T. Both will be accepted, and the count will be at -1, a violation of our desired functionality.

We may prevent this by defining *epochs*, wherein each user can only redeem once and have the server maintain a strike-list only for that epoch. To bind a

redemption request to an epoch, we require every user to include a commitment to a PRF key nk on registration. To redeem in epoch ep, a user computes a *nullifier* $nf = PRF_{nk}(ep)$, and sends it along with a ZKP that the nullifier was computed honestly from its committed nk. The server can then verify that a nullifier has only ever been seen once within an epoch. Epochs are short, as we discuss shortly, in practice limited by how well synced client clocks are. After an epoch expires, the strike-list can be discarded without further communication with any client or reissuing credentials.⁶

Hiding number of redemptions per epoch. An honest-but-curious user can check whether or not there was a redemption during a span of epochs. They simply query the server for T at one epoch, and query again at each subsequent epoch. If the tables remain the same, then no tickets were redeemed. Otherwise, someone redeemed. We can avoid this leakage by requiring the server to rerandomize (equivalently, add an encryption of 0 to) the ciphertexts after every epoch.

Preserving anonymity across epochs and setting epoch length. A malicious server can use different views of the epoch to deanonymize a user. If a server suspects that an epoch query comes from a user who has made a redemption request in a previous epoch ep_{old} , they can claim the epoch number to be ep_{old} . In the case that this user indeed redeemed in epoch ep_{old} , they either generate a duplicated nullifier or have to abort. Either way, the server can use this information to associate the current redeeming attempt with a past redemption.

We fix this by introducing a protocol change and a new assumption of our parties. The new protocol will determine the epoch using a party's local clock, and require the user to send its perceived epoch to the server, who will only proceed if it matches its own epoch (otherwise, a user can redeem with respect to an old state, and thus overflow its counter). This eliminates the maliciously chosen **ep** attack, but it does not remove leakage: a user lagging behind may send several outdated epochs for multiple redeem requests, potentially allowing the server to link these requests together. More formally, the size of the user anonymity set is inversely proportional to the size of the clock drift. We bound this leakage by introducing a new assumption, namely that every party's clock is at least weakly synchronized. This is a reasonable assumption in practice.

Adding anonymity against malicious servers. There are four main attacks that a malicious server can perform. They can tamper with the suspected user's ciphertext, truncate the table to exclude the suspected user, tamper with the ciphertext of non-suspected users, and roll back the table (or a subset of ciphertexts) to a previous state. We describe and patch each of these below.

Redeeming when the ciphertext is modified. A malicious server can deanonymize the user with index idx^* by taking away all its tickets, i.e., setting $ct_{idx^*} \leftarrow$

⁶ In contrast, in the standard-strike list setting, discarding the strike-list requires all clients be online to be issued fresh tickets under new server keys.

 $\mathsf{Enc}_{\mathsf{pk}_{\mathsf{idx}^*}}(0)$. Then, when redeeming, user idx^* will download the table T, see that it cannot proceed with the protocol and abort. This abort leaks the user's index in the table to the server. We can prevent this by adding a way to detect server tampering, and then adding an *escape hatch* in the ZKP that allows a user who detected tampering to do whatever they like with their ciphertext and nullifier.

We have every user choose a MAC key mk and commit to it. We will also include all the other user secrets in the commitment, including the *source randomness* sr used to generate the user keypair, com = Com(mk||nk||sr; r), for some opening randomness r. In addition, when computing the original ct in registration, the user will MAC the ticket count $\tau = Mac_{mk}(t)$. To complete registration, the user sends (pk, com) along with a ZKP that everything was generated correctly. The server computes $ct \leftarrow Enc_{pk}(t||\tau)$, and sends back the signed commitment and position in the table $\sigma \leftarrow Sign_{sk'_{ServID}}(idx||com)$, where (pk'_{ServID}, sk'_{ServID}) is the server's keypair.⁷

Redemption is modified slightly to account for the new value in each ct. Specifically, for the user with index idx^* , each $\widehat{\operatorname{ct}}_{\operatorname{idx}}$ is computed as $\operatorname{Enc}_{\mathsf{pk}_{\operatorname{idx}}}(0||0)$ for $\operatorname{idx} \neq \operatorname{idx}^*$, and $\widehat{\operatorname{ct}}_{\operatorname{idx}^*} \leftarrow \operatorname{Enc}_{\mathsf{pk}_{\operatorname{idx}^*}}(-1||\hat{\tau}-\tau)$ where $\hat{\tau}$ is the MAC of n-1. All of this is ensured via a ZKP.

The escape hatch in the redemption ZKP is as follows. If a user observes a T containing an invalid (ct, τ) pair at index idx^* , under a MAC key whose commitment com is signed for position idx^* (as it is in registration), then the user may pick whatever they want as the new n, and may use any arbitrary nf.

Now if the server tampers with a user's ticket, it will not be able to update the MAC correctly, and will thus allow a user to use the escape hatch. In addition, if the server equivocates and gives different tables to different users, even with the user's own ciphertext intact, the users behave identically to the case where the tables are unmodified. Thus the server gains no advantage in deanonymizing.

Redeeming when the table is truncated. A malicious server can narrow down which user it is talking to by truncating its table T. For example, it can only send the first half of the table to the user. If the user's index is greater than |T|/2, it will have to abort, revealing to the server a bit of information about its index. We fix this by inserting another escape hatch in the redemption ZKP statement: if the signed index idx in σ exceeds the size of the table received, then the user can use any arbitrary **nf** as long as it does not maul anyone else's ciphertext, i.e., $\Delta = (0, \ldots, 0)$.

Redeeming when the table is rolled back. The scheme is now secure against mauled and truncated tables T, but it gives users no way to reject valid T that are simply old, i.e., a state rollback. A malicious server can use rollback to break anonymity of honest clients who do not keep track of their counter locally.

⁷ This keypair is separate from $(\mathsf{pk}_{\mathsf{ServID}}, \mathsf{sk}_{\mathsf{ServID}})$. This prevents users from picking their own table T', letting m = T' in redemption, receiving a signature, and using the knowledge of the signed T' to use the escape hatch indefinitely.

Suppose there are just two clients in the system, Alice and Bob. A malicious server may persistently roll back the counter for Bob, and leave Alice's counter intact. After a point, Alice is out of tickets, but Bob can keep redeeming. Thus, the server can identify Bob.

Fortunately, fixing this requires no additional cryptography. Observe that rolling back to a previous valid state merely gives the user more tickets. In order to preserve their anonymity, then, an honest user need only track their own state and behave as if they are receiving the most up-to-date T. Importantly, the user should stop redeeming when they should have hit their limit. If every honest user behaves this way, then the adversary has no way to distinguish between them via rollbacks. This fix applies equally to all of the escape hatch solutions discussed so far.

Generalizing past counters to arbitrary state. We note that the above techniques do not rely on the fact that the data stored in T is an integer. We can easily prove any state update function $\mathsf{st}_i := f(\mathsf{st}_{i-1}, \mathsf{auxinput})$ by uploading a value $\Delta = \mathsf{st}_i - \mathsf{st}_{i-1}$. This system is secure up to the leakage of f (and ep'). But we encounter an interesting subtlety: as described above, our approach allows rollback to valid states, but prevents any kind of selective failure attack. What functions f are safe to run in this environment?

In the case of tickets, rollback is acceptable. Server malfeasance only gives the client the option to perform more actions. An honest client who tracks their own state locally can simply not take it, thus avoiding deanonymizing themselves. Other protocols may also have this property.

What if running f on a rolled back state is not safe? So far we have considered f which are safe to run directly on a past state, i.e., $f(\mathbf{st}_{-i})$. We extend our protocol to support f which, to make progress, must be iteratively applied to old states, i.e., $f^i(\mathbf{st}_{-i})$, in essence, *fast forwarding* to the correct state. To hide whether or not a fast forward occurred, we make use of recursive SNARK constructions for incremental verifiable computation (IVC). [4, 35, 15, 7, 9]

Thus we support both functions that can safely be run on rolled-back state, and functions that are unsafe to roll back, but safe for malicious clients to roll forward arbitrarily. Readers will note, however, that not all functions fall into these two categories. Consider a single-player game where users play once per day but compete on wins (e.g., Wordle). The function that describes this game cannot be safely fast-forwarded, as a player could pad their win count. Nor can we necessarily safely force the player to play from a rolled-back game state, as their score would change. But the restrictions here are subtle and what applications fit this setup is an open question. For example, if we remove the daily play limit or play a slightly different game where the daily limit stems on signed external events (e.g., the outcome of sporting events for a gambling game), then this appears safe.

1.3 Extensions and Future Work

As described, our schemes require clients to download and then re-upload O(N) ciphertexts, each encrypting a message of length $O(\log(t))$ (for anonymous tickets) or O(s) (for outsourced state). Is it possible to do better? Here we consider avenues for reducing client download and upload.

To reduce download size, the client can utilize single-server Private Information Retrieval (PIR) to download their record instead of the full vector of accounts. This introduces one additional complication: the client needs to now prove their retrieved ciphertext is authentic without the rest of the account vector. Our setting is slightly different from that of authenticated PIR [17] because our outsourced anonymous state scheme already deals with selective failure attacks, we need not design a PIR scheme that prevents them. However, we need both the client's queried index and the server's response to be verifiable in a zeroknowledge proof. It's not sufficient for the server to merely sign the PIR result. as the client could deliberately query the wrong index and then use the signed response value as an escape hatch. If one constructs a single server PIR scheme in the style of Kushilevitz and Ostrovsky [27] from additively homomorphic encryption, a plausible protocol materializes: the server's response shall include a one-time signature over the PIR result, the client's query, and the client's PIR encryption public key. Assuming the public key commits to a unique secret key, then both the index the client queried and the result should be verifiable. The server can verify this proof with respect to the signing key and, because the key is single use, conclude the data is fresh.

Reducing client upload poses a more challenging problem. We need to avoid revealing which record on the server we are modifying. Our scheme does this by giving an encrypted Δ to be added to every account, enforcing that all but one of the values is 0. This seemingly requires N ciphertexts.

Non-interactive Anonymous Router [36] offers a surprising solution to this problem. In this scheme, an untrusted server receives constant-size ciphertexts and routes them to a a recipient without learning the destination. The authors give a special case of this for non-interactive anonymous shuffle where the server and all recipients are the same party but learn only a shuffle of the input messages. If we set the input message to be the encrypted Δ_i and the location of the ciphertext it should be added to, we achieve something resembling an anonymous tickets functionality with constant client upload bandwidth. Note some important caveats are that current versions of non-interactive anonymous routing require a trusted setup or all parties to participate in setup. In addition, every party must submit a message in each epoch. It's an open question if future work can remove some of these requirements.

Reissuing tokens or initial states. Our scheme assumes clients are issued a single set of tokens. What happens when they run out? A simple way to reissue tokens is for clients to register again under a different (potentially linked) public key. However, the server would be forced to store state for existing clients as well as new ones. If we expire tokens and force all clients to re-register periodically, then

the server's state is again optimal. But forcing all clients to synchronize by some expiry deadline is undesirable in many cases. Is it possible to avoid this cost and allow clients asynchronously (subject to whatever authorization constraints the server wishes) either get more tokens or re-register? As the server can always re-run the registration process for an individual client overwrite their existing account, this seems plausible. However, careful consideration of the setting is necessarily, especially how to surface the pre-requisite for any anonymity system : that other honest users are available.

1.4 Related Work

Typical multiparty computation protocols achieving security in the Ideal/Real Model paradigm are inherently non-anonymous (protocol design in the Ideal/Real Model implicitly assumes point-to-point authenticated channels [2]) and require all parties to be online simultaneously and to exchange messages throughout the protocol execution (e.g. using authenticated/unauthenticated Byzantine agreement to implement a broadcast channel [29]). Thus, these protocols would not satisfy our requirements of anonymity and zero communication between clients.

Exceptions to the above paradigm include the works of Halevi et al. [24] and Halevi et al. [22] who studied collusion-resistant MPC protocols for general functionalities with restricted interaction. Their model, however, does not guarantee anonymity (specifically, a PKI and authenticated point-to-point channels are assumed), clients interact with the server in a specified order, and each client is only allowed to interact with the server a single time. More general interaction patterns as well as protocols based on correlated randomness setup have also been studied [3, 23].

In the garbled-circuit-based 2PC setting, a similar phenomenon (called the selective input attack) as the one we encounter in this work occurs: Party P_2 may detect that party P_1 cheated, but aborting the protocol leaks information to P_1 , since it indicates that P_1 's misbehavior was, in fact, detected by P_2 . This problem was solved by Lindell [28] and Huang et al. [25] using a technique somewhat reminiscent of ours: If P_2 detects misbehavior of P_1 , then that in and of itself provides P_2 with an escape hatch for a second secure computation in which it directly receives P_1 's input. Despite the similarities in the high-level idea, we note that our escape hatch approach applies in a completely different setting.

A closely related line of work is anonymous credentials. There has been a long line of work on this. Anonymous *n*-times tokens/authentication focuses on a specific aspect of anonymous credentials: how to prove that possession of credentials to a relying party anonymously, a bounded number of times [11]. There has been other blind-signature and MAC-token based approaches to this problem [1, 37, 32, 26, 13], but all of them take the strike-list approach. Finally, we note that, most of the formal definitions in anonymous credentials are property based, in contrast to our idea/real paradigm. The few exception to these are [10, 8, 5].

Another interesting line of work considers PIR specifically in the context of selective failure attacks and malicious servers PIR [17, 40]. Surprisingly, despite requiring only the retrieval data, not the update of it, this setting appears to be more challenging than the anonymous outsourced ticket- and state-keeping setting. In particular, to achieve security against both selective failure attacks and equivocation attacks, the single server PIR scheme in [17] requires some gossip mechanism fora digest of the database. In contrast, because we outsource state-keeping but allow the client to retain a local copy of its state, we can safely constrain client behavior to be the same regardless of which database a malicious server provides.

2 Preliminaries

For formal definitions of additively homomorphic encryption (AHE) scheme, message authentication code, digital signature, commitment scheme, and noninteractive zero-knowledge proofs (NIZK), see Appendix A.

3 Concurrent Anonymous Tickets

We first present the ideal functionality and then describe our protocol that realizes the functionality.

3.1 Ideal Functionality

We present our ideal functionality (Fig. 1) for the issuance of anonymous tickets.

Input/Output. The functionality takes as input a keypair $(pk_{ServID}, sk_{ServID})$ belonging to the server. We also assume all users know the server's public key pk_{ServID} beforehand. In practice, this can be achieved through Public Key Infrastructure (PKI). During the Register phase, a user non-anonymously signs up and is allocated an initial quota of t tickets. Throughout the Redeem phase, a user seeking to redeem a ticket receives as output a signature from the server over a message of the user's choice, so long as the user has not already redeemed all t tickets. In more detail, in order to enforce a limit of no more than t ticket redemptions per user, the functionality maintains an individual counter, denoted n_{uid} for user uid, which tracks the number of unused tickets.

Anonymous redeem. To preserve the anonymity of redeeming users, the functionality only sends the request contents, not the user identity, to the adversary \mathcal{S} . While the corrupt server is unable to identify a particular redeeming user behind a redeem request, our network model makes no assumption that users have direct communication with each other. This makes achieving complete robustness in the presence of a corrupt server impossible. To address this limitation in the ideal functionality, we allow the adversary \mathcal{S} controlling the server to reject any Redeem request. However, we emphasize that the server cannot selectively target an individual user for rejection, since they are given no user-identifying information on which to base their decision.

Functionality \mathcal{F}_{cat}^t

Initialize:	On message	(Initialize, pk _{Serv}	$_{ID},sk_{ServID})$	from	ServID	and
messag	es (Initialize, p	(k _{ServID} ,) from us	ers, set HU	sers, C	Users :=	= {},
ep := 0), and $T := \{\}$					

- **Register:** On message (Register, ServID) from uid:
 - 1. If ServID is corrupt, send (Register, uid) to S, if S replies with (uid, abort), send \perp to uid. Otherwise, if S replies with (uid, ok), continue.
 - 2. If uid \in HUsers \cup CUsers or ep \neq 0, send \perp to uid skip the remaining steps.
 - Else if uid is corrupt, add uid to CUsers and send idx_{uid} := |HUsers∪CUsers| to S. Else, add uid to HUsers if uid is honest.
 Initialize n_{uid} to t.
- **EndRegister:** On message (EndRegister) from ServID, set the epoch ep := 1 and initialize $U_{ep} := \{\}$ to keep track of redeeming users in the new epoch. Also, send ep to S.
- **Redeem:** On message (Redeem, ep', m, uid) from uid':
 - 1. If $uid' \in HUsers$, which implies uid := uid',
 - (a) If uid $\notin U_{ep}$ and $n_{uid} > 0$, add uid to U_{ep} , set $n_{uid} := n_{uid} 1$. Else, skip the remaining steps.
 - (b) If ServID is honest and ep' = ep, set $\sigma_m \leftarrow \text{Sign}_{\mathsf{sk}_{\mathsf{ServID}}}(m)$ and R := 1. Else, set R := 0.
 - (c) If ServID is corrupt, send (Redeem, ep', m) to the adversary S. Upon receiving σ' from S, set σ_m := σ' and R := 1. Upon receiving abort from S, set R := 0.
 - (d) If R = 1, send σ_m to uid. Otherwise, send \perp to uid.
 - 2. If uid' \in CUsers,
 - (a) If uid \neq uid' and uid \in HUsers, or ep' \neq ep, send \perp to uid' and skip the remaining steps.
 - (b) If uid $\notin U_{ep}$ and $n_{uid} > 0$, add uid to U_{ep} , set $n_{uid} := n_{uid} 1$ and send $\sigma_m \leftarrow \mathsf{Sign}_{\mathsf{sk}_{\mathsf{ServID}}}(m)$ to uid'.
 - (c) Else, send \perp to uid'.

UpdateEpoch: On message (UpdateEpoch) from ServID, set ep := ep + 1 and initialize $U_{ep} := \{\}$ to keep track of redeeming users in the new epoch. Also, send ep to S.

Fig. 1. The ideal functionality for concurrent anonymous ticket

Concurrency. In practice, the server may receive multiple redemption requests from different users within a short span of time. To enhance efficiency, it would be beneficial to enable users to concurrently redeem their tickets. In our protocol, the server maintains state keeping track of the remaining tickets of all users. Allowing concurrent redemptions would imply that users are permitted to update with respect to out-of-date states, possibly allowing a malicious user to overredeem in a single concurrent burst. To prevent this, we divide the redeem phase into *epochs*, denoted **ep**, and restrict each user to redeemming no more than once per epoch. Moreover, the server rerandomizes the table at the end of each epoch.

The ideal functionality enforces this rate-limiting constraint by maintaining a record U_{ep} of users who have made a Redeem request within the current epoch, and rejecting repeats.

Additional leakage on Redeem timing. The constraint mentioned above implies that the redeeming party must be in the same epoch as the server for successful redemption. In practical scenarios, there is a possibility of epoch mismatch between a server and a user due to loose synchronization. In such cases, we contend that the server is able to learn additional information on the perceived epoch of the redeeming user. For instance, a lagging redeeming user may attempt to redeem using outdated epochs, and this potentially allows the server to associate these attempts. Consequently, we account for this information leakage in our ideal functionality by also revealing the perceived epoch of the redeeming user.

3.2 Protocol

We describe our protocol (Figs. 2 and 3) which realizes the anonymous tickets functionality (Fig. 1), allowing users to anonymously redeem acquire up to t signatures from the server.

Initialize. Our protocol starts with the server, denoted ServID, holding a keypair $(pk_{ServID}, sk_{ServID})$ and user, denoted by their ID uid, all of whom know pk_{ServID} . This keypair is used by the server to sign a user's message *m* during the Redeem phase.

Additionally, the server samples a second keypair $(pk'_{ServID}, sk'_{ServID})$ and all users get pk'_{ServID} . Unlike the previous keypair, this keypair is generated and used solely within the Register phase to endorse a user's registration. Here we make an additional setup assumption that this keypair is generated by the server with the public key transmitted to all clients at the beginning of the protocol.

On startup, each user samples a MAC key $\mathsf{mk}_{\mathsf{uid}}$, a nullifier key $\mathsf{nk}_{\mathsf{uid}}$ and a keypair $(\mathsf{pk}_{\mathsf{uid}},\mathsf{sk}_{\mathsf{uid}})$ for an additive homomorphic PKE scheme. Jumping ahead, the counter for the remaining ticket is kept on the server as a ciphertext encrypted under $\mathsf{pk}_{\mathsf{uid}}$, and the MAC key allows the user to deter the malicious server from tampering with this ciphertext. The nullifier key produces a unique string for a user within an epoch, allowing the server to identify a repeated redeeming request from the same user within a single epoch.

Register. In this phase, each user computes and sends a commitment of its keys generated in the previous phase to ServID. This commitment effectively binds the user to these keys throughout the protocol. Also, each user computes a MAC, denoted as $\tau_{\text{uid},t}$, for the initial ticket number t. Subsequently, it sends this MAC along with its public key $\mathsf{pk}_{\mathsf{uid}}$ to the server. This allows the server to compute an encryption of $t \| \tau_{\mathsf{uid},t}$ and adds the resulting ciphertext to the ServID.T table. Finally, the server uses $\mathsf{sk}'_{\mathsf{ServID}}$ to sign the commitment and the location idx of this ciphertext in $\mathsf{ServID}.T$. It returns both the signature σ_{uid} and idx to the user.

$\mathbf{Protocol} \varPi_{cat}^t[\mathcal{F}_{am}, \mathcal{F}_{nizk}^R]$
Initialize: At this phase, $ServID$'s has a key pair $(pk_{ServID},sk_{ServID})$ as
its input and all users have pk_{ServID} as their inputs.
1. ServID initializes ServID. $U := \{\}$, ServID. $T := \{\}$, ep $:= 0$.
 ServID samples a signing key pair (pk'_{ServID}, sk'_{ServID}) and sends pk'_{ServID} to all users.
3. Each user uid sample a MAC key mk_{uid} , nullifier key nk_{uid} , and
randomness $sr_{uid}.$ It then computes the encryption key pair
$(pk_{uid}, sk_{uid}) \leftarrow KeyGen(sr_{uid}) \text{ and set } \rho_{uid} := mk_{uid} nk_{uid} sr_{uid}.$ Register: uid follows the below steps to register with ServID.
1. uid generates randomness r_{uid} and computes com_{uid} :=
$Com(\rho_{uid}; r_{uid})$. Additionally, it computes $\tau_{uid,t} := Mac_{mk_{uid}}(t)$. It calls $\mathcal{F}_{nizk}^{R_0}$ with message (Prove , x, w) where $x :=$
$(\operatorname{com}_{\operatorname{uid}}, c, \tau_{\operatorname{uid}, t}, \operatorname{pk}_{\operatorname{uid}})$ and $w := (r_{\operatorname{uid}}, \rho_{\operatorname{uid}})$, and receives
(Proof, $\pi_{0.uid}$) proving correctness of generating com _{uid} , $\tau_{uid,t}$
and pk_{uid} . Then uid sends (Register, pk_{uid} , com_{uid} , $\pi_{0,uid}$,
to ServID.
2. If $ep \neq 0$ or there already exists uid \in ServID.U, ServID sends
\perp to uid and skip the remaining steps.
3. ServID sends (Verify, $x, \pi_{0,\text{uid}}$) with $x := (\text{com}_{\text{uid}}, c, \tau_{\text{uid},t}, pk_{\text{uid}})$
to $\mathcal{F}_{nizk}.$ If \mathcal{F}_{nizk} return False, $ServID$ outputs \perp to uid and
skips the remaining steps.
4. Otherwise, ServID adds uid to ServID.U. Also, it computes
$ct_{uid} \leftarrow Enc_{pk_{uid}}(t \ \tau_{uid,t}) \text{ and } \mathrm{adds} (pk_{uid},ct_{uid}) \text{ to } ServID.T.$
Finally, it sets $idx := ServID.T $ and generates $\sigma_{uid} \leftarrow Sign_{sk'_{ServID}}(idx com_{uid})$ and outputs (idx, σ_{uid}) to uid .
5. If Vrfv ((idx) coma σ a) returns False, uid aborts. Oth-
5. If $Vrfy_{pk'_{SerVD}}(idx \ com_{uid,\rho}, \sigma_{uid})$ returns False, uid aborts. Otherwise, uid initializes a local counter $n_{uid} := t$ to track the
number of its remaining tickets.
EndRegister: ServID sets the epoch $ep := 1$, initializes a nullifier
set $S := \{\}$ and a list $L := []$ that keeps track of the aggregated
updating ciphertexts within this epoch.
UpdateEpoch: First, if L is non-empty, ServID updates ServID. T
by homomorphically adding each ciphertext in L with the corre-
sponding ciphertext in ServID.T. Next, ServID re-randomizes all ciphertexts in ServID.T. Finally, ServID sets $ep := ep + 1$, $S := \{\}$, and $L = [1]$
and $L := []].$

Fig. 2. Concurrent anonymous tickets protocol. The default number of tickets at registration is t.

	Protocol $\Pi_{cat}^t[\mathcal{F}_{am}, \mathcal{F}_{nizk}^{\mathcal{R}}]$ (Continued)
	n: uid on input (Redeem, ep', m) does the following to acquire ver's signature on m .
	uid ignores the input if either it has already made a redeem
2.	attempt with the same ep' before, or $n_{uid} = 0$. uid uniformly picks sid and sends
	$(Send, sid, (Request, ep'), ServID) \text{ to } \mathcal{F}_{am}.$
3.	Upon receiving (sid, (Request, ep') from \mathcal{F}_{am} , ServID skips the remaining steps if $ep \neq ep'$. Otherwise, ServID sends (Reply, sid, ServID.T) to \mathcal{F}_{am} .
4.	Upon receiving (sid, ServID.T) from \mathcal{F}_{am} ,
	(a) uid tries to retrieve the idx^{th} ciphertext ct_{idx} from ServID.T. Then, it computes $n \ \tau := Dec_{sk_{uid}}(ct_{idx})$ and checks $Vrfy_{mk_{uid}}(n, \tau) = 1$.
	i. If the above step fails, uid picks a uniformly random
	nullifier nf and computes $\hat{ct}_{idx} \leftarrow Enc_{pk_{uid}}(0)$.
	ii. Otherwise, uid computes $nf := PRF_{nk_{uid}}(ep')$. Then,
	it computes $\hat{\tau} \leftarrow Mac_{mkuid}(n-1), \ \Delta := -1 \ (\hat{\tau} - \tau), \ $
	and encrypts $\hat{ct}_{idx} \leftarrow Enc_{pk_{uid}}(\Delta)$. Finally, for every remaining record (pk_j, ct_j) in ServID.T,
	uid computes $\hat{ct}_j \leftarrow Enc_{pk_j}(0)$.
	(b) Let $\hat{ct} := (\hat{ct}_1, \dots, \hat{ct}_{ ServID,T })$. uid calls $\mathcal{F}_{nizk}^{R_1}$ with message (Prove, x, w) where $x := (pk'_{ServID}, ServID, T, nf, ep', ct)$ and
	$w := (idx, \rho_{uid}, r_{uid}, com_{\rho_{uid}}, \sigma_{uid}). \text{ Receive } (Proof, \pi_1) \text{ from } \mathcal{F}_{nizk}^{R_1}.$
	(c) It sends (Send, sid, (nf, ep', \hat{ct}, π_1, m), ServID) to \mathcal{F}_{am} , where <i>m</i> is a message that it requires ServID to sign,
	and sets $n_{\text{uid}} := n_{\text{uid}} - 1$.
5.	Upon receiving $(sid, (nf, ep', ct, \pi_1, m))$ from \mathcal{F}_{am} :
	(a) ServID first checks that $nf \notin S$, $ \hat{ct} := ServID.T $,
	and $ep' = ep$. Then, it sends (Verify, x, π_1) with $x :=$
	$(pk'_{ServID}, ServID.T, nf, ep', \hat{ct})$ to $\mathcal{F}^{R_1}_{nizk}$ to verify the zero knowledge proof. If either one of the four checks fails,
	it sends (Reply, sid, \perp) to \mathcal{F}_{am} and skips the remaining
	steps.
	(b) ServID adds nf to S. If L is empty, it simply sets $L := \hat{ct}$.
	Else, it computes component-wise homomorphic addition
	of L and ct. Finally, it generates $\sigma_m \leftarrow Sign_{sk_{ServID}}(m)$ and sends (Reply , sid, σ_m) to \mathcal{F}_{am} .
6.	uid receives (sid, σ_m) from \mathcal{F}_{am} . If $Vrfy_{pk_{server}}(\sigma_m, m) \neq 1$, it
	outputs \perp . Otherwise, it outputs σ_m .

Fig. 3. Continuation of the figure for the concurrent anonymous tickets protocol. The default number of tickets at registration is t.

Functionality $\mathcal{F}_{nizk}^{\mathcal{R}}$ $\mathcal{F}_{nizk}^{\mathcal{R}}$ is parameterized by an NP relation \mathcal{R} . (The code treats \mathcal{R} as a binary function.)
Proof: On message (Prove, x, w) from party P : Ignore if $\mathcal{R}(x, w) \neq 1$.
Send (Prove, P, x) to S . Upon receiving (Proof, π) from S , store
(x, w, π) and send (Proof , π) to <i>P</i> .
Verification: On message (Verify, x, π) from a party V: if (x, π)
is stored, then return (Verification, $x, \pi, 1$) to V. Else, send
(Verify, x, π) to S . Upon receiving (Witness, w) from S , if
$R(x,w) := 1$, store (x,π) and return (Verification, $x, \pi, 1$) to V.
Else, return (Verification, $x, \pi, 0$).
= 150, 100000, 0000, 000, 00000, 0000, 0000, 0000, 0000, 0000, 0000, 0000, 0000, 0000, 0000,

Fig. 4. The ideal functionality for Single-Proof Non-Interactive Zero-Knowledge Functionality

This gives credibility to the user when it needs to prove a malicious server has tempered or deleted its record during the following redeem phase.

Note that the server does not receive the $\mathsf{mk}_{\mathsf{uid}}$ in the clear (only the commitment). This creates an issue: a malicious user could potentially create an incorrect MAC and later shift the blame to the server regarding this incorrect MAC. To resolve this issue, we rely on Non-Interactive Zero-Knowledge Proofs (NIZK): a user must prove to the server that the MAC it generates authenticates *t* using the MAC key committed. In our protocol, we assume an ideal NIZK functionality $\mathcal{F}_{\mathsf{nizk}}$ (Fig. 4). Such functionality could be implemented using any NIZK proof of knowledge system satisfying simulation-sound extractability. The formal description of the ZK relation, designated as R_0 , can be found in section 3.3.

EndRegister. During this transitional phase, the server initializes its epoch number as ep := 1. To prevent any user from redeeming more than once within this epoch, the server maintains a set, denoted as S, to record all the nullifiers it will receive within this epoch. Additionally, the server initializes an empty list, referred to as L, which is of the same size as ServID.T. This list serves the purpose of keeping all the intended modifications (in an aggregated form) to ServID.T that are expected to occur by the end of this epoch.

Functionality \mathcal{F}_{am}

Send: On message $(\mathsf{Send}, \mathsf{sid}, m_1, R)$ from S. Forward message (sid, m_1) to party R and record (S, sid) if it has not been recorded before.

Reply: On message (Reply, sid, m_2) from R, the functionality checks if there exists some S such that (S, sid) is recorded. If yes, forward message (sid, m_2) to S. Otherwise, it ignores the request.

Fig. 5. The ideal functionality for anonymous messaging

Redeem. During the Redeem phase, a user initiates the process of redeeming a ticket. This involves decrementing its encrypted ticket count stored on the server and obtaining the server's signature on a message m to validate this action.

Throughout this phase, we assume the availability of an anonymous messaging ideal functionality \mathcal{F}_{am} (Fig. 5). All communications between the server and users are passed through \mathcal{F}_{am} . This ensures that the server cannot identify users solely based on communication patterns. To redeem a ticket, the user first requests ServID.*T* from the server. Next, the user needs to decrement its encrypted counter and provide a MAC.

Recall that during an epoch, multiple users may request the same ServID.T and submit modifications at the same time. To facilitate the server's aggregation of these modifications from various parties, the user calculates $\Delta_{idx^*} :=$ $-1||(\hat{\tau} - \tau)$ and produces an updating ciphertext $\hat{ct}_{idx^*} \leftarrow \text{Enc}_{pk}(\Delta_{idx^*})$ for their own ciphertext at index idx^{*} and public key pk. Additionally, the user generates ciphertexts encrypting the value 0 for all other records, and submits them together with its own updated ciphertext to hide the exact record that is modified. The server aggregates the \hat{ct} tables received from multiple users by homomorphically summing them into the current ct table, entry by entry. Therefore, at any time, the server only maintains the aggregated version with the list L.

To convince the server that the above step is executed honestly, a user must provide a zero-knowledge proof (ZKP) demonstrating the correct generation of the updating ciphertexts. However, a user may encounter issues in generating such proof if its record is tampered with or removed. Consequently, the zeroknowledge proof includes a *escape hatch* mechanism that enables the user to prove the server's malicious action using the server's signature σ_{uid} in the registered phase as part of the witness. This takes away the server's ability to selectively manipulate specific table records in an attempt to deanonymize a Redeem request upon failure to generate a zero-knowledge proof. Effectively, it grants the user an infinite number of tickets, as the user can consistently employ the same escape hatch to produce a valid zero-knowledge proof. The formal description of the ZK relation, designated as R_1 , can be found in section 3.3.

We emphasize that we do not implement a mechanism to address server rollbacks of records. This decision is driven by the fact that, unlike tampering, a user can still generate a zero-knowledge proof even in the case of a rollback. Essentially, a rollback can only increase the user's ticket count. However, the user bears the risk of being deanonymized when redeeming these extra tickets. Consider the following attack where the server, in epoch ep, selects a single ciphertext in ServID.T and rolls it back to the ciphertext in a prior epoch ep_{old} , and observe whether this results in a greater total number of redeem requests throughout the protocol lifespan. If this occurs, it indicates that extra tickets have been effectively "granted" to the user associated with that record, implying that the user must have redeemed some tickets between epochs ep_{old} and ep. To mitigate this risk, each user maintains a local record of their remaining tickets, denoted as n_{uid} , to prevent them from over-redeeming.

ZK Relation R ₀
A valid instance of the registration relation R_0 contains a statement including:
 com_ρ: a commitment of value ρ, t: the default number of tickets, τ: a MAC of t, pk: a public key belonging to the prover,
and a witness including:
 - r: the randomness used for generating commitment com_ρ, - ρ := (mk nk sr): a concatenation of a MAC key, a nullifier key, and source randomness used to generate an encryption public/private key pair,
such that the following conditions hold:
Correct Commitment. $com_{\rho} = Com(\rho; r),$ Correct MAC. $\tau = Mac_{mk}(t),$ Correct Public Key. $pk = pk',$ where $(pk', sk') := KeyGen(sr),$

Fig. 6. ZK relation R_0 for ticket registration.

UpdateEpoch. During this transitional phase, the server increments its epoch number as ep := ep + 1 and empties the nullifier set. Further, the server updates ServID.T as mentioned above, by homomorphically summing the current ciphertext table with all the ciphertexts in L. Afterward, it empties L.

3.3 ZK relations

We give the two ZK relations (Fig. 6 and 7) that appeared in our protocol (Fig. 2).

3.4 Standalone Security Proof

In this section, we prove the security of our scheme in the $(\mathcal{F}_{am}, \mathcal{F}_{nizk}^{\mathcal{R}})$ -hybrid model with static corruptions.

Theorem 3 (Concurrent Anonymous Tickets Standalone Security). Protocol $\Pi_{cat}[\mathcal{F}_{am}, \mathcal{F}_{nizk}^{\mathcal{R}}]$ securely realizes \mathcal{F}_{cat}^{t} with abort in the presence of static malicious adversaries in $\mathcal{F}_{am}, \mathcal{F}_{nizk}^{\mathcal{R}}$ -hybrid model with server storage and per redeem communication of O(N) ciphertexts, each encrypting a message of length $O(\log(t))$, where N is the number of users, and t is the number of tickets issued per user.

We defer the proof to appendix B.

ZK Relation R_1 A valid instance of the redemption relation R_1 contains a statement

- pk_{ServID} : a signature public key of the server used to endorse a correct registration,
- $\{(\mathsf{pk}_i, \mathsf{ct}_j)\}_{j \in [N']}$: a table of N' encryption public key and ciphertexts (each belonging to a user) stored by the server,
- nf: a nullifier used to uniquely bind to a user within an epoch,
- ep: the current epoch number,
- ${\hat{ct}_j}_{j \in [N']}$: a list of updating ciphertexts,

and a witness including:

including:

- idx: a user's position in ServID.T,
- $-\rho := (\mathsf{mk} ||\mathsf{nk} ||\mathsf{sr})$: a value that is a concatenation of a MAC key, a nullifier key, and source randomness used to generate a public/private key pair,
- -r: the randomness used for generating commitment com_{ρ} ,
- $\operatorname{com}_{\rho}$: a commitment of value ρ ,
- $-\sigma$: server's signature on idx $\|com_{\rho}$, provided in the register phase,

such that the following conditions hold:

Correct Commitment. $com_{\rho} = Com(\rho; r).$ Correct Signature. $\operatorname{Verify}_{\mathsf{pk}_{\mathsf{ServID}}}(\operatorname{idx} \| \operatorname{com}_{\rho}, \sigma) = 1.$ Correct Rerandomization. $\forall j \neq idx, \hat{ct}_i$ are encryptions of 0 under pk_i .

and at least one of the following three conditions hold:

- Missing Record. idx > N', this suggests the table size is smaller than the user's index.
- Tampered Record. $\tau \neq Mac_{mk}(n)$, where (pk, sk) := KeyGen(sr) and $(n \| \tau) := \mathsf{Dec}_{\mathsf{sk}}(\mathsf{ct}_{\mathsf{idx}}).$
- *Correct Redeem.* Let $(\mathsf{pk}, \mathsf{sk}) := \mathsf{KeyGen}(\mathsf{sr})$ and $(n \| \tau) := \mathsf{Dec}_{\mathsf{sk}}(\mathsf{ct}_{\mathsf{idx}})$, all conditions below hold:
 - -n > 0, which suggests the user has remaining tickets.
 - $-\hat{ct}_{idx}$ is an encryption of $(-1||Mac_{mk}(n-1) \tau)$ under pk, which shows the user correctly decrement its ticket with a correct MAC.
 - $nf = PRF_{nk}(ep)$, which suggests a correctly generated nullifier.

Fig. 7. ZK relation R_1 for ticket redemption.

4 Extension to Anonymous Outsourced State-keeping

We extend our techniques to support arbitrary states. In particular, rather than monotonically decrementing the state, we consider any arbitrary state transition function that additionally takes an input y and a current st and outputs the next state st'.

4.1 Summary of Changes

In our scenario, it might be advantageous to disclose a portion of the input to have the server to sign it, confirming the validity of a state transition that meets certain conditions or constraints. Consequently, we use y_1 to denote the *public* input that is revealed to the server, while y_2 denotes the *private* input that must remain hidden from the server.

Formally, we define the state transition function using the quadruple (Y_1, Y_2, D, f) , where:

- $-Y_1$ is the domain of public input.
- $-Y_2$ is the domain of private input.
- D is the domain of states.
- f is the state transition function: $f: D \times Y_1 \times Y_2 \to D$.

Specifically, $f(\mathsf{st}, y_1, y_2)$ returns the output of a single state transition on state $\mathsf{st} \in D$ with inputs $y_1 \in Y_1, y_2 \in Y_2$. Notice the counter in the previous section is a special case where $Y_1, Y_2 = \{\bot\}$ and $f(\mathsf{st}, \cdot, \cdot) := \mathsf{st} - 1$ if $\mathsf{st} > 0$ and \bot otherwise.

Handling rollback. Recall that in the previous section, the server may roll back a user's record to a previous one. However, as this merely gives the user more tickets, we handled it by simply letting the user carry out the computation using the rollback state, subject to the constraint that the total number of user's redemption does not exceed the maximum number of tickets t.

On the other hand, when dealing with arbitrary state transition functions, rollback attacks may pose a greater threat to anonymity, due to the exposure of y_1 to the server and the (potential) dependency between y_1 and st. For example, consider a state transition function where y_1 assumes a unique value during the initial transition, and a malicious server has a high level of confidence that all users have updated their state at least once after a few epochs. In such a situation, if the malicious server rollbacks a user's record, it can deanonymize the subsequent redemption request made by that user. This is because the user is now forced to use the unique y_1 value again in order to move out of the initial state.

To resolve this, we essentially allow a user to *fast-forward* their rollback state. In more detail, if a user detects their state is outdated, they can simply "fix" the state to the up-to-date version that they maintain locally. Subsequently, the user can continue the redemption request based on the up-to-date state. This introduces an additional component in the ZK relation to enable users to prove they correctly make multiple transitions to the up-to-date state followed by a correct state transition. We provide a detailed description of our modified ZK relation in Section 4.4.

It is worth mentioning that our solution also permits a malicious user to advance their state an arbitrary number of times. We capture such a behavior in our ideal functionality (Fig. 13) as FastForward. We note that many functions of interest exhibit a form of monotonicity, where future states tend to have diminishing value. This naturally discourages the misuse of arbitrary state advancement. For instance, in the ticket redemption scenario discussed in the previous section, a user fast-forwarding its state would result in a loss of their tickets. However, in cases where the function itself does not inherently possess this kind of monotonicity, alternatives may be possible. For example, we can limit the total number of state advances a client is allowed to make , but we cannot constraint when they make them: if a client is allowed to advance its state ten times over 10 redeems, then it can also advance 10 times in the first redeem, but advance no further.

4.2 Ideal Functionality

Compared to the anonymous tickets ideal functionality, we substitute all instances of ticket count n with the generic state variable st and introduce the notations f, y_1 , and y_2 to describe the state transitions. Furthermore, we add a new method *FastForward* to allow a malicious user to fast forward its own state. Due to space limits, we defer the figure to Appendix C.

4.3 Protocol

We describe our protocol in Figs. 8 and 9 with the changes from the anonymous ticket protocol highlighted in blue. Similar to our modification to the ideal functionality, we substitute all instances of n with the generic state variable st and introduce the notation f, y_1 , and y_2 to describe the state transitions. Moreover, in Redeem steps 4.a.ii and 4.a.iii, we show how a user can make a normal state transition and a fast-forward state transition respectively.

4.4 ZK Relations

We give the ZK relation for redemption (Fig. 10) that appeared in our protocol. This relation includes the ability to fast forward invocations of f if the server rolls back. As before, we highlight the changes compared to the anonymous tickets counterparts in blue color. We use the notation $\mathbf{y}_1 \in Y_1^n, \mathbf{y}_2 \in Y_2^n$ and n', where $f^{n'}(\mathsf{st}, \mathbf{y}_1, \mathbf{y}_2)$ denote the n' sequential applications of f using the *last* n' *inputs* in $\mathbf{y}_1, \mathbf{y}_2$ respectively. The ZK relation for registration is rather similar to its counterpart for anonymous tickets, therefore we defer it to Appendix C.

$\mathbf{Protocol} \varPi_{cas}^{f,st_0}[\mathcal{F}_{am},\mathcal{F}_{nizk}^R]$		
Initialize: At this phase, ServID's has a key pair $(pk_{ServID}, sk_{ServID})$ as		
its input and all users have pk_{ServID} as their inputs.		
1. ServID initializes ServID. $U := \{\}$, ServID. $T := \{\}$, ep := 0.		
2. ServID samples a signing key pair $(pk'_{ServID}, sk'_{ServID})$ and sends		
pk _{ServID} to all users.		
3. Each user uid sample a MAC key mk_{uid} , nullifier key nk_{uid} , and		
randomness sr_{uid} . It then computes the encryption key pair		
$(pk_{uid},sk_{uid}) \leftarrow KeyGen(sr_{uid}) \text{ and set } \rho_{uid} := mk_{uid} \ nk_{uid}\ sr_{uid}.$		
Register: uid follows the below steps to register with ServID.		
1. uid generates randomness r_{uid} and computes $com_{uid} :=$		
$Com(\rho_{uid}; r_{uid})$. Additionally, it generates randomness r'_{uid} and		
compute $\tau_{uid,0}$:= Mac _{mkuid} (st ₀). It calls $\mathcal{F}_{nizk}^{R_0}$ with mes-		
sage (Prove, x, w) where $x := (com_{uid}, st_0, \tau_{uid,0}, pk_{uid})$ and		
$w := (r_{uid}, \rho_{uid})$, and receives (Proof, $\pi_{0,uid}$) proving correct-		
ness of generating com_{uid} , $\tau_{uid,0}$ and pk_{uid} . Then uid sends		
(Register, $pk_{uid}, com_{uid}, \tau_{uid,0}, \pi_{0,uid})$ to ServID.		
2. If $ep \neq 0$ or there already exists uid \in ServID.U, ServID sends		
\perp to uid and skip the remaining steps.		
3. ServID sends (Verify, $x, \pi_{0, \text{uid}}$) with $x :=$		
$(com_{uid}, st_0, \tau_{uid,0}, pk_{uid})$ to \mathcal{F}_{nizk} . If \mathcal{F}_{nizk} return False,		
ServID outputs \perp to uid and skips the remaining steps.		
4. Otherwise, ServID adds uid to ServID. U . Also, it computes		
$ct_{uid} \leftarrow Enc_{pk_{uid}}(st_0 \ \tau_{uid,0}) \text{ and } \mathrm{adds} \ (pk_{uid},ct_{uid}) \text{ to } ServID.T.$		
Finally, it sets idx := $ \text{ServID}.T $ and generates $\sigma_{\text{uid}} \leftarrow$		
$Sign_{sk'_{ServID}}(idx \ com_{uid}) \text{ and outputs } (idx, \sigma_{uid}) \text{ to uid.}$		
5. If $Verify_{pk'_{ServID}}(idx \ com_{uid,\rho}, \sigma_{uid})$ returns False, uid aborts.		
Otherwise, uid initializes $st_{uid} := st_0$ to track its current state		
and lists $\boldsymbol{y}_1, \boldsymbol{y}_2 := []$ to keep track of all inputs y_1, y_2 used in		
state transition function.		
EndRegister: ServID sets the epoch $ep := 1$, initializes a nullifier		
set $S := \{\}$ and a list $L := []$ that keeps track of the aggregated		
updating ciphertexts within this epoch.		
UpdateEpoch: First, if L is non-empty, ServID updates ServID. T		
by homomorphically adding each ciphertext in L with the corre-		
sponding ciphertext in ServID. T . Next, ServID re-randomizes all		
ciphertexts in ServID.T. Finally, ServID sets $ep := ep + 1$, $S := \{\}$,		
and $L := \{\}$.		

Fig. 8. Concurrent anonymous outsourced state-keeping protocol parameterized by a state transition function f, and a default state st_0 .

Protocol $\Pi_{cas}^{f,st_0}[\mathcal{F}_{am}, \mathcal{F}_{nizk}^{\mathcal{R}}]$ (Continued)
Redeem: uid on input (Redeem, ep', y_1, y_2, m) does the following to acquire server's signature on $y_1 m$.
1. uid ignores the input if one of the following holds: it has al-
ready made a redeem attempt with the same ep' before, or
$f(st_{uid},y_1,y_2):=\perp.$
2. uid uniformly picks sid and sends
(Send, sid, (Request, ep'), ServID) to \mathcal{F}_{am} .
3. Upon receiving $(sid, (Request, ep')$ from \mathcal{F}_{am} , ServID skip
the remaining steps if $ep \neq ep'$. Otherwise, ServID sends
$(Reply,sid,ServID.T)$ to $\mathcal{F}_{am}.$
4. Upon receiving (sid, ServID.T) from \mathcal{F}_{am} ,
(a) uid tries to retrieve the idx^{th} ciphertext ct_{idx} from
ServID.T. Then, it computes $st \ \tau := Dec_{sk_{uid}}(ct_{idx})$ and
checks $Verify_{mk_{uid}}(st,\tau) = 1.$
i. If the above step fails, uid picks a uniformly random
nullifier nf and computes $\hat{ct}_{idx} \leftarrow Enc_{pk_{uid}}(0)$.
ii. Else if st = st _{uid} , uid computes $nf := PRF_{nk_{uid}}(ep')$
and $\hat{st} := f(st_{uid}, y_1, y_2)$. Then, it computes $\hat{\tau} =$
$Mac_{mk_{uid}}(\hat{st}), \ \varDelta \ := \ (\hat{st} - st) \ (\hat{\tau} - \tau), \ \mathrm{and} \ \hat{ct}_{idx} \ \leftarrow$
$Enc_{pk_{uid}}(\Delta).$
iii. Else, uid computes $nf := PRF_{nk_{uid}}(ep')$ and $\hat{st} :=$
$f(st_{uid}, y_1, y_2)$. Next, it sets n' to be the number of
transitions needed in order to reach st_{uid} from st .
Then, it computes $\hat{\tau} \leftarrow Mac_{mk_{uid}}(\hat{st}), \ \varDelta := (\hat{st} - $
$st) \ (\hat{\tau} - au), \hat{ct}_{idx} \leftarrow Enc_{pk_{uid}}(\Delta).$
Finally, for every remaining record (pk_j, ct_j) in ServID.T,
uid computes $\hat{ct}_j \leftarrow Enc_{pk_j}(0)$.
(b) Let $\hat{ct} := (\hat{ct}_1, \dots, \hat{ct}_{ ServID.T })$. uid calls $\mathcal{F}_{nizk}^{R_1}$ with message
(Prove, x, w) where $x := (pk'_{ServID}, ServID, rf, ep', \hat{ct}, y_1)$
and $w := (idx, \rho_{uid}, com_{\rho_{uid}}, \sigma_{uid}, n', y_2, y_1, y_2)$. Receive
(Proof, π_1) from $\mathcal{F}_{nizk}^{R_1}$.
(c) uid sends (Send, sid, (nf, ep', \hat{ct}, π_1, y_1, m), ServID) to \mathcal{F}_{am} ,
sets $\mathbf{st}_{uid} := \hat{\mathbf{st}}$, and adds y_1, y_2 to y_1, y_2 respectively.
5. Upon receiving $(sid, (nf, ep', ct, \pi_1, y_1, m))$ from \mathcal{F}_{am} :
(a) ServID first checks that $nf \notin S$, $ ct := ServID.T $,
and $ep' = ep$. Then, it sends (Verify, x, π_1) with $x :=$
$(pk'_{ServID},ServID,nf,ep',ct,y_1)$ to $\mathcal{F}_{nizk}^{R_1}$ to verify the zero
knowledge proof. If either one of the four checks fails, it
sends (Reply , sid, \perp) to \mathcal{F}_{am} and skips the remaining steps.
(b) ServID adds of to S. If L is empty, it simply sets
$L := \hat{ct}$. Else, it computes component-wise homomor-
phic addition of L and \hat{ct} . Finally, it generates $\sigma_{y_1,m} \leftarrow$
Find a control of D and ct. Finally, it generates $\theta_{y_1,m} \leftarrow$
Sign _{skServID} $(y_1 m)$ and sends (Reply, sid, $\sigma_{y_1,m}$) to \mathcal{F}_{am} . 6. uid receives (sid, $\sigma_{y_1,m}$) from \mathcal{F}_{am} . If
Verify _{pkServiD} $(\sigma_{y_1,m}, y_1 m) \neq 1$, it outputs \perp . Otherwise, it outputs σ
it outputs σ_m .

٦

Fig. 9. Continuation of the figure for the concurrent anonymous outsourced state-keeping protocol parameterized by a state transition function f, and a default state $\mathsf{st}_0.$

ZK Relation R_1

A valid instance of the redemption relation R_1 contains a statement including:

- $-\ pk_{ServID}$: a signature public key of the server, which is used to endorse a correct registration from the user,
- $\{(\mathsf{pk}_j, \mathsf{ct}_j)\}_{j \in [N']}$: a table of N' encryption public key and ciphertexts (each belonging to a user) stored by the server,
- nf: a nullifier used to uniquely bind to a user within an epoch,
- ep: the current epoch number,
- ${\hat{\mathsf{ct}}_j}_{j \in [N']}$: a list of updating ciphertexts,
- $-y_1$: the public input of the current state transition

and a witness including:

- idx: a user's position in ServID.T,
- $-\rho := (\mathsf{mk}||\mathsf{nk}||\mathsf{sr})$: a value that is a concatenation of a MAC key, a nullifier key, and source randomness used to generate a public/private key pair,
- -r: the randomness used for generating commitment com_{ρ} ,
- $\operatorname{com}_{\rho}$: a commitment of value ρ ,
- $-\sigma$: server's signature on idx $\|com_{\rho}$, provided in the register phase,
- $-y_2$: the private input of the current state transition,
- -n': number of additional state transitions before the current state transition,
- y_1 : a list of public inputs used in the previous transitions,
- $-y_2$: a list of private inputs used in the previous transitions,

such that the following conditions hold:

Correct Commitment. $\operatorname{com}_{\rho} = \operatorname{Com}(\rho; r)$. Correct Signature. $\operatorname{Vrfypk}_{\operatorname{ServID}}(\operatorname{idx} \| \operatorname{com}_{\rho}, \sigma) = 1$. Correct Rerandomization. $\forall j \neq \operatorname{idx}, \widehat{\operatorname{ct}}_{j}$ are encryptions of 0 under pk_{j} .

and at least one of the following four conditions hold:

Missing Record. idx > N', this suggests the table size is smaller than the user's index.

Tampered Record. $\tau \neq Mac_{mk}(st)$, where (pk, sk) := KeyGen(sr) and $(st||\tau) := Dec_{sk}(ct_{idx})$.

Correct Redeem. Let (pk, sk) := KeyGen(sr) and $(st || \tau) := Dec_{sk}(ct_{idx})$, all conditions below hold:

- $-f(\mathsf{st}, y_1, y_2) \neq \perp$, which suggests the transition is valid.
- \hat{ct}_{idx} is an encryption of $(f(st, y_1, y_2))$
 - st $\|\text{Mac}_{mk}(f(\text{st}, y_1, y_2)) \tau)$ under pk, which shows the user correctly applies the transition function with inputs y_1, y_2 and generates a correct MAC.

 $- nf = PRF_{nk}(ep)$, which suggests a correctly generated nullifier. Correct Redeem with Fast Forward. Let (pk, sk) := KeyGen(sr) and $(st || \tau) := Dec_{sk}(ct_{idx})$, all conditions below hold:

- $-f(f^{n'}(\mathbf{st}, \mathbf{y_1}, \mathbf{y_2}), y_1, y_2) \neq \perp$, which suggests the transitions are valid.
- Let $\hat{\mathbf{st}} := f(f^{n'}(\mathbf{st}, \boldsymbol{y_1}, \boldsymbol{y_2}), y_1, y_2)$. $\hat{\mathbf{ct}}_{\mathsf{idx}}$ is an encryption of $(\hat{\mathbf{st}}_{\mathsf{T}} - \mathbf{st} \| \mathsf{Mac}_{\mathsf{mk}}(\hat{\mathbf{st}}) - \tau)$ under pk , which shows the user correctly applies multiple transitions function with inputs $\boldsymbol{y_1}, \boldsymbol{y_2}, y_1, y_2$ and generates a correct MAC.
- $nf = PRF_{nk}(ep)$, which suggests a correctly generated nullifier.

4.5 Standalone Security Proof

We prove the security of our scheme in the $(\mathcal{F}_{am}, \mathcal{F}_{nizk}^{\mathcal{R}})$ -hybrid model with static corruptions.

Theorem 4 (Concurrent Anonymous State Transition Standalone Security). Protocol $\Pi_{cas}^{f,st_0}[\mathcal{F}_{am}, \mathcal{F}_{nizk}^{\mathcal{R}}]$ securely realizes $\mathcal{F}_{cas}^{f,st_0}$ with abort in the presence of static malicious adversaries in $\mathcal{F}_{am}, \mathcal{F}_{nizk}^{\mathcal{R}}$ -hybrid model with server storage and per redeem communication of O(N) ciphertexts, each encrypting a message of length O(s), where N is the number of users, and s is the size of the client's state.

The proof of this Theorem is given in Appendix D.

5 Conclusion

In this paper, we construct functionalities for anonymous tickets and anonymous outsourced state-keeping with optimal storage. We achieve the lower bound of $O(N \log(t))$ storage for the ticket setting under standard cryptographic assumptions, and $O(N \cdot s)$ for the outsourced state-keeping setting for restricted classes of functions. By outsourcing the client state to a server, our protocols ensure dishonest clients cannot roll back their state or equivocate. At the same time, honest clients get anonymity and confidentiality against a malicious server who equivocates or injects selective failures. We achieve these results without requiring byzantine broadcast, gossip, or that clients communicate with any other party.

The proposed functionalities offer a secure and efficient alternative to traditional strike-lists, addressing the challenges of rapid storage growth and computational overhead associated with large strike-lists. Furthermore, our approach maintains the simplicity and (nearly) the minimal round complexity characteristic of strike-lists, providing a practical alternative when the number of clients is small relative to the size of the strike-list. As our schemes require clients to download and then upload re-upload O(N) ciphertexts, the approach may not be desirable in all cases.

We leave to future work three questions. First, can the client bandwidth requirments be reduced, e.g., with non-interactive anonymous shuffles [36], in a practical setting? Second, how can additional tokens be efficiently issued once a user has run out? And third, for the outsourced state-keeping setting, what applications are viable in the fast-forward setting?

References

- Foteini Baldimtsi and Anna Lysyanskaya. Anonymous credentials light. In Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, editors, ACM CCS 2013: 20th Conference on Computer and Communications Security, pages 1087–1098, Berlin, Germany, November 4–8, 2013. ACM Press.
- Boaz Barak, Ran Canetti, Yehuda Lindell, Rafael Pass, and Tal Rabin. Secure computation without authentication. In Victor Shoup, editor, Advances in Cryptology – CRYPTO 2005, volume 3621 of Lecture Notes in Computer Science, pages 361–377, Santa Barbara, CA, USA, August 14–18, 2005. Springer, Heidelberg, Germany.
- Amos Beimel, Ariel Gabizon, Yuval Ishai, Eyal Kushilevitz, Sigurd Meldgaard, and Anat Paskin-Cherniavsky. Non-interactive secure multiparty computation. In Juan A. Garay and Rosario Gennaro, editors, *Advances in Cryptology – CRYPTO 2014, Part II*, volume 8617 of *Lecture Notes in Computer Science*, pages 387–404, Santa Barbara, CA, USA, August 17–21, 2014. Springer, Heidelberg, Germany.
- Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. Scalable zero knowledge via cycles of elliptic curves. In Juan A. Garay and Rosario Gennaro, editors, Advances in Cryptology – CRYPTO 2014, Part II, volume 8617 of Lecture Notes in Computer Science, pages 276–294, Santa Barbara, CA, USA, August 17– 21, 2014. Springer, Heidelberg, Germany.
- 5. Dmytro Bogatov, Angelo De Caro, Kaoutar Elkhiyaoui, and Björn Tackmann. Anonymous transactions with revocation and auditing in hyperledger fabric. In Mauro Conti, Marc Stevens, and Stephan Krenn, editors, CANS 21: 20th International Conference on Cryptology and Network Security, volume 13099 of Lecture Notes in Computer Science, pages 435–459, Vienna, Austria, December 13–15, 2021. Springer, Heidelberg, Germany.
- Sean Bowe, Alessandro Chiesa, Matthew Green, Ian Miers, Pratyush Mishra, and Howard Wu. ZEXE: Enabling decentralized private computation. In 2020 IEEE Symposium on Security and Privacy, pages 947–964, San Francisco, CA, USA, May 18–21, 2020. IEEE Computer Society Press.
- Sean Bowe, Jack Grigg, and Daira Hopwood. Recursive proof composition without a trusted setup. Cryptology ePrint Archive, Paper 2019/1021, 2019. https:// eprint.iacr.org/2019/1021.
- Joakim Brorsson, Bernardo David, Lorenzo Gentile, Elena Pagnin, and Paul Stankovski Wagner. Papr: Publicly auditable privacy revocation for anonymous credentials. In Mike Rosulek, editor, *Topics in Cryptology – CT-RSA 2023*, pages 163–190, Cham, 2023. Springer International Publishing.
- Benedikt Bünz, Alessandro Chiesa, William Lin, Pratyush Mishra, and Nicholas Spooner. Proof-carrying data without succinct arguments. In Tal Malkin and Chris Peikert, editors, Advances in Cryptology – CRYPTO 2021, Part I, volume 12825 of Lecture Notes in Computer Science, pages 681–710, Virtual Event, August 16–20, 2021. Springer, Heidelberg, Germany.
- 10. Jan Camenisch, Manu Drijvers, and Maria Dubovitskaya. Practical UC-secure delegatable credentials with attributes and their application to blockchain. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, ACM CCS 2017: 24th Conference on Computer and Communications Security, pages 683–699, Dallas, TX, USA, October 31 – November 2, 2017. ACM Press.

- 11. Jan Camenisch, Susan Hohenberger, Markulf Kohlweiss, Anna Lysyanskaya, and Mira Meyerovich. How to win the clonewars: Efficient periodic n-times anonymous authentication. In Ari Juels, Rebecca N. Wright, and Sabrina De Capitani di Vimercati, editors, ACM CCS 2006: 13th Conference on Computer and Communications Security, pages 201–210, Alexandria, Virginia, USA, October 30 – November 3, 2006. ACM Press.
- Melissa Chase, Apoorvaa Deshpande, Esha Ghosh, and Harjasleen Malvai. SEEMless: Secure end-to-end encrypted messaging with less trust. In Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, ACM CCS 2019: 26th Conference on Computer and Communications Security, pages 1639–1656, London, UK, November 11–15, 2019. ACM Press.
- Melissa Chase, F. Betül Durak, and Serge Vaudenay. Anonymous tokens with stronger metadata bit hiding from algebraic macs. In Helena Handschuh and Anna Lysyanskaya, editors, *Advances in Cryptology – CRYPTO 2023*, pages 418– 449, Cham, 2023. Springer Nature Switzerland.
- David Chaum. Blind signatures for untraceable payments. In David Chaum, Ronald L. Rivest, and Alan T. Sherman, editors, *Advances in Cryptology – CRYPTO'82*, pages 199–203, Santa Barbara, CA, USA, 1982. Plenum Press, New York, USA.
- Alessandro Chiesa, Dev Ojha, and Nicholas Spooner. Fractal: Post-quantum and transparent recursive proofs from holography. In Anne Canteaut and Yuval Ishai, editors, Advances in Cryptology – EUROCRYPT 2020, Part I, volume 12105 of Lecture Notes in Computer Science, pages 769–793, Zagreb, Croatia, May 10–14, 2020. Springer, Heidelberg, Germany.
- Chrome for Developers. Private State Tokens, May 2021. https://developer. chrome.com/docs/privacy-sandbox/private-state-tokens/.
- Simone Colombo, Kirill Nikitin, Henry Corrigan-Gibbs, David J. Wu, and Bryan Ford. Authenticated private information retrieval. In 32nd USENIX Security Symposium (USENIX Security 23), pages 3835–3851, Anaheim, CA, August 2023. USENIX Association.
- 18. Scott E. Coull, Matthew Green, and Susan Hohenberger. Controlling access to an oblivious database using stateful anonymous credentials. In Stanislaw Jarecki and Gene Tsudik, editors, *PKC 2009: 12th International Conference on Theory and Practice of Public Key Cryptography*, volume 5443 of *Lecture Notes in Computer Science*, pages 501–520, Irvine, CA, USA, March 18–20, 2009. Springer, Heidelberg, Germany.
- Alex Davidson, Ian Goldberg, Nick Sullivan, George Tankersley, and Filippo Valsorda. Privacy pass: Bypassing internet challenges anonymously. *Proceedings on Privacy Enhancing Technologies*, 2018(3):164–180, July 2018.
- T. Elgamal. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Transactions on Information Theory*, 31(4):469–472, 1985.
- Jens Groth. Simulation-sound NIZK proofs for a practical language and constant size group signatures. In Xuejia Lai and Kefei Chen, editors, Advances in Cryptology – ASIACRYPT 2006, volume 4284 of Lecture Notes in Computer Science, pages 444–459, Shanghai, China, December 3–7, 2006. Springer, Heidelberg, Germany.
- 22. Shai Halevi, Yuval Ishai, Abhishek Jain, Ilan Komargodski, Amit Sahai, and Eylon Yogev. Non-interactive multiparty computation without correlated randomness. In Tsuyoshi Takagi and Thomas Peyrin, editors, Advances in Cryptology ASIACRYPT 2017, Part III, volume 10626 of Lecture Notes in Computer Science,

pages 181–211, Hong Kong, China, December 3–7, 2017. Springer, Heidelberg, Germany.

- 23. Shai Halevi, Yuval Ishai, Abhishek Jain, Eyal Kushilevitz, and Tal Rabin. Secure multiparty computation with general interaction patterns. In Madhu Sudan, editor, *ITCS 2016: 7th Conference on Innovations in Theoretical Computer Science*, pages 157–168, Cambridge, MA, USA, January 14–16, 2016. Association for Computing Machinery.
- Shai Halevi, Yehuda Lindell, and Benny Pinkas. Secure computation on the web: Computing without simultaneous interaction. In Phillip Rogaway, editor, Advances in Cryptology – CRYPTO 2011, volume 6841 of Lecture Notes in Computer Science, pages 132–150, Santa Barbara, CA, USA, August 14–18, 2011. Springer, Heidelberg, Germany.
- 25. Yan Huang, Jonathan Katz, and David Evans. Efficient secure two-party computation using symmetric cut-and-choose. In Ran Canetti and Juan A. Garay, editors, Advances in Cryptology – CRYPTO 2013, Part II, volume 8043 of Lecture Notes in Computer Science, pages 18–35, Santa Barbara, CA, USA, August 18–22, 2013. Springer, Heidelberg, Germany.
- Ben Kreuter, Tancrède Lepoint, Michele Orrù, and Mariana Raykova. Anonymous tokens with private metadata bit. In Daniele Micciancio and Thomas Ristenpart, editors, Advances in Cryptology – CRYPTO 2020, Part I, volume 12170 of Lecture Notes in Computer Science, pages 308–336, Santa Barbara, CA, USA, August 17– 21, 2020. Springer, Heidelberg, Germany.
- 27. Eyal Kushilevitz and Rafail Ostrovsky. Replication is NOT needed: SINGLE database, computationally-private information retrieval. In 38th Annual Symposium on Foundations of Computer Science, pages 364–373, Miami Beach, Florida, October 19–22, 1997. IEEE Computer Society Press.
- Yehuda Lindell. Fast cut-and-choose based protocols for malicious and covert adversaries. In Ran Canetti and Juan A. Garay, editors, Advances in Cryptology – CRYPTO 2013, Part II, volume 8043 of Lecture Notes in Computer Science, pages 1–17, Santa Barbara, CA, USA, August 18–22, 2013. Springer, Heidelberg, Germany.
- Yehuda Lindell, Anna Lysyanskaya, and Tal Rabin. On the composition of authenticated byzantine agreement. In 34th Annual ACM Symposium on Theory of Computing, pages 514–523, Montréal, Québec, Canada, May 19–21, 2002. ACM Press.
- Harjasleen Malvai, Lefteris Kokoris-Kogias, Alberto Sonnino, Esha Ghosh, Ercan Oztürk, Kevin Lewi, and Sean Lawlor. Parakeet: Practical key transparency for end-to-end encrypted messaging. Cryptology ePrint Archive, Report 2023/081, 2023. https://eprint.iacr.org/2023/081.
- Pascal Paillier. Public-key cryptosystems based on composite degree residuosity classes. In Jacques Stern, editor, Advances in Cryptology – EUROCRYPT'99, volume 1592 of Lecture Notes in Computer Science, pages 223–238, Prague, Czech Republic, May 2–6, 1999. Springer, Heidelberg, Germany.
- 32. Christian Paquin and Greg Zaveruch. U-prove cryptographic specification v1.1 (revision 3), 2013.
- 33. Torben P. Pedersen. Non-interactive and information-theoretic secure verifiable secret sharing. In Joan Feigenbaum, editor, Advances in Cryptology – CRYPTO'91, volume 576 of Lecture Notes in Computer Science, pages 129–140, Santa Barbara, CA, USA, August 11–15, 1992. Springer, Heidelberg, Germany.
- Claus-Peter Schnorr. Efficient signature generation by smart cards. Journal of Cryptology, 4(3):161–174, January 1991.

- 35. Srinath Setty. Spartan: Efficient and general-purpose zkSNARKs without trusted setup. In Daniele Micciancio and Thomas Ristenpart, editors, Advances in Cryptology – CRYPTO 2020, Part III, volume 12172 of Lecture Notes in Computer Science, pages 704–737, Santa Barbara, CA, USA, August 17–21, 2020. Springer, Heidelberg, Germany.
- 36. Elaine Shi and Ke Wu. Non-interactive anonymous router. In Anne Canteaut and François-Xavier Standaert, editors, Advances in Cryptology – EUROCRYPT 2021, Part III, volume 12698 of Lecture Notes in Computer Science, pages 489–520, Zagreb, Croatia, October 17–21, 2021. Springer, Heidelberg, Germany.
- 37. Stefano Tessaro and Chenzhi Zhu. Short pairing-free blind signatures with exponential security. In Orr Dunkelman and Stefan Dziembowski, editors, Advances in Cryptology EUROCRYPT 2022, Part II, volume 13276 of Lecture Notes in Computer Science, pages 782–811, Trondheim, Norway, May 30 June 3, 2022. Springer, Heidelberg, Germany.
- The Cloudflare Blog. Cloudflare is now powering Microsoft Edge Secure Network, September 2023. https://blog.cloudflare.com/ cloudflare-now-powering-microsoft-edge-secure-network/.
- The LetsEncrypt Blog. Nurturing Continued Growth of Our Oak CT Log Let's Encrypt. https://letsencrypt.org/2022/05/19/nurturing-ct-log-growth. html.
- 40. Xingfeng Wang and Liang Zhao. Verifiable single-server private information retrieval. In David Naccache, Shouhuai Xu, Sihan Qing, Pierangela Samarati, Gregory Blanc, Rongxing Lu, Zonghua Zhang, and Ahmed Meddahi, editors, Information and Communications Security - 20th International Conference, ICICS 2018, Lille, France, October 29-31, 2018, Proceedings, volume 11149 of Lecture Notes in Computer Science, pages 478–493. Springer, 2018.

A Definitions

A.1 Cryptographic Primitives

Definition 1 (Additive Homomorphic Encryption Scheme). An additively homomorphic encryption (AHE) scheme is a public encryption scheme (KeyGen, Enc, Dec) that has the following property:

Homomorphic addition There is a homomorphic addition operation \bigoplus , such that for any (pk,sk) \leftarrow KeyGen (1^n) , any m, m', and any ct \leftarrow Enc_{pk}(m), ct' \leftarrow Enc_{pk}(m'), we have Dec(ct \bigoplus ct') = m + m'.

An AHE scheme usually supports rerandomization of the ciphertext ct with pk to a new ciphertext ct', which we denote as, $ct' \leftarrow \text{Rerand}_{pk}(ct)$. In practise, this can be achieved by homomorphically adding a fresh ciphertext encrypting 0.

Furthermore, we consider AHE schemes that also satisfy the following property, which we term as *addition and rerandomization indistinguishable*, if for any probabilistic polynomial-time (PPT) adversary \mathcal{A} :

$$\mathsf{Adv}_{\mathcal{A}}(1^{\kappa}) = \Pr \begin{bmatrix} (\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{KeyGen}(1^{n}); \\ b \leftarrow \{0, 1\}; \\ \mathsf{ct}_{1} \leftarrow \mathsf{Enc}(\mathsf{pk}, m_{1}); \\ \mathbf{t}_{2} \leftarrow \mathsf{Enc}(\mathsf{pk}, m_{2}); \\ \mathsf{ct}_{3,0} \leftarrow \mathsf{Rerand}_{\mathsf{pk}}(\mathsf{ct}_{1} + \mathsf{ct}_{2}) \\ \mathsf{ct}_{3,1} \leftarrow \mathsf{Enc}_{\mathsf{pk}}(m_{1} + m_{2}) \\ b' \leftarrow \mathcal{A}(\mathsf{pk}, \mathsf{ct}_{1}, \mathsf{ct}_{2}, \mathsf{ct}_{3,b}) \end{bmatrix} - \frac{1}{2}$$

is negligible in κ .

It is easy to verify both El Gamal encryption [20] (based on Decisional Diffie-Hellman Assumption) and Paillier encryption [31] (based on the Decisional Composite Residuosity assumption) satisfy the above property.

Definition 2 (Message Authentication Code). A message authentication code (MAC) consists of a triplet of algorithms (Gen, Mac, Vrfy) such that:

Validity For every $k \leftarrow \text{Gen}(1^n)$, every $m \in \{0, 1\}^*$, it holds that

$$Vrfy(m, Mac_k(m)) = 1.$$

Existential Unforgeability For all PPT adversaries \mathcal{A} with access to a oracle $Mac_k(\cdot)$, there is a negligible function negl such that

 $\Pr[\mathcal{A}^{\mathsf{Mac}_k(\cdot)} = (m, t) \text{ s.t. } m \text{ wasn't queried by } \mathcal{A} \text{ and } \mathsf{Vrfy}_k(m, t) = 1] \leq \mathsf{negl}(n).$

A MAC scheme can be constructed from one-way function.

Definition 3 (Digital Signature Scheme). A digital signature scheme is a triplet of algorithms (Gen, Sign, Vrfy) such that

Validity For every pair $(sk, vk) \leftarrow Gen(1^n)$, every $m \in \{0, 1\}^n$, it holds that

$$Vrfy_{vk}(m, Sign_{sk}(m)) = 1$$

Existential Unforgeability For every PTT adversary \mathcal{A} with access to signing oracle $\mathsf{Sign}_{\mathsf{sk}}(\cdot)$, there is a negligible function negl such that:

 $\Pr[\mathcal{A}^{\mathsf{Sign}_{\mathsf{sk}}(\cdot)}(\mathsf{vk}) = (m,\sigma) \text{ s.t. } m \text{ wasn't queried by } \mathcal{A} \text{ and } \mathsf{Vrfy}_{\mathsf{vk}}(m,\sigma) = 1] \leq \mathsf{negl}(1^n)$

One simple example is Schnorr signature scheme [34], which can be built under the discrete logarithm (DL) assumption

Definition 4 (Commitment Scheme). A commitment scheme allows one party to commit to its value, i.e., given a value m, a commitment scheme output $com \leftarrow Com(m; r)$, where r is a random uniform string. Informally, a commitment scheme satisfies the following two properties:

Hiding The commitment reveals nothing about m.

Binding It is infeasible for the committer to generate a commitment that can "open" as two different messages m, m'.

One common construction is the Pederson commitment scheme [33]. It provides information-theoretically hiding property and computationally binding under the discrete logarithm (DL) assumption.

A.2 Non-interactive Zero-knowledge Proofs (NIZK)

Let R be an efficiently computable binary relation. Let L be the language consisting of statements x such that there exists witness w where $(x, w) \in R$.

A proof system for relation R includes a key generation algorithm K, a prover P, and a verifier V. K generates a common reference string σ . The prover takes as input (σ, x, w) and outputs a poof π . The verifier takes as input (σ, x, π) and outputs 1 if the proof is acceptable and 0 otherwise. (K, P, V) is a proof system for R if it satisfies the completeness and soundness properties. PERFECT COMPLETENESS. For any adversary \mathcal{A} , we have

$$\Pr[\sigma \leftarrow K(1^k); (x, w) \leftarrow A(\sigma); \pi \leftarrow P(\sigma, x, w) : V(\sigma, x, \pi) = 1 \text{ if } (x, w) \in R] = 1.$$

PERFECT SOUNDNESS. For any adversary \mathcal{A} , we have

$$\Pr[\sigma \leftarrow K(1^k); (x, \pi) \leftarrow A(\sigma) : V(\sigma, x, \pi) = 0 \text{ if } x \notin L] = 1.$$

Additionally, (K, P, V) is a proof of knowledge for R if it satisfies the following property

PERFECT KNOWLEDGE EXTRACTION There exists a knowledge extractor $E = (E_1, E_2)$ such that for any adversary \mathcal{A} , we have

$$\Pr[\sigma \leftarrow K(1^k) : A(\sigma) = 1] = \Pr[(\sigma, \xi) \leftarrow E_1(1^k) : A(\sigma) = 1],$$

and for any adversary \mathcal{A} , we have

$$\Pr[(\sigma,\xi) \leftarrow E_1(1^k); (x,\pi) \leftarrow A(\sigma); w \leftarrow E_2(\sigma,\xi,x,\pi) : V(\sigma,x,\pi) = 0 \text{ or } (x,w) \in R] = 1.$$

Additionally, (K, P, V) is a NIZK proof system if it additionally satisfies the following (computational) zero-knowledge property.

COMPUTATIONAL ZERO-KNOWLEDGE There exists a polynomial time simulator $S = (S_1, S_2)$ such that for any non-uniform polynomial time adversary \mathcal{A} , we have

$$\Pr[\sigma \leftarrow K(1^k) : \mathcal{A}^{P(\sigma, \cdot, \cdot)} = 1] \approx \Pr[(\sigma, \tau) \leftarrow S_1(1^k) : \mathcal{A}^{S(\sigma, \tau, \cdot, \cdot)} = 1],$$

where $S(\sigma, \tau, \cdot, \cdot) = S_2(\sigma, \tau, x)$ for $(x, w) \in R$ and both oracles output failure if $(x, w) \notin R$. Also, \approx means that the difference between LHS and RHS is bounded by a negligible function of k.

[21] shows that NIZK proof with the following simulation-sound extractability can securely realize the UC NIZK-functionality.

SIMULATION-SOUND EXTRACTABILITY Consider an NIZK proof of knowledge system $(K, P, V, E_1, E_2, S_1, S_2)$. Let SE_1 be an algorithm that outputs (σ, τ, ξ) where (σ, τ) are identical to those output from S_1 . We say the NIZK proof of knowledge is simulation sound if for any non-uniform polynomial time adversary \mathcal{A} , we have

$$\Pr[(\sigma, \tau, \xi) \leftarrow SE_1(1^k); (x, \pi) \leftarrow A^{S_2(\sigma, \tau, \cdot)}; w \leftarrow E_2(\sigma, \xi, x, \pi) :$$

(x, \pi) \notice Q and V(\sigma, x, \pi) = 0 and (x, w) \notice R] \approx 0.

where Q is the list of simulation queries and responses (x_i, π_i) .

B Proof of Theorem 3

Throughout the proof, we abuse the notations for NP relations, e.g., R_0 , R_1 , and treat it as a binary function.

Corrupt server, some corrupt users. We show that for any \mathcal{A} corrupting the server and any subset of corrupt users in the hybrid world, there exists a simulator $\mathcal{S}_{cat}^{ServID,CUsers}$ (Figure 11) in the ideal world such that \mathcal{E} cannot distinguish whether it is in the hybrid world or the ideal world based on the adversarial view and honest users' outputs.

Note that the direct interaction between the corrupt server and any corrupt user doesn't require simulation. However, in our specific case, they engage with each other through several ideal functionalities that are emulated by the simulator. Consequently, the simulator must simulate this aspect of the protocol. Nonetheless, the simulator faithfully emulates these functionalities between the corrupt parties. (This is no required if only the server is corrupt.) As a result, this part of the simulated adversarial view is perfectly indistinguishable from the hybrid world, and for the sake of simplicity, we leave it out in the following discussion and simply use the term "users" to specifically refer to honest users.

Initialize and Register Phases. First, notice that in the initialize phase, S_{cat}^{ServID} simply relay the server's signing key to other corrupt users therefore, we simply couple the two worlds and use notations $\mathsf{pk}_{\mathsf{ServID}}'$ directly. Also in this phase, $\mathcal{S}_{cat}^{ServID}$ follows exactly the users' protocols to generate keys. Next, in the register phase, it learns a user uid's register requests from $\mathcal{F}_{\mathsf{cat}}^t$ and follows exactly its protocols. Again, we simply use the notation $mk_{uid}, nk_{uid}, sr_{uid}, pk_{uid}$ $\mathsf{sk}_{\mathsf{uid}}, r_{\mathsf{uid}}, \tau_{\mathsf{uid},t}, \pi_{0,\mathsf{uid}}$ as used in the real world as they are perfectly indistinguishable. Furthermore, we capture the part of "coupled" view of \mathcal{A} in these two phases with the abbreviation $pp_{uid} := (pk'_{ServID}, pk_{uid}, \tau_{uid,t}, \pi_{0,uid})$. As for the remaining adversarial view, we keep the notations $\widetilde{\mathsf{com}}_{\mathsf{uid}}$ and $\widehat{R}_0(\widetilde{x}_{0,\mathsf{uid}}, w_{0,\mathsf{uid}})$ to distinguish them from the hybrid world. (We use notation $w_{0,uid}$ here as its contents (r_{uid}, ρ_{uid}) are coupled in the two worlds.) Finally, if \mathcal{A} replies abort or provides an incorrect signature, the simulator triggers uid's abort through \mathcal{F}_{cat}^{t} , which matches what uid will do in such a scenario in the hybrid world. Therefore, the simulated outputs of the users follow exactly their outputs in the hybrid world, if \mathcal{A} acts the same in these two worlds. We argue that this is indeed true due to the computationally hiding property of the commitment scheme, and both $R_0(\tilde{x}_{0,\text{uid}}, w_{0,\text{uid}}), R_0(x_{0,\text{uid}}, w_{0,\text{uid}})$ equal 1. Finally, let U denote the set of honest users that successfully register in both worlds.

Redeem Phases. Now, we consider the adversarial view in the redeem phase. (For clearer presentation, we omit the honest outputs. This is fine as the simulator simply relays them from the adversary to the honest parties.) Note that (sid, (Request, ep') is perfectly indistinguishable from the hybrid world, as \mathcal{F}_{cat}^t receives ep' from the honest user and directly forwards it to the simulator. Moreover, let ℓ denote the total number of redeem requests. We denote the simulated

adversarial view as

 $\mathsf{Hyb}_0 := (\{(\mathsf{pp}_{\mathsf{uid}}, \widetilde{\mathsf{com}}_{\mathsf{uid}}, \widetilde{R}_0(\widetilde{x}_{0,\mathsf{uid}}, w_{0,\mathsf{uid}}))\}_{\mathsf{uid}\in U}, \{\widetilde{\mathsf{nf}}_i, \mathsf{ep}_i, \widetilde{\mathsf{ct}}_i, \pi_{1,i}, m_i, \widetilde{R}_1(\widetilde{x}_{1,i}, \widetilde{w}_{1,i})\}_{i\in[\ell]}).$

We argue that it is computationally indistinguishable from its counterpart in the hybrid world using a sequence of hybrid settings, with the first hybrid setting being

 $\mathsf{Hyb}_1 := (\{(\mathsf{pp}_{\mathsf{uid}}, \widetilde{\mathsf{com}}_{\mathsf{uid}}, \widetilde{R}_0(\widetilde{x}_{0,\mathsf{uid}}, w_{0,\mathsf{uid}}))\}_{\mathsf{uid}\in U}, \{\widehat{\mathsf{nf}}_i, \mathsf{ep}_i, \widetilde{\mathsf{ct}}_i, \pi_{1,i}, m_i, \widetilde{R}_1(\widetilde{x}_{1,i}, \widetilde{w}_{1,i})\}_{i\in[\ell]}),$

where \mathbf{nf}_i is replaced with \mathbf{nf}_i . Specifically, for each uid, sample a uniform random function f_{uid} . Then, for every i^{th} redeem in the hybrid world made by uid, replace random string \mathbf{nf}_i with $\mathbf{nf}_i := f_{\text{uid}}(\mathbf{ep}'_i)$ (Note that $\tilde{x}_{1,i}$ should also be changed as it contains the nullifier, but we abuse the notations here, as the binary function \tilde{R}_1 always outputs 1.) It is straightforward to see that this hybrid is perfectly indistinguishable from Hyb_0 . As an honest user never redeems more than once in an epoch, f_{uid} never evaluates on the same input.

 $\mathsf{Hyb}_{2,1}, \ldots, \mathsf{Hyb}_{2,|U|}$ where each subsequent $\mathsf{Hyb}_{2,j}$ modifies the nullifiers generated by a user uid: for each $\hat{\mathsf{nf}}_i$ generated by f_{uid} , replace it with $\mathsf{nf}_i := \mathsf{PRF}_{\mathsf{nk}_{\mathsf{uid}}}(\mathsf{ep}'_i)$. Finally,

$$\mathsf{Hyb}_{2,|U|} := (\{(\mathsf{pp}_{\mathsf{uid}}, \widetilde{\mathsf{com}}_{\mathsf{uid}}, R_0(\widetilde{x}_{0,\mathsf{uid}}, w_{0,\mathsf{uid}}))\}_{\mathsf{uid}\in U}, \{\mathsf{nf}_i, \mathsf{ep}_i, \widetilde{\mathsf{ct}}_i, \pi_{1,i}, m_i, R_1(\widetilde{x}'_{1,i}, \widetilde{w}_{1,i})\}_{i\in[\ell]}).$$

Each adjacent pair of hybrid settings in $\mathsf{Hyb}_1, \mathsf{Hyb}_{2,1}, \ldots, \mathsf{Hyb}_{2,|U|}$ are computationally indistinguishable. Otherwise, given any distinguishable pair of adjacent hybrid settings, we can build an adversary $\mathcal{A}_{\mathsf{prf}}$ to break the pseudorandom property of PRF. In particular, let uid be the user whose nullifiers are modified between this pair.

- 1. \mathcal{A}_{prf} interacts with an oracle that either runs an instance of PRF or a truly random function.
- 2. For every i^{th} redeem request belonging to uid, it queries ep_i on the corresponding oracle to generate a nullifer. Additionally, it follows the remaining steps of the simulator to generate the remaining entries in the view and sends the view to \mathcal{E} .
- 3. If \mathcal{E} outputs the first hybrid setting of the pair, \mathcal{A}_{prf} outputs random function, otherwise, it outputs PRF.

 $\mathsf{Hyb}_{3,1}, \ldots, \mathsf{Hyb}_{3,|U|}$ where each subsequent $\mathsf{Hyb}_{3,j}$ changes a user's simulated ciphertexts $\widetilde{\mathsf{ct}}_i$ with the hybrid world ciphertexts $\widehat{\mathsf{ct}}_i$. Finally,

$$\mathsf{Hyb}_{3,|U|} := (\{(\mathsf{pp}_{\mathsf{uid}}, \widetilde{\mathsf{com}}_{\mathsf{uid}}, R_0(\widetilde{x}_{0,\mathsf{uid}}, w_{0,\mathsf{uid}}))\}_{\mathsf{uid}\in U}, \{\mathsf{nf}_i, \mathsf{ep}_i, \widehat{\mathsf{ct}}_i, \pi_{1,i}, m_i, R_1(x_{1,i}, \widetilde{w}_{1,i})\}_{i\in[\ell]}\}.$$

Also notice that we now replace $\tilde{x}_{1,\text{uid}}$ with its hybrid world counterpart $x_{1,\text{uid}}$ as all the entries in each statement are now the same. We argue that each pair of adjacent hybrid settings in $\mathsf{Hyb}_{2,|U|},\mathsf{Hyb}_{3,1},\ldots,\mathsf{Hyb}_{3,|U|}$ are computationally indistinguishable. Otherwise, given any pair of adjacent hybrid settings that is distinguishable, we can build the following adversary $\mathcal{A}_{\mathsf{cpa}}$ that given \mathcal{E} can break the multi-challenge IND-CPA security of the PKE scheme. Again, let uid be the user whose ciphertexts we modify between this pair.
- 1. For each ct_i corresponding to uid's redeem request in the hybrid world, $\mathcal{A}_{\mathsf{cpa}}$ sends two messages $m_0 := 0$ or $m_1 := -1 \| \mathcal{\Delta} (Recall \text{ that } \mathcal{\Delta} \text{ is the difference}$ of the MAC value between the previous count and the current count.). The challenger always replies with either the encryptions of the first message or the encryptions of the second message
- 2. It generates the remaining entries in the views and sends the views to \mathcal{E} .
- 3. If \mathcal{E} outputs the first hybrid setting, \mathcal{A}_{cpa} outputs 0, otherwise, it outputs 1.

 $\mathsf{Hyb}_{4,1},\ldots,\mathsf{Hyb}_{4,|U|}$ where each subsequent $\mathsf{Hyb}_{4,j}$ changes replaces the NP relation (Treated as a binary function) \widetilde{R}_1 with \widehat{R}_1 for all redeem requests of a user. Specifically, \widehat{R}_1 is defined as follows: Let uid be the user making the i^{th} redeem request in the hybrid world and let $\mathsf{ct}_{i,\mathsf{uid}}$ be its ciphertext in $x_{1,i}$ and $n_i || \tau_i := \mathsf{Dec}_{\mathsf{skuid}}(\mathsf{ct}_{i,\mathsf{uid}})$. If $n_i \leq 0$ and $\tau_i := \mathsf{Mac}(n_i)$, \widehat{R}_1 outputs 0, else it outputs 1. Finally,

 $\mathsf{Hyb}_{4,|U|} := (\{(\mathsf{pp}_{\mathsf{uid}}, \widetilde{\mathsf{com}}_{\mathsf{uid}}, \widetilde{R}_0(\widetilde{x}_{0,\mathsf{uid}}, w_{0,\mathsf{uid}}))\}_{\mathsf{uid}\in U}, \{\mathsf{nf}_i, \mathsf{ep}_i, \widehat{\mathsf{ct}}_i, \pi_{1,i}, m_i, \hat{R}_1(x_{1,i}, \widetilde{w}_{1,i})\}_{i\in[\ell]}\}.$

Still, we argue for each pair of adjacent hybrid settings in $\mathsf{Hyb}_{3,|U|}, \mathsf{Hyb}_{4,1}, \ldots, \mathsf{Hyb}_{4,|U|}$, there is a negligible probability that $n_i \leq 0$ and $\tau_i := \mathsf{Mac}(n_i)$, thus an overwhelming probability that \hat{R}_1 outputs 1, which is what \tilde{R}_1 always outputs. (Therefore, each pair of hybrid settings is computationally indistinguishable.) Recall that an honest party will never make more than t redeem requests. Therefore the only time \hat{R}_1 can output 0 is when \mathcal{A} successfully decrements its counter and forges a MAC. In such case, we can build an adversary $\mathcal{A}_{\mathsf{mac}}$ that given \mathcal{A} can break the existential unforgeability of the MAC scheme.

- 1. \mathcal{A}_{mac} plays the simulator and interacts with \mathcal{A} with the following exception: to generate the updating ciphertext $\hat{ct}_{i,uid}$ corresponding to a uid's redeem request, \mathcal{A}_{mac} query a MAC oracle that generates the MAC of the new count.
- 2. Upon receiving ServID.T from \mathcal{A} on behalf of the corrupt server, if there exists a record that decrements uid's count, send the MAC of that count to MAC oracle.
- 3. Clearly, if \mathcal{A} can forge a MAC for a decremented count (not query before) with non-negligible probability, \mathcal{A}_{mac} can win the game with the same non-negligible probability.

The next hybrid setting we consider is

 $\mathsf{Hyb}_{5} := (\{(\mathsf{pp}_{\mathsf{uid}}, \mathsf{com}_{\mathsf{uid}}, \widetilde{R}_{0}(x_{0,\mathsf{uid}}, w_{0,\mathsf{uid}}))\}_{\mathsf{uid}\in U}, \{\mathsf{nf}_{i}, \mathsf{ep}_{i}, \hat{\mathsf{ct}}_{i}, \pi_{1,i}, m_{i}, \hat{R}_{1}(x_{1,i}, w_{1,i})\}_{i\in[\ell]]})$

where each user's simulated commitment $\widetilde{\operatorname{com}}_{uid}$ is replaced with the real commitment com_{uid} . (Hence, $\widetilde{x}_{0,uid} \to x_{0,uid}$ and $\widetilde{w}_{1,i} \to w_{1,i}$ as both include the commitment). We argue that Hyb_5 and $\mathsf{Hyb}_{4,|U|}$ are statistically indistinguishable, as the statically hiding property of the commitment schemes.

 $\mathsf{Hyb}_6 := (\{(\mathsf{pp}_{\mathsf{uid}}, \mathsf{com}_{\mathsf{uid}}, R_0(x_{0,\mathsf{uid}}, w_{0,\mathsf{uid}}))\}_{\mathsf{uid}\in U}, \{\mathsf{nf}_i, \mathsf{ep}_i, \hat{\mathsf{ct}}_i, \pi_{1,i}, m_i, R_1(x_{1,i}, w_{1,i})\}_{i\in[\ell]})$

. This hybrid replaces the NP relations (treated as binary functions) \widetilde{R}_0 , \widehat{R}_1 with R_0 , R_1 in the hybrid world. It is straightforward to see that $\widetilde{R}_0(x_{0,\text{uid}}, w_{0,\text{uid}}) = R_0(x_{0,\text{uid}}, w_{0,\text{uid}}))$ as honest users always prove correctly. Thus, it suffices to argue that $\widehat{R}_1(x_{1,i}, w_{1,i}) = R_1(x_{1,i}, w_{1,i})$ when n > 0. This follows from our definition of R_1 , intuitively, it prevents the adversary from disrupting an honest user's zero-knowledge proof.

Finally, $\mathcal{S}_{cat}^{SerVD}$ faithfully reflect \mathcal{A} 's response of either σ or \perp to ideal functionality. And the output of an honest party is exactly the same between the ideal world and the hybrid world. This concludes our proof.

Some corrupt users, honest server. We show that for any adversary \mathcal{A} corrupting some users in the hybrid world, there exists a simulator $\mathcal{S}_{cat}^{\text{CUsers}}$ (Figure 12) in the ideal world such that \mathcal{E} cannot distinguish whether it is interacting with \mathcal{A} in the hybrid world or $\mathcal{S}_{cat}^{\text{CUsers}}$ in the ideal world based on the corrupt server's view and honest users' outputs.

In the initialize phase, S_{cat}^{CUsers} samples the signing key pair for the server and sends the public key to the corrupt users. We simply use the same notation pk'_{ServID} , sk'_{ServID} as in the hybrid world as they follow the same distribution in both worlds. In the register phase, we simply use $idx_{uid} ||\sigma_{uid}$ instead of $idx_{uid} ||\tilde{\sigma}_{uid}$ as they are perfectly indistinguishable with the hybrid world. (The index is directly relayed from \mathcal{F}_{cat}^t , and the signature is signed with the coupled sk'_{ServID} .) Next, we partition ServID.T into ServID.T_H := {($pk_{uid}, \tilde{ct}_{uid}$)}_{uid∈HUsers} and ServID.T_C := {($pk_{uid}, \tilde{ct}_{uid}$)}_{uid∈CUsers}. Note that we use pk_{uid} instead of \tilde{pk}_{uid} as both are perfectly indistinguishable from their counterparts in the hybrid world. Finally, let ℓ denote the total number of redeem requests from the corrupt users.

Let $\mathsf{Hyb}_0 := (\{\pi_{0,\mathsf{uid}},\mathsf{idx}_{\mathsf{uid}} \| \sigma_{\mathsf{uid}} \}_{\mathsf{uid} \in \mathsf{CUsers}}, \{\mathsf{ServID}.T_{H,j}, \mathsf{ServID}.T_{C,j}, \tilde{\pi}_j, \tilde{\sigma}_j \}_{j \in [\ell]})$ denote the view of corrupt users. We slightly abuse the notations to allow $\pi_{0,\mathsf{uid}},\mathsf{idx}_{\mathsf{uid}} \| \sigma_{\mathsf{uid}}, \tilde{\pi}_j, \tilde{\sigma}_j$ to take values of \bot , representing the cases that some adversarial behavior is caught by the simulator and abort is sent instead.

To show indistinguishability from the hybrid world, we consider using the following hybrid settings:

We define $\mathsf{Hyb}_{1,1}, \ldots, \mathsf{Hyb}_{1,|\mathsf{HUsers}|}$ where each $\mathsf{Hyb}_{1,i}$ replaces the i^{th} user's ciphertext within each epoch with freshly generated ciphertext encrypting the same value as the ciphertext in the hybrid world. Finally, we denote:

$$\mathsf{Hyb}_{1,|\mathsf{HUsers}|} := (\{\pi_{0,\mathsf{uid}},\mathsf{idx}_{\mathsf{uid}} \| \sigma_{\mathsf{uid}} \}_{\mathsf{uid} \in \mathsf{CUsers}}, \{\overline{\mathsf{ServID}}.T_{H,j}, \mathsf{ServID}.T_{C,j}, \widetilde{\pi}_j, \widetilde{\sigma}_j \}_{j \in [\ell]})$$

We argue that each pair of adjacent settings in $Hyb_0, Hyb_{1,1}, \ldots, Hyb_{1,|HUsers|}$ are computationally indistinguishable. Otherwise, given any pair that is computationally distinguishable, we can build the following adversary \mathcal{A}_{cpa} that given \mathcal{E} can break the multi-challenge IND-CPA security of the PKE scheme. Let uid be the user whose ciphertexts we modify between this pair.

1. For each epoch ep, to generate uid's ciphertext, \mathcal{A}_{cpa} sends two messages $m_0 := 0$ and $m_1 := v_{ep}$, where v_{ep} is the value encrypted in the hybrid

Simulator $\mathcal{S}_{cat}^{ServID,CUsers}$
$ \begin{array}{llllllllllllllllllllllllllllllllllll$
2. Upon receiving \widetilde{pk}'_{ServID} from the corrupt $ServID$, relay it to corrupt users.
 3. For every honest uid, follow their initialize protocol to generate mk_{uid}, nk_{uid}, sr_{uid}, pk_{uid}, sk_{uid}, set ρ_{uid} := mk_{uid} nk_{uid} sr_{uid}. Register: Throughout this phase, the simulator truthfully emulates <i>F</i>^{R₀}_{nizk} to generate proof for a corrupt user and later verify this proof to the corrupt server. On the other hand, to simulate the view related to the honest users, the simulator instead does the following:
 Upon receiving (Register, uid) from <i>F</i>^t_{cat}, sample uniform ran- dom strings ρ[']_{uid}, <i>r</i>_{uid} and set com_{uid} := Com(ρ[']_{uid}, <i>r</i>_{uid}). Follow uid's protocol to generate <i>τ</i>_{uid,0}.
3. Emulate $\mathcal{F}_{nizk}^{\widetilde{R}_0}$ where $\widetilde{R}_0(\widetilde{x}_{0,uid}, \widetilde{w}_{0,uid}) := 1$ for any $\widetilde{x}_{0,uid}, \widetilde{w}_{0,uid}$ used by uid. Specifically, generate $\widetilde{\pi}_{0,uid}$ with $\widetilde{x}_{0,uid} :=$ $(\widetilde{\text{com}}_{uid}, c, \widetilde{\tau}_{uid,0}, \widetilde{\text{pk}}_{uid}), \widetilde{w}_{0,uid} := (\widetilde{r}_{uid}, \widetilde{\rho}_{uid}))$ for uid and send (Register, $\widetilde{\text{pk}}_{uid}, \widetilde{\text{com}}_{uid,\rho}, \widetilde{\tau}_{uid,0}, \widetilde{\pi}_{0,uid})$ to \mathcal{A} .
 Upon receiving (Verify, <i>x̃</i>_{0,uid}, <i>π̃</i>_{0,uid}) from <i>A</i>, return <i>R̃</i>₀(<i>x̃</i>_{0,uid}, <i>w̃</i>_{0,uid}). Upon receiving ⊥ from <i>A</i>, send (uid, abort) to <i>F</i>^t_{cat}. Upon receiving (idx, <i>σ̃</i>_{uid}) from <i>A</i>, if Vrfy_{p̃k'_{ServID}}(idx <i>cõ</i>_{muid}, <i>p̃</i>_{uid}) returns false, reply (uid, abort) to <i>F</i>^t_{cat}. Else, record (uid, idx, <i>σ̃</i>_{uid})
and reply (uid, ok) to \mathcal{F}_{cat}^{t} . EndRegister: Upon receiving (EndRegister) from \mathcal{E} , send (EndRegister) to \mathcal{A} and (EndRegister) to \mathcal{F}_{cat}^{t} . UpdateEpoch: Upon receiving (UpdateEpoch) from \mathcal{E} , send
(UpdateEpoch) to \mathcal{A} and (UpdateEpoch) to \mathcal{F}_{cat}^{t} . Redeem: Throughout this phase, the simulator truthfully emulates \mathcal{F}_{am} to handle communications between the server and (poten- tially corrupt) users. It also truthfully emulates $\mathcal{F}_{nizk}^{R_1}$ to generate
proof for a corrupt user and later verify this proof to the cor- rupt server. For generating the view related to honest users, the simulator does the following instead: 1. Upon receiving (Redeem, ep', m) from \mathcal{F}_{cat}^t , uniformly sample
 sid and send (sid, (Request, ep') to A through F_{am}. 2. Upon receiving (sid, ServID.T) from A through F_{am}, Pick a uniformly random string nf and computes ct_j ← Enc_{pk_j}(0) for every record in ServID.T. Let ct := (ct₁,, ct_{ServID.T}).
3. Emulate $\mathcal{F}_{nizk}^{\widetilde{R}_1}$ where $\widetilde{R}_1(\widetilde{x}_{1,\text{uid}},\widetilde{w}_{1,\text{uid}}) := 1$ for any $\widetilde{x}_{1,\text{uid}},\widetilde{w}_{1,\text{uid}}$ used by uid. Specifically, gener- ate $\widetilde{\pi}_{1,\text{uid}}$ with $\widetilde{x}_{1,\text{uid}} := (\widetilde{pk}_{ServID},ServID.T,\widetilde{nf},ep',\widetilde{ct})$ and $\widetilde{w}_{1,\text{uid}} := (idx,\widetilde{\rho}_{uid},\widetilde{com}_{\rho_{uid}},\widetilde{\sigma}_{uid})$ and send
 (Send, sid, (nf, ep', ct, π̃_{1,uid}, m), ServID) to A through <i>F</i>_{am}. Upon receiving (Verify, x̃_{1,uid}, π̃_{1,uid}), return (Verification, x̃_{1,uid}, π̃_{1,uid}, R̃₁(x̃_{1,uid}, w̃_{1,uid})) to A. 4. If receiving (sid, σ_m) from <i>t</i>_{am} and Vrfy_{pk_{ServID}}(σ_m, m) = 1, send (Signature, σ_m) to <i>F</i>^t_{cat}. Otherwise, send (abort) to <i>F</i>^t_{cat}.

Fig. 11. Simulator for adversary corrupting the server and some corrupt users.

world. The challenger always replies with either the encryptions of the first message or the encryptions of the second message for every epoch.

- 2. It generates the remaining entries in the views and sends the views to \mathcal{E} .
- 3. If \mathcal{E} outputs the first hybrid setting, \mathcal{A}_{cpa} outputs 0, otherwise, it outputs v_{ep} .

Next, let $Hyb_2 := (\{\pi_{0,uid}, idx_{uid} || \sigma_{uid} \}_{uid \in CUsers}, \{ServID.T_{H,j}, ServID.T_{C,j}, \tilde{\pi}_j, \tilde{\sigma}_j \}_{j \in [\ell]})$ be the hybrid setting where all honest ciphertexts are replaced with the ciphertexts in the hybrid world. In particular, the corresponding ciphertexts in $Hyb_{1,|HUsers|}$ and Hyb_2 encrypted the same values, while the former are freshly generated and the latter are homomorphically computed and re-randomized. Therefore, due to the perfectly/statistically homomorphic re-randomization indistinguishability of the PKE scheme, these two views are also perfectly/statistically indistinguishable.

Then, let $\mathsf{Hyb}_3 := (\{\pi_{0,\mathsf{uid}}, \mathsf{idx}_{\mathsf{uid}} || \sigma_{\mathsf{uid}} \}_{\mathsf{uid} \in \mathsf{CUsers}}, \{\mathsf{ServID}.T_{H,j}, \mathsf{ServID}.T_{C,j}, \hat{\pi}_j, \hat{\sigma}_j \}_{j \in [\ell]}),$ where we run $\mathcal{F}_{\mathsf{nizk}}^{\mathcal{R}_1}$ instead of $\mathcal{F}_{\mathsf{nizk}}^{\mathcal{R}'_1}$ to generate $\hat{\pi}_j$ and acquires $\hat{\sigma}_j$ afterward. It suffices to argue that $\tilde{\pi}_j$ and $\hat{\pi}_j$ are computationally indistinguishable.

Per our definition of $\mathcal{F}_{nizk}^{\mathcal{R}'_1}$, it only deviates from $\mathcal{F}_{nizk}^{\mathcal{R}_1}$ when the witness $w := (idx, \rho, com, \sigma)$ does not match a record in the register phase. More specifically, $\mathcal{F}_{nizk}^{\mathcal{R}'_1}$ sends an abort in this scenario. Therefore, we argue that $\mathcal{F}_{nizk}^{\mathcal{R}_1}$ also sends an abort with overwhelming probability. To see this, we further divide into the following two cases:

- 1. There does not exist a record in the register phase containing both idx, com, and therefore, a signature of idx||com is not returned earlier. In this case, $\mathcal{F}_{nizk}^{\mathcal{R}_1}$ output abort unless the signature σ provided in the witness is a valid signature of idx||com. If there is a non-negligible probability that this is a valid signature, then, we can build an adversary using \mathcal{A} to break the existential unforgeability of the signature scheme.
- 2. Else, there exists a record in the register phase containing both idx, com, but not the same ρ . In this case, $\mathcal{F}_{nizk}^{\mathcal{R}_1}$ output abort unless com is also a commitment of this different ρ . If there is a non-negligible probability that this is a valid commitment, then, we can build an adversary using \mathcal{A} to break the computational binding property of the commitment scheme.

 $Hyb_4 := (\{\pi_{0,uid}, idx_{uid} || \sigma_{uid}\}_{uid \in CUsers}, \{ServID.T_{H,j}, ServID.T_{C,j}, \pi_j, \sigma_j\}_{j \in [\ell]}),$ this hybrid setting simply replaces the ciphertexts, proofs, and signatures from the corrupt users with those in the hybrid world. Notice that the indistinguishability in previous hybrid settings suggests that the \mathcal{A} 's actions in the setting and the hybrid world are also indistinguishable, allowing us to make the switch.

It is trivial to add the honest users' outputs to this joint view, as corrupt users do not directly impact honest users, and the honest server rejects any attempts that modify the underlying plaintext of the honest ciphertexts.

Complexity. Throughout the protocol, the server maintains a table consisting of N ciphertexts. Also, during any epoch, it keeps a list of N ciphertexts,

Simulator S_{cat}^{CUsers}

Initialize: Upon receiving (Initialize, pk_{ServID}) from \mathcal{E} :

- 1. Run the $\mathcal A$ as a black box and send (Initialize, $\mathsf{pk}_{\mathsf{ServID}})$ to it. Also, send (Initialize, pk_{ServID}) on behalf of corrupt users to \mathcal{F}_{cat}^t .

2. Sample $(\widetilde{\mathsf{pk}}'_{\mathsf{ServID}}, \widetilde{\mathsf{sk}}'_{\mathsf{ServID}})$, send $\widetilde{\mathsf{pk}}'_{\mathsf{ServID}}$ to corrupt users. **Register:** Throughout this phase, emulate $\mathcal{F}^{R_0}_{\mathsf{nizk}}$ faithfully to interact with corrupt users.

- 1. Upon receiving (Register, $\mathsf{pk}_{\mathsf{uid}}, \widetilde{\mathsf{com}}_{\mathsf{uid},\rho}, \widetilde{\tau}_{\mathsf{uid},0}, \widetilde{\pi}_{0,\mathsf{uid}})$ from some uid \in CUsers, if the content is verified in an earlier interaction through $\mathcal{F}_{\mathsf{nizk}}^{R_0}$ and there does not already exist a record in ServID.U for uid, send (Register, ServID) on behalf of uid to \mathcal{F}_{cat}^t . Else, send \perp to \mathcal{A} and skip the remaining steps.
- 2. Upon receiving idx_{uid} from \mathcal{F}_{cat}^t , compute $\tilde{\sigma}_{uid} \leftarrow$ $\operatorname{Sign}_{\operatorname{sk}_{\operatorname{ServID}}}(\operatorname{idx}_{\operatorname{uid}} \| \widetilde{\operatorname{com}}_{\operatorname{uid}}) \text{ and send } \operatorname{idx}_{\operatorname{uid}} \| \widetilde{\sigma}_{\operatorname{uid}} \text{ to } \mathcal{A}.$
- 3. Finally, if $|ServID| < idx_{uid} 1$, pack the table with random public keys pk and ciphertexts $\widetilde{\mathsf{ct}} \leftarrow \mathsf{Enc}_{\widetilde{\mathsf{pk}}}(0)$ to simulate the part of the table corresponding to the honest users. Next, add $\mathsf{pk}_{\mathsf{uid}}, \widetilde{\mathsf{ct}}_{\mathsf{uid}} \leftarrow \mathsf{Enc}_{\widetilde{\mathsf{pk}}_{\mathsf{uid}}}(t \| \widetilde{\tau}_{\mathsf{uid},0})$ to $\mathsf{ServID}.T$. Also, add uid to ServID.U.

EndRegister: Follow ServID's protocol.

- UpdateEpoch: Upon receiving the new epoch ep from the ideal functionality, for all records corresponding to a corrupt user, follow ServID's protocol to update and re-randomize the ciphertexts. For all remaining records corresponding to honest users, replace their records' ciphertexts with fresh ciphertexts encrypting 0 under the same keys.
- **Redeem:** During this phase, emulate \mathcal{F}_{am} truthfully to handle requests from corrupt users.
 - 1. Upon receiving (sid, (Request, ep') from uid' through \mathcal{F}_{am} , follow ServID's protocol to send back (sid, ServID.T).
 - 2. Upon receiving a corrupt user's call on $\mathcal{F}_{nizk}^{R_1}$ with message (Prove, $\widetilde{x}, \widetilde{w}$) where $\widetilde{w} := (\widetilde{\mathsf{idx}}, \widetilde{\rho}, \widetilde{\mathsf{com}}, \widetilde{\sigma})$, emulates $\mathcal{F}_{\mathsf{nizk}}^{R'_1}$ instead with the following twist: if during register phase, there exists uid that successfully registers using $\tilde{\rho}, \widetilde{com}$ and receives $\widetilde{\mathsf{idx}}$ as its index, then simply emulating the $\mathcal{F}_{\mathsf{nizk}}^{R_1}$ functionality. Else, send \perp to the corrupt user and skip the remaining steps.
 - 3. Upon receiving $(sid, (nf, ep', \hat{ct}, \pi, m))$ from uid' through \mathcal{F}_{am} , check $|\hat{ct}| := |ServID.T|$, and ep' = ep. Then, verify that π with a consistent statement is previously recorded in $\mathcal{F}_{nizk}^{R_1}$. If either check fails, send $(\mathsf{Reply},\mathsf{sid},\bot)$ to uid' though $\mathcal{F}_{\mathsf{am}}$ and skips the remaining steps.
 - 4. Use the extracted uid from $\mathcal{F}_{nizk}^{R_1}$ in Step 2, and make a request (Redeem, ep', m, uid) to \mathcal{F}_{cat}^t . If \mathcal{F}_{cat}^t replies with σ_m , compute $L := L \bigoplus \hat{\mathsf{ct}}$ and send (uid, σ_m) to \mathcal{A} . Otherwise, it sends \perp to \mathcal{A} .

Fig. 12. Simulator for adversary corrupting users.

aggregated from the updating ciphertexts sent from the redeeming users. Furthermore, the server maintains a set of nullifiers within the current epoch, which could contain up to N pseudorandom strings. Clearly, the server storage is dominated by O(N) ciphertexts.

Moving on to the communication cost, during the register phase, each register request incurs a constant communication cost. In particular, a user transmits a commitment com_{uid} , a public key pk_{uid} , a MAC $\tau_{uid,t}$, and a zero-knowledge proof $\pi_{0,uid}$, and the server responds with an index idx and a signature σ_{uid} .

The communication cost per redeem is dominated by the server sending ServID.T and the user replying with O(N) updating ciphertexts. Consequently, the communication cost is also dominated by O(N) ciphertexts. This concludes our proof.

C Additional Figures in Section 4

We present our ideal functionality (Fig. 13) for arbitrary state transition functions. For clarity, we highlight the change with regard to our anonymous ticket ideal functionality in blue. Specifically, we use the notation $f^{n'}(\mathsf{st}, y_1, y_2)$ to denote the n' sequential applications of f starting from state st , using the last n'public/private inputs from y_1, y_2 .

Also, we give the ZK relation for registration (Fig. 14) here. The only change is generalizing a counter to state, and it is highlighted in blue.

Functionality $\mathcal{F}_{cas}^{f,st_0}$
Initialize: On message (Initialize, pk_{ServID} , sk_{ServID}) from ServID and messages (Initialize, pk_{ServID}) from users, set HUsers, CUsers := {}, ep := 0, and $T := $ {}.
Register: On message (Register, ServID) from uid:
 If ServID is corrupt, send (Register, uid) to S, if S replies with (uid, abort), send ⊥ to uid. Otherwise, if S replies with (uid, ok), continue.
2. If uid \in HUsers \cup CUsers or ep $\neq 0$, send \perp to uid and skip the remaining steps.
 3. Else if uid is corrupt, add uid to CUsers and send idx_{uid} := HUsers∪CUsers to S. Else, add uid to HUsers if uid is honest. 4. Initialize st_{uid} to st₀.
EndRegister: On message (EndRegister) from ServID, set the epoch
$e_p := 1$ and initialize $U_{e_p} = \{\}$ to keep track of redeeming users
in the new epoch. Also, send ep to \mathcal{S} .
Redeem: On message (Redeem, ep', y_1, y_2, m , uid) from uid':
1. If uid' \in HUsers, which implies uid := uid',
(a) If uid $\notin U_{ep}$, and $f(st_{uid}, y_1, y_2) \neq \bot$, add uid to U_{ep} , set $st_{uid} := f(st_{uid}, y_1, y_2)$. Else, skip the remaining steps.
(b) If ServID is honest and $ep' = ep$, set $\sigma_{y_1,m} \leftarrow \operatorname{Sign}_{sk_{ServID}}(y_1 m)$ and $R := 1$. Else, set $R := 0$.
(c) If ServID is corrupt, send (Redeem, ep', y_1, m) to the adversary S . Upon receiving (Signature, σ') from S , set
$\sigma_{y_1,m} := \sigma'$ and $R := 1$. Upon receiving (abort) from S , set $R := 0$. Otherwise, do nothing.
(d) If $R = 1$, send $\sigma_{y_1,m}$ to uid. Otherwise, send \perp to uid.
2. If uid' \in CUsers,
(a) If uid \neq uid' and uid \in HUsers, or ep' \neq ep, send \perp to uid'
and skip the remaining steps.
(b) If uid $\notin U_{ep}$ and $f(st_{uid}, y_1, y_2) \neq \perp$, update $st_{uid} :=$
$f(st_{uid}, y_1, y_2)$, and send $\sigma_{y_1, m} \leftarrow Sign_{sk_{ServID}}(y_1, m)$ to uid' .
(c) Else, send \perp to uid'.
FastForward: On message (FastForward, n' , y_1 , y_2 , uid) from adver-
sary S , if uid \in CUsers and $f^{n'}(st_{uid}, \mathbf{y}_1, \mathbf{y}_2) \neq \bot$, update $st_{uid} :=$
$f^{n'}(st_{uid}, \boldsymbol{y_1}, \boldsymbol{y_2}), \text{else, return} \perp \text{to} \mathcal{S}.$
UpdateEpoch: On message (UpdateEpoch) from ServID, set ep :=
$ep + 1$ and initializes a set $U_{ep} := \{\}$ to keep track of redeeming
users in the new epoch. Also, send ep to S .

Fig. 13. The ideal functionality for concurrent anonymous outsourced state-keeping functionality parameterized by a state transition function f, and a default state st_0 .

ZK Relation R_0

A valid instance of the registration relation R_0 contains a statement including:

- $\operatorname{com}_{\rho}$: a commitment of value ρ ,
- st₀: the default state of the transitional function,
- $-\tau$: a MAC of st₀,
- pk: a public key belonging to the prover,

and a witness including:

- -r: the randomness used for generating commitment com_{ρ} ,
- $-\rho := (\mathsf{mk} ||\mathsf{nk}||\mathsf{sr})$: a concatenation of a MAC key, a nullifier key, and source randomness used to generate an AHE key pair,

such that the following conditions hold:

Correct Commitment. $com_{\rho} = Com(\rho; r)$, Correct MAC. $\tau = Mac_{mk}(st_0)$, Correct Public Key. pk = pk', where (pk', sk') := KeyGen(sr),

Fig. 14. ZK relation R_0 for registration.

D Proof of Theorem 4

Since the proof is largely similar to the proof of Theorem 3. we mainly highlight the differences here.

Corrupt server, some corrupt users. We present a modified simulator $\mathcal{S}_{\mathsf{cas}}^{\mathsf{ServID},\mathsf{CUsers}}$ in Figure 15, with the difference highlighted in blue. In particular, the counter *n* is replaced with the generic state st and the simulator additionally relays to the server the public input y_1 for each redemption request, which they receive from the ideal functionality.

As for our hybrid argument, the $\mathsf{Hyb}_1, \mathsf{Hyb}_{2,1}, \ldots, \mathsf{Hyb}_{2,|U|}$ as they are merely dealing with the nullifers. $\mathsf{Hyb}_{3,1}, \ldots, \mathsf{Hyb}_{3,|U|}$ remain the same and rely on the same multi-challenge IND-CPA of PKE, except for the updating ciphertexts are encrypting changes to states than counters.

We spell out the next hybrid settings as the change is slightly more complex: Let $\mathsf{Hyb}_{4,1}, \ldots, \mathsf{Hyb}_{4,|U|}$ be the hybrid settings, where each subsequent $\mathsf{Hyb}_{4,j}$ changes replaces the NP relation (treated as a binary function) \widetilde{R}_1 with \widehat{R}_1 for all redeem requests of one user. Specifically, \widehat{R}_1 is defined as follows: Let uid be the user making the i^{th} redeem request in the hybrid world and let $\mathsf{ct}_{i,\mathsf{uid}}$ be its ciphertext in $x_{1,i}$ and $\mathsf{st}_i ||_{\tau_i} := \mathsf{Dec}_{\mathsf{sk}_{\mathsf{uid}}}(\mathsf{ct}_{i,\mathsf{uid}})$. If st_i is not a state that has been used by uid before and $\tau_i := \mathsf{Mac}(n_i)$, then \widehat{R}_1 outputs 0, else it outputs 1. Similar to our previous proof, \widetilde{R}_1 's outputs (which is always 1) and \widehat{R}_1 's output are computationally indistinguishable, cause otherwise, it will suggest that a PPT adversary can forge a MAC with a non-negligible probability.

 Hyb_5 is again the same as before, which just replaces the commitment of the user keys.

Finally, Hyb_6 replaces the NP relations (treated as binary functions) \tilde{R}_0 , \hat{R}_1 with R_0 , R_1 in the hybrid world. $\tilde{R}_0(x_{0,\mathsf{uid}}, w_{0,\mathsf{uid}}) = R_0(x_{0,\mathsf{uid}}, w_{0,\mathsf{uid}}))$ follows from the same argument: honest users always prove correctly in the register phase. Thus, it suffices to argue that $\hat{R}_1(x_{1,i}, w_{1,i}) = R_1(x_{1,i}, w_{1,i})$ under the case no states encrypted under the ciphertexts are tampered. (In the other case both will output 0.) Therefore, $R_1(x_{1,i}, w_{1,i})$ always outputs 1 as it includes a trapdoor allowing a successful proof, even in the case that the state is rollback. This concludes the proof for security when the server and some users are malicious.

Corrupt users, honest server. We present a modified simulator $S_{cas}^{ServID,CUsers}$ (Figure 16), with the difference highlighted in blue. Our hybrid argument remains largely the same as the adversarial view remains the same as before except for the adversary's output, which can potentially be a signature on a fast-forwarded state. However, as the simulator can extract such a fast-forward attempt from the witness of $\mathcal{F}_{nizk}^{R_1}$, the simulator ensures the state stored at the ideal functionality is also fast-forwarded by calling the backdoor FastForward. Therefore, the simulator can acquire a correct signature $\sigma_{y_1,m}$ on $y_1 \parallel m$ as the one in the hybrid world.

Complexity. Similar to the claim and proof of Theorem 3, both the server storage and per redeem communication are dominated by O(N) ciphertexts cost, with the difference being the ciphertext now encrypted a state of size s. Also in

Simulator $S_{cas}^{ServID, CUsers}$
$ \begin{array}{llllllllllllllllllllllllllllllllllll$
2. Upon receiving \widetilde{pk}'_{ServID} from the corrupt $ServID$, relay it to corrupt users.
 For every honest uid, follow their initialize protocol to generate mk_{uid}, nk_{uid}, sr̃_{uid}, pk_{uid}, sk_{uid}, set ρ̃_{uid} := mk_{uid} nk_{uid} sr̃_{uid}. Register: Throughout this phase, the simulator truthfully emulates <i>F</i>^{R0}_{nizk} to generate proof for a corrupt user and later verify this proof to the corrupt server. On the other hand, to simulate the view related to the honest users, the simulator instead does the following: Upon receiving (Register, uid) from <i>F</i>^{f,sto}, sample uniform
random strings ρ'_{uid} , \tilde{r}_{uid} and set $\widetilde{com}_{uid} := Com(\rho'_{uid}, \tilde{r}_{uid})$. 2. Follow uid's protocol to generate $\tilde{\tau}_{uid,c}$.
 Emulate \$\mathcal{F}_{nizk}^{R_0}\$ where \$\tilde{R}_0(\tilde{x}_{0,uid}, \tilde{w}_{0,uid}) := 1\$ for any \$\tilde{x}_{0,uid}\$, \$\tilde{w}_{0,uid}\$ used by uid. Specifically, generate \$\tilde{\pi}_{0,uid}\$ with \$\tilde{x}_{0,uid}\$:= \$
 R₀(<i>x̃</i>_{0,uid}, <i>w̃</i>_{0,uid}). Upon receiving ⊥ from A, send (uid, abort) to <i>F</i>^{f,sto}_{cas}. Upon receiving (<i>i</i>, <i>σ̃</i>_{uid}) from A, if Vrfy_{<i>ρ̃</i>k'_{servID}}(<i>i</i> <i>c̃õ</i>m_{uid}, <i>ρ</i>, <i>σ̃</i>_{uid}) returns false, reply (uid, abort) to <i>F</i>^{f,sto}_{cas}. Else, record (uid, <i>i</i>, <i>σ̃</i>_{uid}) and reply (uid, ok) to <i>F</i>^{f,sto}_{cas}.
EndRegister: Upon receiving (EndRegister) from \mathcal{E} , send (EndRegister) to \mathcal{A} and (EndRegister) to $\mathcal{F}_{cas}^{f,st_0}$. UpdateEpoch: Upon receiving (UpdateEpoch) from \mathcal{E} , send (UpdateEpoch) to \mathcal{A} and (UpdateEpoch) to $\mathcal{F}_{cas}^{f,st_0}$.
Redeem: Throughout this phase, the simulator truthfully emulates \mathcal{F}_{am} to handle communications between the server and (potentially corrupt) users. It also truthfully emulates $\mathcal{F}_{nizk}^{R_1}$ to generate proof for a corrupt user and later verify this proof to the corrupt server. For generating the view related to honest users, the
 simulator does the following instead: 1. Upon receiving (Redeem, ep', y₁, m) from <i>F</i>^{f,sto}_{cas}, uniformly sample sid and send (sid, (Request, ep') to <i>A</i> through <i>F</i>_{am}. 2. Upon receiving (sid, ServID.<i>T</i>) from <i>A</i> through <i>F</i>_{am}, Pick a uniformly random string nf and computes ct_j ← Enc_{pk_j}(0) for every record in ServID.<i>T</i>. Let ct̃ := (ct₁,, ct_{ServID.T}).
 Emulate \$\mathcal{F}_{nizk}^{\tilde{R}_1}\$ where \$\tilde{R}_1(\tilde{x}_{1,uid}, \tilde{w}_{1,uid}) := 1\$ for any \$\tilde{x}_{1,uid}, \tilde{w}_{1,uid}\$ used by uid. Specifically, generate \$\tilde{\pi}_{1,uid}\$ with \$\tilde{x}_{1,uid}\$:= \$(\tilde{pk}'_{ServID}, ServID.T, \tilde{n}, ep', \tilde{ct}, y_1\$) and \$\tilde{w}_{1,uid}\$:= (idx, \$\tilde{\pi}_{uid}, \tilde{com}_{\rho_{uid}}, \tilde{\pi}_{uid}, n' := 0, y_2 := \pi, y_1 := {}, y_2 := {}) and send (Send, sid, \$(\tilde{nf}, ep', \tilde{ct}, \tilde{\pi}_{1,uid}, y_1, m\$), ServID) to \$\mathcal{A}\$ through \$\mathcal{F}_{am}\$. Upon receiving (Verify, \$\tilde{x}_{1,uid}, \$\tilde{\pi}_{1,uid}, \$\tilde
4. If receiving $(\operatorname{sid}, \sigma_{y_1,m})$ from $\mathcal{F}_{\operatorname{am}}$ and $\operatorname{Vrfy}_{pk_{ServID}}(\sigma_{y_1,m}, y_1 m) = 1$, send $(\operatorname{Signature}, \sigma_{y_1,m})$ to $\mathcal{F}_{cas}^{f, \operatorname{sto}}$. Otherwise, send (abort) to $\mathcal{F}_{cas}^{f, \operatorname{sto}}$.

Fig. 15. Simulator for adversary corrupting the server and some users.

Simulator S_{cas}^{CUsers}

 $\textbf{Initialize:} \ \text{Upon receiving} \ (\textbf{Initialize}, pk_{\mathsf{ServID}}) \ \mathrm{from} \ \mathcal{E} \text{:}$

- 1. Run the $\mathcal A$ as a black box and send (Initialize, $\mathsf{pk}_{\mathsf{ServID}})$ to it. Also, send (Initialize, pk_{ServID}) on behalf of corrupt users to $\mathcal{F}_{\mathsf{cas}}^{f,\mathsf{st}_0}.$

2. Sample $(\widetilde{\mathsf{pk}}'_{\mathsf{ServID}}, \widetilde{\mathsf{sk}}'_{\mathsf{ServID}})$, send $\widetilde{\mathsf{pk}}'_{\mathsf{ServID}}$ to corrupt users. **Register:** Throughout this phase, emulate $\mathcal{F}_{\mathsf{nizk}}^{R_0}$ faithfully to interact with corrupt users.

- 1. Upon receiving (Register, $\mathsf{pk}_{\mathsf{uid}}, \widetilde{\mathsf{com}}_{\mathsf{uid},\rho}, \widetilde{\tau}_{\mathsf{uid},0}, \widetilde{\pi}_{0,\mathsf{uid}})$ from some uid \in CUsers, if the content is verified in an earlier interaction through $\mathcal{F}_{\mathsf{nizk}}^{R_0}$ and there does not already exist a record in ServID.U for uid, send (Register, ServID) on behalf of uid to $\mathcal{F}_{cas}^{f,st_0}$. Else, send \perp to \mathcal{A} and skip the remaining steps.
- 2. Upon receiving idx_{uid} from $\mathcal{F}_{cas}^{f,st_0}$, compute $\tilde{\sigma}_{uid}$ \leftarrow $\mathsf{Sign}_{\mathsf{sk}_{\mathsf{ServID}}}(\mathsf{idx}_{\mathsf{uid}}\|\widetilde{\mathsf{com}_{\mathsf{uid}}}) \text{ and send } \widetilde{\mathsf{idx}_{\mathsf{uid}}}\|\widetilde{\sigma}_{\mathsf{uid}} \text{ to } \mathcal{A}.$
- 3. Finally, if $|ServID| < idx_{uid} 1$, pack the table with random public keys pk and ciphertexts $\widetilde{\mathsf{ct}} \leftarrow \mathsf{Enc}_{\widetilde{\mathsf{pk}}}(0)$ to simulate the part of the table corresponding to the honest users. Next, add $\mathsf{pk}_{\mathsf{uid}}, \widetilde{\mathsf{ct}}_{\mathsf{uid}} \leftarrow \mathsf{Enc}_{\widetilde{\mathsf{pk}}_{\mathsf{uid}}}(c \| \widetilde{\tau}_{\mathsf{uid},c})$ to $\mathsf{ServID}.T$. Also, add uid to ServID.U.

EndRegister: Follow ServID's protocol.

- UpdateEpoch: Upon receiving the new epoch ep from the ideal functionality, for all records corresponding to a corrupt user, follow ServID's protocol to update and re-randomize the ciphertexts. For all remaining records corresponding to honest users, replace their records' ciphertexts with fresh ciphertexts encrypting 0 under the same keys.
- Redeem: During this phase, emulate \mathcal{F}_{am} truthfully to handle requests from corrupt users.
 - 1. Upon receiving (sid, (Request, ep') from uid' through \mathcal{F}_{am} , follow ServID's protocol to send back (sid, ServID.T).
 - 2. Upon receiving a corrupt user's call on $\mathcal{F}_{nizk}^{R_1}$ with message (Prove, $\widetilde{x}, \widetilde{w}$), where $\widetilde{w} := (idx, \widetilde{\rho}, \widetilde{com}, \widetilde{\sigma}, \sigma_{uid}, n', y_2, y_1, y_2)$, emulates $\mathcal{F}_{nizk}^{R'_1}$ instead with the following twist: if during register phase, there exists uid that successfully registers using $\tilde{\rho}, \widetilde{\text{com}}$ and receives idx as its index, then simply emulating the $\mathcal{F}_{nizk}^{R_1}$ functionality. Else, send \perp to the corrupt user and skip the remaining steps.
 - 3. Upon receiving (sid, (nf, ep', \hat{ct}, π, y_1, m)) from uid' through \mathcal{F}_{am} , check $|\hat{ct}| := |ServID.T|$, and ep' = ep. Then, verify that π with a consistent statement is previously recorded in $\mathcal{F}_{nizk}^{R_1}$ If either check fails, send (Reply, sid, \perp) to uid' though \mathcal{F}_{am} and skips the remaining steps.
 - 4. Use the extracted uid from $\mathcal{F}_{nizk}^{R_1}$ in Redeem step 2. If n' > 0, send (FastForward, n', y_1, y_2, uid) to $\mathcal{F}_{cas}^{f, st_0}$, skip the remaining steps if receiving \perp . Finally, send (Redeem, ep', y_1, y_2, m, uid) to $\mathcal{F}_{cas}^{f,st_0}$. If $\mathcal{F}_{cas}^{f,st_0}$ replies with $\sigma_{y_1,m}$, compute $L := L \bigoplus \hat{ct}$ and send $(\mathsf{uid}, \sigma_{y_1,m})$ to \mathcal{A} . Otherwise, it sends \perp to \mathcal{A} .

Fig. 16. Simulator for adversary corrupting users.

practice, NIZK can be implemented with recursive SNARKs with constant proof size[4], so it does not affect our asymptotic cost. This concludes our proof. \Box