ALLOSAUR: Accumulator with Low-Latency Oblivious Sublinear Anonymous credential Updates with Revocations

Samuel Jaques* Michael Lodder† Hart Montgomery‡

Abstract

A *cryptographic accumulator* is a space- and time-efficient data structure with associated algorithms used for secure membership testing. In the growing space of digital credentials, accumulators found in managing a set of valid credentials, giving efficient and anonymous methods for credential holders to prove their validity. Unlike traditional credentials like digital signatures, one can easily revoke credentials with an accumulator; however, each revocation forces existing credential holders to engage in an expensive update process. Previous works make this faster and easier by sacrificing anonymity. To improve performance without compromising privacy, we present ALLOSAUR, a multi-party accumulator based on pairings. In ALLOSAUR, we eliminate the cost of accumulating new credentials, let "credential managers" manage the accumulator values with secure multiparty computation, and allow anonymous credential updates with a square-root reduction in communication costs as compared to existing work.

A deployed digital credential system is a vast and complicated system, and existing formalisms do not fully address the scope or power of a real-world adversary. We develop a thorough UC-style formalism that allows arbitrary malicious behaviour from an adversary controlling a minority of credential managers and arbitrary numbers of users, credentials, and verifiers. In our new formalism we present a novel definition of privacy that captures as much anonymity as possible while accounting for inevitable losses from interaction with the system. The detail in our formalism reveals real-world issues in existing accumulator constructions, all of which ALLOSAUR avoids.

Our proof-of-concept implementation can update over 1000 revocations with less than half a second of total computation and 16 kB communication, at least a 5x improvement over the previous state-of-the-art in both metrics.

*Oxford University. Email: sam@samueljaques.com

†redmike7@gmail.com

‡The Linux Foundation. Email: hart.montgomery@gmail.com

1 Introduction

A major application of cryptography is for an authority to provide digital credentials to users, which users can present to others at a later point to be validated. Digital signatures are the simplest and most common form of such credentials; however there are numerous issues with using simple digital signatures: for instance, credential revocation.

An alternative approach is a *cryptographic accumulator* [Bd94]. An accumulator is a primitive that produces a short commitment to a set of elements as well as proofs of membership (or sometimes non-membership) for any element in the set. Accumulators can be extremely simple: for instance, a Merkle tree [Mer88] is an accumulator. However, many useful accumulators have additional properties.

An accumulator is called *additive* if the commitment and membership proofs can be efficiently updated as set elements are added, *subtractive* if the same holds for removing set elements, and *dynamic* if both are true [BCD$^+$17]. Further, we call accumulators *positive* if they can prove that they possess a set element in the accumulator, *negative* if they can prove that they possess a set element not in the accumulator, and *universal* if they are both positive and negative. The most interesting accumulators are of course universal and dynamic.

Since their invention by Benaloh and de Mare [Bd94], accumulators have found a wide variety of applications. Early use focused on topics such as certificate management [NN98, BLL00] and anonymous credentials [CL02, CKS09] (which we will discuss in detail later). Other accumulator applications include authenticated, outsourced computations [ABC$^+$12] and updatable signatures [CJ10,PS14]. An extensive line of work applies accumulators to blockchain-adjacent primitives, such as anonymous e-cash [ST99a, ST99b], decentralized public key infrastructure [FVY14], decentralized anonymous credentials [GGM14], stateless blockchains [BBF19], verifiable registration-based encryption [GV20], verifiable computation [OWWB20], recursive proof composition [BCMS20], and more.

1.1 Digital Credentials

One of the most popular applications for accumulators today is digital credentials [CL02]. Digital credentials not only allow users to electronically handle everyday activities that require their identity (e.g. proving ownership of a credit card used to rent a car), but also enable new functionalities like self-sovereign identity [MGGM18] and selective disclosure. In a nutshell, *self-sovereign identity* is the concept that a user should control their own credentials without ceding this power to third parties (e.g. Apple, Meta, Google, etc.). *Selective disclosure* means providing only the minimum amount of information when using a digital credential: for instance, today a UK resident might use their entire driver's license to gain access to a bar. This reveals sensitive information, such as their home address, which the bar staff should not see. With selective disclosure, the same person only needs to reveal their age and enough information to confirm

their identity (e.g. an authenticated picture associated to the credential).

The digital identity space is becoming enormous. Widely accepted market research [Res] predicts a digital identity market larger than 70 billion dollars by the year 2028. Numerous production systems focus almost exlusively on digital identity, such as the Sovrin Network [Fou], the ION Network [Din], MATTR [Hel], Polygon ID [Pol], SpruceID [Spr], zCloak [zCl], Disco.xyz [0xa], and many others. Even governments have started the transition to digital identity, including the Canadian provinces of British Columbia [oBC] and Ontario [oO] and the US state of Rhode Island [Tan].

Despite substantial business and government interest, digital identity has not recently been a popular topic in the applied cryptography community. Popular systems have not been analyzed in a rigorous cryptographic manner [Smi], and much of the formal work in the area suffers from a theoretical presentation that makes implementation difficult (more on this later). Digital identity will be a very important topic for the applied cryptography community going forward.

The Digital Identity Ecosystem. We imagine an ecosystem of digital credentials with four types of parties. **Credential holders (users)** want to obtain some digital credential from a **credential issuer**, based on some out-of-band certification (e.g., a professional license). The credentials of all users are maintained by one or more **accumulator managers (servers)**, who do not have the authority to issue certificates, but improve availability and bandwidth of the system. Users prove possession of their credentials to **verifiers**. This high-level approach matches the systems we mentioned above, as well as standards bodies such as the W3C [SLS$^+$].

For accumulators, a user who adds credentials to the accumulator is given a random ID, uncorrelated to the user's out-of-band identity. This ID remains constant among updates to the accumulator, and so the user also has a *witness* for that ID which allows the user to prove that the accumulator contains their credentials. The witness must update when the accumulator value changes. For verification, there will be another method to connect the user's accumulator ID to their other identity. For example, since the accumulator ID remains constant, the credential issuer could sign a message verifying that it belongs to the same user.

1.2 Anonymous Accumulators

Consider three motivating examples for credential anonymity:

1. A person wants to enter a bar and must prove their age is above a certain value. Which bars they enter and at which time should stay private.

2. A Canadian wants to purchase marijuana, and again must prove their age. Another country that criminalizes marijuana use may forbid them entry if that country learns about this purchase.

3. A person driving is stopped by police, who check the validity of the person's identity and driver's licence. Despite the authoritarian and/or public safety reasons to record such stops if the police make an arrest or charge, people should expect privacy if there is no charge. The racial bias of such stops (e.g. [LHMP21]) creates more need for such privacy.

Today's physical credentials (ID cards) are typically verified offline by a human and thus are difficult to track. Switching to digital credentials would make surveillance and digital tracking easy, since computers can readily log the presentation of any digital credential. There are clear financial and political incentives to do so. It is critical that credentials be difficult or impossible to track.

Existing accumulators usually provide zero-knowledge proofs of possession of the relevant credentials. This is a good first step to avoid tracking, but there is another avenue of attack. Revoking a credential must change the value of the accumulator, whichs means all users with valid credentials *must* update their witness to match the new accumulator value. When a user requests an updated witness from the accumulator manager(s), if the accumulator manager colludes with a verifier, simple timing analysis will deanonymize the verification, unless the update request is *also* anonymous.

Anonymous updates avoid the need for a "blind" accumulator, wherein the set of accumulated credentials is kept secret. Since we assume user IDs are pseudonymous, there is little need for such blindness as long as user IDs cannot be correlated with other user behaviour, which should be possible with anonymous updates and verifications.

To provide anonymous updates, previous constant-size accumulators include methods to update credentials locally using the list of changes to the accumulator since their last update. All users thus use the same data, so this is anonymous, but it is computationally intensive. In fact, [CH10] show that with m updates to the accumulator value, there must be $\Omega(m)$ communication. Current local update methods such as in [VB20,CKS09] require $\Omega(m)$ local computation as well.

Our goal is to preserve the anonymity guarantees of a local update while reducing computational load to the user. Unfortunately, previous work on accumulators has not included a proper definition of user anonymity, so we will formalize this goal later.

1.3 Our Contributions

We advance anonymous accumulators in a number of different ways, culminating in a new construction of an anonymous accumulator called ALLOSAUR that we prove secure and implement.[1] We explain our core contributions in detail below.

[1] We have all of the core functionalities of ALLOSAUR implemented but do not have a production-grade system; code available at `https://github.com/sam-jaques/allosaurust`

Formal Definitions for an Anonymous Accumulator. We provide (to our knowledge) the first formal, rigorous definition of an anonymous accumulator. Our formalism allows a powerful adversary with a global view of all public messages and metadata on all private messages, allowed to corrupt a minority of central servers and an arbitrary number of users, and to deviate in any way from the stated protocol. Despite this power, we show that our construction not only provides the basic accumulator functionality, it guarantees progress and liveness, has identifiable aborts, and also gives user indistinguishability. We use some basic building blocks (which we also define) such as a public message board (implying some sort of consensus protocol or blockchain).

To motivate the painstaking detail in our formalism, we stress that applications of accumulators are highly asynchronous multi-user protocols. Simple definitions do not capture the full scope of attack avenues. To demonstrate this, we show a practical attack on the accumulator of [VB20]. Their construction is completely secure under the definitions they use; however, their definition relies on an adversary producing a specific witness instead of just passing the anonymous verification protocol. We demonstrate how the public data will always provide the adversary enough data to produce a valid zero-knowledge proof. Other existing formalisms also have ambigious security properties: for instance, we are not certain if Nguyen's accumulator [Ngu05] resists replay attacks on verifications.[2] In our formalism, the adversary's power and our careful accounting of message passing provides much more assurance against such attacks.

By definition, using an accumulator must erode some privacy: for example, if a user proves that they have credentials, then they must belong to the set of users with valid credentials. A user that updates their credentials also reveals the time of their last update. The expressive power of our formalism allows us to precisely track such anonymity losses. Within the anonymity provided, we allow for a strong sense of indistinguishability to be proven: that any two messages cannot be correlated more than this metadata.

Single-Server Accumulators and a PIR Lower Bound. We start by considering single-server accumulators. Our overall goal is to improve the trade-off between privacy and efficiency for credential holders. The communication lower bound from [CH10]–namely, that with m updates to the accumulator value, there must be $\Omega(m)$ communication–no longer holds when credential holders can *interact* with the authority. An interactive protocol that blindly retrieves one value from a set strongly resembles private information retrieval (PIR) [CGKS95]. In our appendix we formalize that similarity with a reduction from PIR to blind witness updates.

Despite substantial recent improvements to single-server PIR [CHK22], we want to avoid the inherent drawbacks of single-server PIR [BIM00], so we focus on a multi-server accumulator protocol. However, we still provide a secure construction of a dynamic, positive, single-server, anonymous accumulator, which

[2]This may depend on implementation details of the protocol which are omitted from the paper.

is loosely based on Nguyen's pairing-based accumulator [Ngu05]. We use the batch update protocols of Vitto and Biryukov [VB20] and the static update of [KB21] in our construction. Our contribution here is a faster implementation of batch update protocol. Since recent work breaks the non-membership witness proofs of this scheme [BUV21], this is not a universal accumulator. This is not a huge practical limitation [BCD+17]: we can replace a universal accumulator with two dynamic positive accumulators, one which stores member elements of a set and one which stores non-member elements who need non-membership proofs.

ALLOSAUR. Our main contribution is the construction of a multi-server accumulator which we call ALLOSAUR. This construction is closely based on our single-server construction and assumes that more than a $\frac{2}{3}$-fraction of the servers are honest. It works in the "identifiable abort" model where if a malicious server misbehaves, it is provably identified. Our protocol is extremely long and complicated–we formalize it in excruciating detail to obtain strong and formal security guarantees–but we outline the core ideas here.

Once again, our starting point is Nguyen's pairing-based accumulator [Ngu05]. [VB20] provide a batch update protocol by evaluating polynomials and note that if servers evaluate these, then upates could be made anonymous with oblivious polynomial evaluation (OPE). OPE techniques are far more complex than we require, since the update polynomial itself is public. Instead we propose a simple one-round interactive update with secure multi-party computation (MPC). While [HKRW21] proposed MPC for Nguyen's accumulator, their technique is not anonymous and they leave out most of the formalism, relying on the UC-security of previous MPC works for the foundations.

Since the accumulator functions themselves can be verified publicly, we can strip away most of the MPC overhead. Many MPC protocols like SPDZ [DPSZ12] and its derivatives [DKL+13] spend a large percentage of their overall computation on verification, which we can mostly eliminate. The only MPC building block we need is a protocol to produce Beaver triples [Bea92b] with a public commitment to the shares created. On top of that we need a message board (which could in practice be a blockchain), private authenticated channels, and an extractable commitment scheme.

We formalize our protocol in a UC-style model. While a full UC-secure [Can01] protocol would ideal, this would add substantial complexity to an already long, complicated, and formal paper, so we defer this to future work. In particular, our need for extractability is a formidable obstacle a UC-secure formalization.

We divide the functionality of ALLOSAUR into two aspects: the core accumulator functions which maintain the accumulator value and any metadata (e.g., a list of valid credentials) and user functions which maintain and use credentials. We push as much computation as possible to local server computation, so that adding a credential needs one MPC inversion and deletion needs none. This requires a different security assumption over bilinear groups which we call the *n-Inversion Symmetric Diffie-Hellman Problem*, which we define and prove

secure in the generic group model in the appendix.

User updates require at most 2 rounds of communication (only 1 round for users whose credentials remain valid through the update), and even less MPC overhead. The core functionality provides soundness, so user privacy is our only security goal in updates; thus the user can be a trusted third party. With D deletions and A updates, the communication complexity and the user's computational complexity is $O(\sqrt{D})$. This violates the communication lower bound of [CH10], which is possible because our update is interactive. Verification in Nguyen's original protocol provides sound, anonymous, non-interactive proofs. We add an interactive challenge to prevent replay attacks.

We emphasize that ALLOSAUR is, to our knowledge, the first anonymous cryptographic accumulator with a fully rigorous proof of security.

ALLOSAUR Implementation. We implemented ALLOSAUR in Rust and evaluated the timings of our new update protocol. When updating over 1,000 changes to the accumulated set, our single-server implementation provides a 25x speed-up to server-side computation over previous work, and ALLOSAUR reduces communication from 80 kB to 16 kB. Comparing to our single-server implementation, with 10,000 changes ALLOSAUR reduces the user-side computation from 4 s to 172 ms and drops communication from 801 kB to 51 kB.

1.4 Related and Future Work

The lower bounds of [CH10] and our reduction to PIR show why other works do not satisfy our anonymity and efficiency requirements. An anonymous update must be at least as hard as PIR on the set of changes to the accumulator; however, applying PIR directly to this set is not enough for an update protocol. We could find no efficient method to do this for Merkle-tree accumulators nor RSA accumulators.[3] The update in ALLOSAUR only works because of the batch update polynomials of [VB20], which we compute with a distributed inner product (which is sufficient for PIR, as expected).

An alternative use of PIR for any accumulator (or even just digital signatures) is to have the accumulator manager maintain a list of *all* valid witnesses, and update them all with every change. Users could make a PIR request into this database to get their up-to-date witness. This is secure and anonymous, but the number of computations for any known PIR scheme is at least linear in the database size. If the server must be do one PIR for each user with each update, then the total computation of the accumulator manager is *quadratic* in the number of users, and must be done once per accumulator epoch. This is infeasible for systems with millions of users or more. Perhaps a multi-user batch PIR could work, though this is a challenging protocol to construct since users must protect against both adversarial servers and other adversarial users,

[3] We could certainly use general zero knowledge proof techniques in these instances but don't believe they would outperform our current constructoin.

all potentially colluding to de-anonymize them. Solving this problem would be interesting future work.

Adding expiry dates to credentials fails for the same reason, because we imagine frequent turnover (e.g., updating once per day). If all user's credentials expire after one day, each user needs to request a new credential every day. To issue credentials to only valid users, servers could compute a credential for all valid users which the users would retrieve with PIR. This case also has a quadratic computational cost each day. We do not see a more efficient method to do this anonymously.

In the use cases we imagine, users and verifiers have limited computational power compared to servers. This forbids methods like bloom filters [HRMM18] or sending a list of revoked credentials to each verifier (like certificate revocation lists). Our work concerns only *membership*, and in practice our systems would rely on other techniques such as [BCF+21] to prove other properties of the credential. Future work could combine anonymous proofs of membership with a number of different things, such as digital signatures, which, as demonstrated in [BDET19], could be combined with expiry dates for a "revoke-by-default" sort of system.

For other approaches to digital credentials, the authors of [HRMM19] build a system where witnesses are single-use and users request them in a batch, with a revocation system on top of that. This system lacks our strong anonymity guarantees, though. The Camenisch-Kohlweiss-Soriente [CKS09] accumulator update seems hard to delegate anonymously, presenting interesting future work. Finally, there have been a number of post-quantum accumulators [LLNW16, DRS18] but these are all hash-based, have non-constant sized witnesses, and update methods that do not appear nearly as fast as those of group-based accumulators.

Schul-Ganz and Segev [SGS20] formally prove generic group lower bounds on batch verification, implying that certain efficiency barriers are unlikely to be crossed. In [BBDE19], the authors build an updatable anonymous credential system, but it lacks several features of our accumulator-based system. Most recently, Agievich constructed a basic blind accumulator [Agi22], but this accumulator is linear in the number of accumulated elements, is purely additive, and its notion of blindness is simply equivalent to blind addition, which gives much less functionality than our construction.

1.5 Paper Outline

The rest of the paper proceeds as follows. We begin with basic accumulator definitions in Section 2. Then in Section 3, we define our single-server accumulator and explain some limitations of the single-server scenario and issues with some existing single-server accumulators. In Section 4, we explain the basics of our multi-party framework and our multi-party accumulator ALLOSAUR. This is quite complicated so we unfortunately have to leave most of the details to the appendix. Finally, we provide some performance metrics in Section 5.

Due to the complexity of our formalisms and proofs, most of the real work in

our paper happens in the appendix. In Section 6, we show that a single-server accumulator with blind updates implies a *private information retrieval* (PIR) protocol. We then get to the "meat" of our construction: Section 7 formally defines our MPC formalism and some primitives we use (like message boards and random oracles), and Section 8 formally defines an accumulator itself. Next, Section 9 defines the control program flow that we use in our security proofs. This lets us define the security of a multi-server accumulator, which we do in Section 10.

In Section 11, we define more formalisms that help us model server and user behavior. We then formalize necessary secret sharing protocols in Section 12. Finally, in Section 13, we formally present our multi-server accumulator ALLOSAUR. We provide an overview of our proof in Section 14. In the remaining 14 sections, we formally (and in painstaking detail) prove the security of ALLOSAUR. We have attempted to subdivide the proof so that it is easy to follow and can be read in "bite-size" segments.

2 Definitions

2.1 Definition

The basic structure of an accumulator is a large set of *user IDs* $\mathcal{Y}$. The accumulator manager creates an initial *accumulator value* A, which represents an *accumulated set* of elements from $\mathcal{Y}$. The accumulator value is often short, so it cannot truly store the accumulated set; the accumulator manager typically stores the accumulated set as local auxiliary data. A function Gen creates the first accumulator value, and subsequently, Add and Del instruct the accumulator manager to modify the accumulator value to add or delete elements from the accumulated set.

Each user has a specific ID $y \in \mathcal{Y}$. They can interact with the accumulator manager to retrieve a *witness* w for their element y. The witness, combined with the accumulator value, should allow the user to prove that they are part of the accumulated set. This is captured in a verify protocol Ver, where a verifier who knows the accumulator value (e.g., by downloading it from the accumulator manager) interacts with a user so that the user can prove they possess a valid witness.

Finally, there is an interactive update between the user and the accumulator manager, which allows the user to update their witness when the accumulator value changes.

To formalize this, we start from Helminger et al. [HKRW21], although we make some departures from their definition which we will discuss.

We will write interactive protocols as $\langle \mathcal{A}(a), \mathcal{B}(b) \rangle \to (s_a, s_b, \Lambda)$, where a (resp. b) is the private input to $\mathcal{A}$ (resp. $\mathcal{B}$) and s_a (resp. s_b) is its private output. Λ is the transcript of exchanged messages.

A dynamic accumulator consists of PPT algorithms $(\mathsf{Gen}, \mathsf{Add}, \mathsf{Del}, \mathsf{Ver})$ and PPT interactive protocols $(\langle \mathsf{Wit}_u, \mathsf{Wit}_s \rangle, \langle \mathsf{Prove}, \mathsf{Ver}_v \rangle, \langle \mathsf{Upd}_u, \mathsf{Upd}_s \rangle)$, with a key

space $K_{\lambda,N}$, input space $\mathcal{Y}_k$ for each $k \in K_{\lambda,N}$, such that:

- $\mathsf{Gen}(N) \to (\mathsf{sk}, A_0, \mathsf{aux}_0)$ creates an initial accumulator value A_0, an accumulator key sk, and auxiliary information aux_0. Other definitions assume an output pk, but we follow [BCY20] and include pk as part of the accumulator.
- $\mathsf{Add}(\mathsf{sk}, A_i, y, \mathsf{aux}_i) \to (A_{i+1}, \mathsf{aux}_{i+1}, \mathsf{aux}_{upd})$ should add y to the accumulator. It takes an accumulator value and a user ID y, updates the accumulator and its auxiliary information and produces some auxiliary information to use for updating witnesses. Add is allowed to output $\perp$ (e.g., if y is already in the set).
- $\mathsf{Del}(\mathsf{sk}, A_i, S, \mathsf{aux}) \to (A_{i+1}, \mathsf{aux}_{i+1}, \mathsf{aux}_{upd})$ has the same functionality as Add, but deletes elements from the accumulated set.
- $\mathsf{Ver}(y, w, A_i) \to \{0, 1\}$ is a deterministic algorithm to verify a witness and a signature. If w was generated as a witness for y for the accumulator A_i, this should output 1, and otherwise 0.
- $\langle \mathsf{Wit}_u(y), \mathsf{Wit}_s(\mathsf{sk}, \mathsf{aux}, A_i) \rangle \to (w, \emptyset, \Lambda)$ allows a user to obtain a witness w for an accumulator A_i, if y has been added to the accumulator. The user runs Wit_u and the accumulator manager(s) run Wit_s.
- $\langle \mathsf{Prove}(y, w, A_i), \mathsf{Ver}_v(A_i) \rangle \to (\emptyset, \{0, 1\}, \Lambda)$ allows a user running Prove to prove to a verifier running Ver_v that the user possesses a witness and an element y, such that y is in the accumulated set.
- $\langle \mathsf{Upd}_u(y, w, A_i), \mathsf{Upd}_s(\mathsf{sk}, A_{i+t}, \mathsf{aux}_{upd}) \rangle \to (w', \emptyset, \Lambda)$ updates a membership witness, using the auxiliary information from some combination of additions and deletions.

To avoid cluttering notation:

- Other accumulators have an Eval function which initializes the accumulator with an initial set. We assume the accumulator is initialized with the empty set, and elements are added one at a time.
- We assume that Upd can parse the concatenation of update information from multiple ordered additions or deletions.
- All the algorithms take 1^λ as an input, and we omit this.

We also define dynamic functions $\overline{\mathsf{Add}}$, $\overline{\mathsf{Del}}$, and $\overline{\mathsf{Upd}}$ for security games. We assume there is an environment for $\overline{\mathsf{Add}}$ that contains an accumulator A, a secret key sk, auxiliary information aux, update information aux_{upd}, and an accumulated set S. $\overline{\mathsf{Add}}$ is called with just a set S' as input, and calls Add for each $y \in S'$ and the appropriate environment variables. It then updates A and appends to aux_{upd} with the output of each Add, and updates $S \leftarrow S \cup S'$. To avoid cluttering notation, we assume $\overline{\mathsf{Add}}(\emptyset)$ does nothing and returns.

$\overline{\mathsf{Del}}(S')$ is the same except it calls Del and sets $S \leftarrow S \setminus S'$.

For $\overline{\mathsf{Upd}}$, we also assume it uses the environment variables sk, A, and aux_{upd}.

Issuing Credentials. Our definition does not include a credential issuer. In practice we assume the credential issuer will sign all requests for additions and deletions to certify to the accumulator manager that they are valid. For our security analysis, we remove this functionality, as in all cases we focus on a powerful adversary who can control additions and deletions into the accumulated set.

2.1.1 Security.

The algorithms of an accumulator must satisfy the following properties:

Correctness: Any sequence of additions and deletions implicitly defines a set of elements that are supposed to be in the accumulator; if we update a witness from the auxiliary information of these additions and deletions, it should verify properly.

Let $S_0 = \emptyset$, and let $T_1, \ldots, T_n$ and $R_1, \ldots, R_n$ be a collection of subsets of $\mathcal{Y}$ such that: (a) $S_{i+1} := (S_i \cup T_{i+1}) \setminus R_{i+1}$; (b) $T_{i+1} \cap S_i = \emptyset$ (c) $R_{i+1} \subseteq S_i \cup T_{i+1}$.

Then for any $j \leq n$ and any y such that $y \in S_i$ for all $i \geq j$,

$$\Pr\left[\mathsf{Ver}(y, w_n, A) = 1 \;\middle|\; \begin{array}{c} (A, \mathsf{aux}, \mathsf{sk}) \leftarrow \mathsf{Gen}_A(N) \\ \{\overline{\mathsf{Add}}(T_i), \overline{\mathsf{Del}}(R_i)\}_{i=1}^{j-1} \\ w_{j-1}, \bot, \Lambda \leftarrow \langle \mathsf{Wit}_u(y), \mathsf{Wit}_s((\mathsf{sk}), \mathsf{aux}, A)\rangle \\ \left.\begin{array}{c} \overline{\mathsf{Add}}(T_i) \\ w_i' \leftarrow \overline{\mathsf{Upd}}(y, w_{i-1}) \\ \overline{\mathsf{Del}}(R_i) \\ w_i \leftarrow \overline{\mathsf{Upd}}(y, w_i') \end{array}\right\}_{i=j}^{n} \end{array}\right] = 1$$

Collision-freeness: This is analogous to soundness for a digital signature. If an adversary is not offered a valid witness, or their ID y was removed from the accumulator, their witness should not verify. In the security game, an adversary requests as many additions, deletions, and witness updates as it likes. This implicitly defines some set S of elements in the accumulator, and the adversary should not be able to produce any membership witnesses for elements outside that set.

With our definitions of the dynamic add, delete, and update functions, we can define collision-freeness formally for a PPT algorithm $\mathcal{A}$ as

$$\Pr\left[\begin{array}{c} y \notin S \\ \mathsf{Ver}(A, y, w) = 1 \end{array} \;\middle|\; \begin{array}{c} (A, \mathsf{aux}, \mathsf{sk}) \leftarrow \mathsf{Gen}(N) \\ (y, w) \leftarrow \mathcal{A}^{\overline{\mathsf{Add}}, \overline{\mathsf{Del}}, \overline{\mathsf{Upd}}_s}(A) \end{array}\right] \leq \mathrm{negl}(\lambda)$$

We based our definition on the universally composable accumulator from [BCD+17], but unfortunately their definition does not give an adversary access to witness updates, only to aux_{upd}. With publicly-computable updates, this is equivalent since the adversary can compute witness updates themselves; however, we consider cases where witness updates use the secret key, so we must give the adversary oracle access to this.

2.2 Blind Verification

The first anonymity property is to verify membership or non-membership blindly. This means it should be indistinguishable among users.

For notational ease, we will define $(\mathsf{sk}, A, \mathsf{aux}, w) \leftarrow \mathsf{Acc}(N, y)$ as the serial evaluation of Gen then some sequence of Add and Del and Upd, such that the accumulator produces w as a valid witness for y. At this point we assume a server, not a verifier, performs all of the accumulator updates and witness updates, so the security properties of the verification are distinct.

Completeness: Witness proofs should still work in the interactive protocol. Formally, for any $y \in \mathcal{Y}$, and any sequence of additions and deletions to construct any accumulator representing a set containing y,

$$\Pr\left[a_v = \mathsf{Ver}(A, y, w) \middle| \begin{array}{c} (\mathsf{sk}, A, \mathsf{aux}, w) \leftarrow \mathsf{Acc}(N, y) \\ (a_p, a_v, \Lambda) \leftarrow \langle \mathsf{Prove}(A, y, w), \mathsf{Ver}(A) \rangle \end{array}\right] = 1$$

Commitment soundness: Since the verification is blind and interactive, it changes our security notion. An adversary could pass an insecure protocol without needing any values of y or w. We must ensure this does not happen.

Further, if an adversary ever obtains a value of y and w, then they *should* be able to pass the verification protocol at any time, at least until y is removed from the accumulator. We also want to allow the adversary to add and remove any elements it wants from the accumulator.

Ultimately, we must only prevent an adversary from passing the protocol without possessing a valid pair of y and w. Thus, we define soundness with an extractor: If an adversary can pass the protocol, then an extractor algorithm should be able to produce a valid y and w from the adversary's state.

We assume here that during the interactive witness issuance, one of the messages to the accumulator manager is the element y for which a witness is requested. From this, we can define the set S' as the intersection of the implicitly-defined set S of accumulated elements with the set of all y that an adversary sends during interaction with Wit_s. With this definition, for any PPT adversary $\mathcal{A} = (\mathcal{A}_0, \mathcal{A}_1)$ such that

$$\Pr\left[a_v = 1 \middle| \begin{array}{c} (A, \mathsf{aux}, \mathsf{sk}) \leftarrow \mathsf{Gen}(N) \\ \mathsf{st} \leftarrow \mathcal{A}_0^{\mathsf{Wit}_s, \overline{\mathsf{Add}}, \overline{\mathsf{Del}}, \overline{\mathsf{Upd}}}(A, \mathsf{st}) \\ (\mathsf{st}, a_V, \Lambda) \leftarrow \langle \mathcal{A}_1(A, \mathsf{aux}, \mathsf{aux}_{upd}, \mathsf{st}), \mathsf{Ver}_v(A) \rangle \end{array}\right] \geq \mathrm{negl}(\lambda)$$

then there *exists* an extractor $\mathcal{E}$ such that

$$\Pr\left[\mathsf{Ver}(A, y, w) = 1 \middle| (y, w) \leftarrow \mathcal{E}(A, \mathsf{aux}, \mathsf{aux}_{upd}, S, \mathsf{st})\right] \geq \mathrm{negl}(\lambda)$$

and for *any* extractor $\mathcal{E}$,

$$\Pr\left[y \notin S' \wedge \mathsf{Ver}(A, y, w) = 1 \middle| (y, w) \leftarrow \mathcal{E}(A, \mathsf{aux}, \mathsf{aux}_{upd}, S, \mathsf{st})\right] < \mathrm{negl}(\lambda).$$

The extractor is not explicitly given oracle access to $\mathcal{A}$, but one can assume the first extractor implements any algorithms that $\mathcal{A}$ implements based on the order of existential quantifiers: for all $\mathcal{A}$, we can choose $\mathcal{E}$ in this way.

The restriction of y to S' (and not just S) ensures that the extractor is able to extract one of the adversary's witnesses. [VB20] do not satisfy this definition; we detail an attack in Section 3.1.

Proof indistinguishability: If an adversary controls all parameters of the scheme, even generating a user's ID and witness, they should still not be able to distinguish the proofs produced by different users. In many use cases, all the users are public, and the number of users is not cryptographically large (e.g., on the order of 2^{30}), and an adversarial verifier may have extra information that can narrow down the set of possible users. Hence, we assume the worst case and require that an adversary cannot distinguish users even among a set of just 2.

Formally: For all adversaries $\mathcal{A} = (\mathcal{A}_0, \mathcal{A}_1)$,

$$\Pr\left[b' = b \,\middle|\, \begin{array}{c} (\mathsf{st}, A, y_0, w_0, y_1, w_1) \leftarrow \mathcal{A}_0(N) \\ w_0, w_1 \leftarrow \perp \text{ if } \mathsf{Ver}(A, y_0, w_0) \neq \mathsf{Ver}(A, y_1, w_1) \\ b \leftarrow \{0,1\} \\ b' \leftarrow \mathcal{A}_1^{\mathsf{Prove}(A, y_b, w_b)}(\mathsf{st}) \end{array}\right] \leq \frac{1}{2} + \mathrm{negl}(\lambda)$$

Here we emphasize that in our definitions of the algorithms of an accumulator, the only persistent state that belongs to a user is its values of y and w. This means that $\mathcal{A}_0$ can simulate as many users as it wants, and as many updates as it wants, before finally producing the outputs to the challenger.

2.3 Blind Updates

For this blindness game, we assume a single update interaction. An adversary creates an entire accumulator, and passes two elements to a challenger, ensuring that both (or neither) have valid witnesses. Then the adversary interacts with one at random. As with blind verification, because the user has no persistent state, an adversary can simulate any number of interactions with any number of users before providing the challenges.

Without anonymous connections (e.g., the user connects from the same IP address each time), then witness updates are only pseudonymous, even with a perfectly blind protocol. This definition does not capture the power of a stronger adversary that has this ability to connect different sessions. Our lengthy formalism in the appendix does capture this, however.

For blind updates, all adversaries $(\mathcal{A}_0, \mathcal{A}_1)$ must satisfy:

$$\Pr\left[b = b' \,\middle|\, \begin{array}{c} (\mathsf{sk}, \mathsf{aux}, A, y_0, w_0, y_0, w_1, \mathsf{st}) \leftarrow \mathcal{A}_0() \\ w_0, w_1 \leftarrow \perp \text{ if } \mathsf{Ver}(A, x_0, w_0) \neq \mathsf{Ver}(A, x_1, w_1) \\ b \leftarrow \{0,1\} \\ b', w_b', \Lambda \leftarrow \langle \mathcal{A}_1(\mathsf{st}_0), \mathsf{Upd}_u((\mathsf{sk}), A_0, x_b, w_b) \rangle \end{array}\right] \leq \frac{1}{2} + \mathrm{negl}(\lambda)$$

If the server colludes with a verifier, they can break anonymity by revoking a user's credentials and checking whether they are still able to verify. This is unavoidable in a collision-free accumulator.

3 Single-Server Protocol

The starting point for our single-server protocol is Nguyen's pairing-based accumulator [Ngu05]. We use the batch update protocols of Vitto and Biryukov [VB20] and and the static update of [KB21]. We construct a dynamic positive accumulator, since recent work breaks this kind of construction with non-membership witness proofs [BUV21].

The accumulator is parameterized by two elliptic curve groups G_1 and G_2 of prime order q, with a Type 2 pairing $e : G_1 \times G_2 \to G_T$. There are public generators P, $\tilde{P}$, and $e(P, \tilde{P})$ of the three groups, as well as another six points $K, K_0, X, Y, Z \in G_1$ and $\tilde{K} \in G_2$. These points should all be selected uniformly at random and no one should know the discrete logarithm of any one point with respect to any other point.

We also use a (collision-resistant) hash function $H : \{0,1\}^* \to \{0,1\}^{2\lambda}$, which we will model as a random oracle in our security proofs.

Generation. $\mathsf{Gen}()$ starts by selecting uniformly random values α, s_m, and r. It sets $\mathsf{sk} \leftarrow (\alpha, s_m)$, $\tilde{Q} \leftarrow \alpha\tilde{P}$, $V \leftarrow rP$, $\tilde{Q}_m \leftarrow s_m\tilde{K}$, and $A_0 \leftarrow (V, \tilde{Q}, \tilde{Q}_m)$.

Because we have a trapdoor and we do not use non-membership witnesses, there is no need to "pre-load" the accumulator with any points.

Additions. For $\mathsf{Add}((\mathsf{sk}), y, A, \mathsf{aux})$, assume $\mathsf{aux} = (Y, W)$. If $y \notin Y$, it sets $Y = Y \cup \{y\}$ and $W = W \cup \{(y, \frac{1}{y+\alpha}A)$. The set Y represents the accumulated set, and W is a set of witnesses for all accumulated elements.

Add does not change A. The notion of the "accumulated set" is only implicitly defined by the set of valid witnesses. Further, since we have not revoked any credentials, it is fine to have existing witnesses verify with this accumulator after this addition.

Deletions. $\mathsf{Del}((\mathsf{sk}), y, A, \mathsf{aux})$ parses aux as (Y, W). If $y \in Y$, it parses A as $(V, \tilde{Q}, \tilde{Q}_m)$ and sets $V \leftarrow (y + \mathsf{sk})^{-1}V$, concatenates (y, V) to aux_{upd}, and sets $Y \leftarrow Y \setminus \{y\}$.

For all $(y', C) \in W$, it sets $C = \frac{1}{y-y'}(C - V)$. This updates the list of valid witnesses. This isn't necessary for a single-server accumulator, as it possesses the secret key and can create each witness by computing $\frac{1}{y+\alpha}V$, or even compute witnesses on-the-fly during other operations. However, we explain it this way because our multi-server protocol maintains a database of witnesses to avoid any MPC for deletions.

Verifications. $\mathsf{Ver}(A, y, w)$ parses A as $(V, \tilde{Q}, \tilde{Q}_m)$ and w as (x, C, R_m). It outputs 1 if $e(C, y\tilde{P} + \tilde{Q}) = e(V, \tilde{P})$ and $e(R_m, y\tilde{K} + \tilde{Q}_m) = e(xK + K_0, \tilde{Q}_m)$.

Witness issuance. This is an interactive protocol. It starts with $\mathsf{Wit}_u(y)$, which selects a random $x, k \in \mathbb{F}_q$, computes $R_{ID} \leftarrow xK$, and set $h \leftarrow H(R_{ID}, kK)$. Then Wit_u sends $(y, h, k - hx \mod q, R_{ID})$. This is a basic Schnorr proof-of-knowledge for the discrete log of R_{ID}.

Upon receiving (y, h, r, R), Wit_s checks that $H(R, rK + hR) = h$; if so, it computes $R_m \leftarrow \frac{1}{y+s_m}(R + K_0)$. It looks up $(y, C) \in W$ (as part of aux) and sends (C, R_m) back.

On receiving (C, R_m), Wit_u sets $w \leftarrow (x, C, R_m)$.

The point R_m is a long-term signature which never changes, which allows the user to prove that they are the valid "owner" of the ID y, even when performing zero-knowledge proofs that do not reveal y. In contrast, the point C must change with the accumulator value.

Proofs. We use a non-interactive zero-knowledge proof from Nguyen [Ngu05] to prove knowledge of (x, y, C, R_m) such that $e(C, y\tilde{P} + \tilde{Q}) = e(V, \tilde{P})$ and $e(R_m, y\tilde{K} + \tilde{Q}_m) = e(xK + K_0, \tilde{K})$, i.e., that $\mathsf{Ver}(A, y, w) = 1$. This protocol uses a Fiat-Shamir transform of a single-round interactive protocol. The full details are in the appendix. To prevent replay attacks, the verifier sends a random challenge string to the prover and requires a fast response from the prover.

Updates. To start, Upd_u first sends a request of the number of accumulator changes since their witness was last valid. Following [VB20], we use exactly the tuples corresponding to deletions, to produce a list $(y_d, V_d), \ldots, (y_1, V_1)$ of all deletions used. These deleted elements are considered public knowledge. Since all user interactions hide their ID y, deleting the element should not be correlatable with any of the activity from the user owning y.

The deletions are numbered so that y_d was the most recently deleted element. This defines two polynomials over $\mathbb{F}_q$

$$v(x, \alpha)V_0 = \sum_{s=1}^{d} V_s \prod_{j=1}^{s-1}(y_j - x) \text{ and } d(x) = \prod_{t=1}^{d}(y_t - x),$$

where V_0 is the accumulator value before the first deletion, the last accumulator value for which the user's witness is valid.

A user *could* compute $v(x, \alpha)V_0$ without knowledge of the secret α, since it is implicit in the form of the intermediate accumulator values V_s. To save user computation, the server computes the coefficients of both polynomials, where the coefficients of $v(x, \alpha)V_0$ will be points Ω_i in G_1. Then Upd_s sends $\{\Omega_0, \ldots, \Omega_d\}$ and $(d_0, \ldots, d_d)$ to the user.

Upd_u then computes $d(y)$ and $v(y, \alpha)V_0$ using these coefficients. If $d(y) \neq 0$, the user parses their witness as (x, C, R_m) and sets $C \leftarrow d(y)^{-1}(C - v(y, \alpha)V)$; otherwise, they set $C = \mathcal{O}$.

So far this is identical to [VB20]. Notice that computing the update polynomials is quadratic in the number of roots. However, update polynomials from repeated updates can be applied sequentially. Thus, the server need not compute the full update polynomial, but can split it into repeated updates of some fixed size. The server sends each one, and the user updates sequentially. Because the update polynomials require neither the user's element y nor their existing witness, the server can compute and send all of these polynomials with-

out further user input, and the user can apply them all without further server input.

3.1 Security

We next discuss the security of an anonymous accumulator. Proving security of an anonymous accumulator–which amounts to a complicated set of interactive protocols–is extremely complicated, particularly when we want to prove privacy/anonymity properties. This takes us dozens of pages to formalize in a UC-style framework, so we cannot present this in full here. While we would be thrilled with a simpler proof of security, we could not conceive of one and leave it as interesting future work.

So, we defer the formal proofs of security to our more expressive formalism in the appendix. Here, we give some intuition for the proofs and describe the new hardness assumption we require. We also provide examples of shortcomings of security definitions in previous work, giving evidence as to why our security proofs unfortunately must be so complicated. We note that completeness follows straightforwardly from the definitions.

In other instantiations of this style of accumulator [Ngu05, VB20, HKRW21], addition of an element y requires changing the accumulator value V to $(y + \alpha)V$. Since we do not provide non-membership witnesses, we can exclude this. [KB21] make the same choice but are only able to prove security against a static adversary who must choose which elements to obtain witnesses from before starting the security game. For active security, we require the hardness of a new group problem, the "n-Inversion Symmetric Diffie-Helman problem" (n-ISDH), defined as follows:

Definition 3.1. Let G_1, G_2, and G_T be groups with a type-3 pairing from $G_1 \times G_2$ to G_T. Given G, $\lambda G, \gamma G \in G_1$, $\tilde{G}, \lambda\tilde{G} \in G_2$, and query access to a function $f : (y, Q) \mapsto \frac{1}{\lambda+y}Q$ for $Q \in G_1$, compute

$$\left(\prod_{i=1}^{k}(\lambda + y_i)\gamma G, y_1, \ldots, y_k\right)$$

such that at least one value of y appears in the list $y_1, \ldots, y_k$ at least one more time than we queried it to f.

Such "one-more" styles of problem appear in other blind signature schemes. In the appendix we prove in the generic group model that this problem has (generic) security approximately equal to the cube root of the group size. Since the n-ISDH problem almost captures witness issuance directly, the proof of soundness is straightforward, once we extract a witness from the zero knowledge proofs with the techniques of [Ngu05].

To show update blindness, the only message the user sends is the time since their last update. In the blind update game, the two "challenge users" have witnesses valid for the same accumulator, so they send them at the same time. This makes them indistinguishable.

A Practical Attack on [VB20]. We stress here the importance of the long-term signature R_m. Since verification uses a *zero-knowledge* proof of knowledge, an adversary with *any* witness can prove that it is a valid witness, and the verifier has no way of knowing which ID y the adversary used. Since the values y are public, and witnesses are not assumed to be private, an adversary has easy access to this data. Hence, we need users to generate their long-term secret x which the server associates to the element y via the blind signature R_m. As described, we rely on the server to correctly decide when it should send a witness to some party; in ALLOSAUR, the servers only make one signature R_m for any ID y.

The long-term signature started with [Ngu05] but is missing from [VB20]. Because the authors of [VB20] modify the accumulator for additions, if an element y is added to an accumulator value V, then the old accumulator value becomes a witness for y in the new accumulator. They note this problem and point out that *single* additions, as opposed to batch operations, create such impersonation attacks. However, a batch addition creates the same problem. In their specification, the accumulator manager publishes a polynomial $d_{\mathcal{A}}(x)$ consisting of all additions to the accumulator. Factoring this polynomial – which is efficient over a finite field – reveals all additions y.

Thus, an adversary with no valid ID y nor witness can request the batch update from accumulator V_0 to V_t. They receive $d_{\mathcal{A}}(x)$ and a sequence of elliptic curve points $\Omega_0, \ldots, \Omega_{t-1}$. They can factor $d_{\mathcal{A}}(x)$ to obtain the list of additions $\{y_1, \ldots, y_t\}$. They know that

$$v_{\mathcal{A}}(x)V_0 := \sum_{s=1}^{t} \left(\prod_{i=1}^{s-1} (y_i + \alpha) \prod_{j=s+1}^{t} (y_j - x) \right) V_0 = \sum_{i=0}^{t-1} x^i \Omega_i.$$

$v_{\mathcal{A}}(x)$ is homogenous[4] in α and x; thus, the coefficient of x^{t-2} is some linear polynomial $a_1\alpha + a_0$, with the coefficients of *this* polynomial being known functions of $y_1, \ldots, y_t$. Since the adversary knows $y_1, \ldots, y_t$, they can compute the coefficients and find $a_1^{-1}(\Omega_{t-2} - a_0 V_0) = \alpha V_0$. Similarly, the coefificient of x^{t-3} is a polynomial $a_2\alpha^2 + a_1\alpha + a_0$; since the adversary knows all the coefficients *and* now they know αV_0, they can use Ω_{t-3} to compute $\alpha^2 V_0$.

Continuing, the adversary obtains $V_0, \alpha V_0, \ldots, \alpha^{t-1} V_0$ with $O(t^2)$ finite field and elliptic curve operations. From this they compute $\prod_{i=1}^{t-1}(y_t + \alpha)V_0$, which is a valid witness for y_t relative to the updated accumulator value V_t.

Thus, receiving the public batch update polynomial gives enough information to efficiently find an accumulated value y and a valid witness for it, breaking soundness of the accumulator as we define it.

A tempting mitigation is to require a valid membership proof from each user before responding with the batch update information; however, this fails. Once an adversary has a single valid ID-witness pair, they can successfully request updates and obtain valid ID-witness pairs for all users added to the accumulator

[4] Their more general update polynomial $v_{\mathcal{A},\mathcal{D}}(x)$ is also homogenous; we only analyze $v_{\mathcal{A}}(x)$ here for simplicity.

from that point. Then even if the adversary's original ID is deleted from the accumulator, they can continue to successfully request updates using any of the other pairs they obtained.

Of course, if the server locally computes the batch updates, this prevents the attack. But in this case the server could compute updates more easily by directly issuing a new witness $\frac{1}{y+\alpha}V$. This is also completely deanonymizing.

The definition of collision resistance in [VB20] (Definition 1) requires an adversary to produce both a membership proof and non-membership proof for the same element, which our attack does not allow. We claim that the missing piece in their definition is that they define the adversary's success in terms of an offline verification function, rather than in terms of passing the online verification protocol. Their definition forces the adversary to use the *same* element, whereas the verification protocol – being zero-knowledge – does not allow any such restrictions. In practice, it is likely a real security risk if someone can pass the verification protocol using a credential that does not "belong" to them in some sense. To us, this reinforces the need for extreme care in the definitions of anonymous primitives like this, and this is why we spend such effort defining security and anoymity in our protocol.

Replay Attacks. A much simpler real-world attack is for an adversary to play the role of a verifier, retrieve a zero-knowledge proof of membership, and present this to any other verifier. Nguyen's [Ngu05] verification protocol has a prover send a commitment, a verifier send a challenge in response, and a prover respond to the challenge. In this case an adversary could fool a verifier by mounting a "person in the middle" attack and forwarding all messages between the verifier and an honest prover.

Preventing this attack needs some method to ensure every proof is verifier-specific. For example, if verifiers had unique verifiable IDs, this could be part of the challenge given to provers. However, this requires another digital credential ecosystem for verifiers! More promising might be for the prover to include some unique aspect of the verification session, such as the IP address of the prover's TOR exit node. In ALLOSAUR, verifiers only accept proofs that respond fast enough to their challenge, which is secure if messages follow a consistent round structure.

Formalizing any of these mitigations in the single-server case is more difficult, so we do not include it. However, we hope that these problems motivate our complicated security definitions (and thus, unfortunately, a very long paper).

4 Multi-Party Framework

To fully describe the formalism for our multi-party setting requires 11 pages; we summarize the main ideas here instead. The formalism is in a "UC-style", where we have a control program coordinating messages in our proofs, an adversarial machine, and honest user machines.

The control program starts by creating N *server players*, of which the adversary corrupts at most $N/3$. All players have read access to a public, append-only message board. Mainly we require a consensus ordering and guaranteed delivery; to model this, we divide execution into rounds. All players can post to the public message board, but the control program stores all public messages posted in one round in an internal queue, and at the end of the round allows the adversary to arbitrarily permute the messages before adding them to the actual public message board. A practical implementation of a message board could be a blockchain with reasonable timing guarantees.

There are also private message boards between all players, which just model private communication channels with guaranteed delivery. Similarly, the adversary can permute messages in these boards arbitrarily, though it can only read the metadata of each message when doing so.

To execute functions, any player can post a function message to the message board. We divide functions into core accumulator functions (Gen, Add, Del, Wit_s) which only server players execute and which are executed atomically, and user functions (Wit_u, $\mathsf{Upd}_{s,u}$, Prove, Ver) which users execute simultaneously.

The adversary creates new honest user players by request, which prompts the user to post an Add message with their own ID. There is no verification of a "legitimate" Add; the accumulator will accept any Add message posted. This allows the adversary to create as many corrupted or honest user players as it wishes, though it cannot add or corrupt more server players than it chose at the beginning. In practice, this allows us to model corrupted issuers.

Tracking the accumulator is complicated, so rather than define an ideal functionality outright, we define an *observer* program which implicitly defines the ideal functionality through the messages it expects to see. When players post calls to Add or Del, the observer program tracks a set S which represents the elements that *should* be in the accumulator. It also stores all values of the accumulator which are posted. This is essential for us to prove soundness.

4.1 Security

In this framework, we need only three notions of security. Unfortunately, these notions of security are complicated and interactive, so we only outline them here.

Correctness. Whenever a user posts a Prove message to another user (which tells the second user to prove ownership of its credentials), this prompts the observer program to record a challenge. If an honest user (a prover) is requested to prove its credentials to another honest user (a verifier), then the observer program's challenge notes some time limit before which the verifier must post the results of the verification. If the user is in the accumulated set, the posted result must be "true", and otherwise "false". If the result is wrong or it is not posted soon enough, the observer program "fails". If the observer program fails, the adversary wins. We say an accumulator is *correct* if it does not fail against any PPT adversary.

This definition ensures that not only do honest users see the expected behaviour, but it also guarantees progress. If the adversary were able to stall the accumulator (say, by posting nonsense, or not posting at all) then this would delay the user from responding. We assume guaranteed delivery of messages so the adversary cannot mount denial of service attacks that would slow communication by too much.

Blind Commitment Soundness. We define blind commitment soundness using an extractor almost identically to the single-server case.

Indistinguishability. Our security definition for anonymity is by far the most complicated of our definitions. Privacy notions for digital credentials can be complicated and varied. We aimed for two notions of privacy: first, user interactions with the protocol should not reveal more information than necessary (e.g., users that prove ownership of credentials reveal that they are one of the users with valid credentials), and second, different user interactions should not be correlatable. The second goal is critical because out-of-band privacy loss happens in many use cases. For example, if valid digital credentials permit someone to vote, this interaction *must* be de-anonymized to prevent voter fraud. However, if that same person presents the same credentials moments later to buy cigarettes, no one should be able to connect the two events.

In our model, the control program manages *pseudonyms* for each user. When a user posts a message, they add a pseudonym to a "sender" field. Users can request as many pseudonyms as they wish and they can switch between them at any time. This models any internet anonymity protocol, such as TOR, I2P, mixnets, and so forth. We assume that, in practice, a user of ALLOSAUR would connect to the servers managing the accumulator through such a protocol.

Our definition of anonymity is then simply indistinguishability: the adversary should not be able to determine if two pseudonyms p_1 and p_2 belong to the same user. As stated, this is impossible to achieve because basic accumulator operations break anonymity. For example, if a user owns the ID y which is currently the only ID in the accumulator, then any verification of their credentials will immediately de-anonymize them, since they are the only user that could successfully verify.

Thus, we instead define anonymity sets. Each pseudonym p has an anonymity set consisting of user IDs y. The goal is that the anonymity set for p contains the IDs of all users which were equally likely to send all the messages that p sends. The observer program updates anonymity sets whenever it sees a message requesting a user to perform some operation.

We allow the adversary access to a function which partially de-anonymizes users: the adversary sends a list of IDs $\{y_1, \ldots, y_n\}$, each user (represented by their ID) is provided a new pseudonym $\{p_1, \ldots, p_n\}$, and the new pseudonyms are returned in a random order to the adversary. This initializes the anonymity set of each pseudonym p_i to $\{y_1, \ldots, y_n\}$.

Given all of this, the baseline probability that p_1 and p_2 belong to the same

user is $\frac{|A_1|\cdot|A_2|}{|A_1\cap A_2|}$, where A_1 and A_2 are the anonymity sets of p_1 and p_2 respectively. Thus, we say an accumulator is *indistinguishable* if the adversary cannot correlate two pseudonyms with significantly higher probability than this.

This construction means that the construction of the anonymity sets is *very* protocol-specific, so, while we formally define indistinguishability, we leave it to the protocol designer to define a function to manage anonymity sets. This means one could define an accumulator which creates singleton anonymity sets for each message. The baseline probability defined above is 1, so technically this is "indistinguishable", but obviously not anonymous. Rather than explicitly define a meaning for "anonymous", we leave that to the protocol. This forces a protocol specification to include a precise accounting of the anonymity loss from each message.

Modelling precisely how an adversary can trigger user behaviour is challenging. In real life, an adversary may know that a specific user, or maybe a certain demographic of users, will attempt to verify at a certain place at a certain time. There are many different ways to model this. For example, an adversary could send a set of users and the control program could randomly select one. However, this prevents an adversary from ensuring that *distinct* users perform some action, so we would then want to give the adversary the ability to exclude users that were previously selected from being selected again. Such attempts quickly become complicated, and we would not be surprised if there were a way to make it computationally infeasible to track anonymity sets with these more complicated definitions.

We opted for a fairly simple approach, where an adversary makes requests to pseudonyms for which it has some de-anonymizing information. This means users will not be anonymous unless the adversary sends large lists of IDs to be de-anonymized, and requests many of them to perform the same function. In our formalism, an adversary must do this if it wants to win the indistinguishability game, since it cannot win if each message's anonymity set contains only a single user! We formally define and spend a lot of time explaining our approach to anonymity in the appendix, so we encourage an interested reader to peruse that.

4.2 ALLOSAUR

Our full protocol ALLOSAUR, for "**A**ccumulator with **L**ow-**L**atency **O**blivious **S**ublinear **A**nonymous credential **U**pdates with **R**evocations" is described and implemented in the appendix. It starts from the single-server protocol. The secret keys of the accumulator, α and s_m, are shared among N server players, and the inversions necessary for Wit_s and Add are computed using standard MPC techniques (i.e. Beaver triples).

A protocol like MASCOT [KOS16] can supply the Beaver triples. Because the accumulator functionality in our protocol can verify each MPC computation (e.g., pairings can verify that Del was performed correctly), we do not need the overhead of something like SPDZ [DKL$^+$13]. In fact, we only need two verification components: a secure method to open secret shares (which can be done in 2 rounds using any commitment scheme), and commitments to the

randomness used to generate the Beaver triples. Together these not only give us secure MPC, but also identifiable abort. This means that if some dishonest players deviate from the protocol, we can identify at least one of these players.

The identifiable abort requires revealing the trapdoors of the accumulator. We assume that in practice, the consequences of being blamed by this abort process will be severe enough to deter such behaviour, even for a denial of service attack. An identifiable abort which does not reveal these secrets may yet be possible, since commitments to Beaver triples could be verified by pairings. We leave this to future work.

Witness Updates. The main deviation in ALLOSAUR from the single server protocol is the witness updates. We use a simple method for oblivious polynomial evaluation of the polynomials $d(y)$ and $v(y, \alpha)$. A user divides the values y, y^2, ..., y^k with an affine secret sharing scheme and sends one share of each power to each server player. As long as the polynomials $d(x)$ and $v(x, \alpha)$ have degree at most k, each server can compute a share of $d(y)$ by multiplying the ith coefficient of d by their share of y^i and summing the results. As the secret sharing scheme is affine, when the user reconstructs the returned shares, they will obtain $d(y)$. A similar technique works for $v(y, \alpha)V$, except the coefficients of the polynomial will be elliptic curve points. In this way a user sends k shares to each server (each in $\mathbb{F}_q$), and each server responds with a pair of shares, in $F_q \times G_1$.

Like our single-server approach, ALLOSAUR partitions larger updates into many update polynomials of degree k. The full updates just repeats the above method; however, notice that once the server has shares of $y, y^2, \ldots, y^k$, they can use these *same* shares to evaluate each update polynomial. This means with one message of k shares from the user, the server can compute mk updates for arbitrary m. The server will return m pairs of shares back to the user. The user then reconstructs each pair of evaluated update polynomials and applies them sequentially.

Using t-out-of-N Shamir secret-sharing, for D deletions we take $k \approx \sqrt{D}$ and this requires only $O(N\sqrt{D})$ communication and $O(t^2\sqrt{D})$ computation for the user. In contrast, our single-server method requires $O(D)$ communication and $O(D)$ user-side computation. The multi-party approach provides just as much anonymity if at least $N - t$ server players are honest. In practice, the servers may also run the message board using some consensus algorithm which will require some honest majority assumption, so assuming a constant fraction of the servers are honest is a small extra assumption.

Because the coefficients of the polynomials $d(x)$ and $v(x, \alpha)$ are entirely computed from public data, there is no security or integrity risk from treating the user as a trusted third party. If the user sends invalid shares of $y, y^2, \ldots$, this will only cause problems for that user (they will be blamed). This means we can use bare Shamir secret sharing.

For integrity of the result, a user can wait for $t + 1$ servers to return shares. If all the shares are points on the same polynomial of degree t, then the user

is assured that they were computed correctly, since we assume an honest majority. If some shares do not fit the same polynomial, the user will download all the public data used for the update, allowing them to compute precisely the share each server was *supposed* to send. All private messages must be signed; therefore, the user can post the signed messages from any server that did not compute the correct polynomial, which provides a reliable blame mechanism for the other servers.

We assume honest servers are also "live", meaning they return their shares within some bounded time (in our formalism, 1 round). Thus, a user will immediately receive shares from all honest servers, but may not receive any from corrupted servers. At some point the user must time out and use the shares it did receive, which must contain all honest shares by the liveness property. Corrupted servers cannot deny service to the user, since the user only needs t shares to finish the update, and we assume at least t honest server players.

We see that user availability needs at least t honest servers but user privacy needs at least $N-t$ honest players; it may be that $N-t < t$. While our formalism considers only fully active adversaries, ALLOSAUR may still be robust against a malicious majority if fewer than $N - t$ are actively malicious and the rest are only honest-but-curious.

If a user is deleted, they will find $d(y) = 0$ for one of the returned polynomials and their update will fail. In this case the user checks the public message board, to see if they were re-added to the accumulator. In ALLOSAUR the public messages to perform an addition will post part of a witness to the message board, and the user can simply download this if it exists. From there, they can update from this new witness.

Anonymity. Users send two types of messages in ALLOSAUR: updates and verifications. Verifications are zero-knowledge [Ngu05] and the update protocol above only sends randomized secret shares which are indistinguishable from random. Thus, the only de-anonymizing data is the metadata. Here we describe the anonymity loss of this metadata.

An update must specify from which epoch the user updates in order to know how to construct the update polynomials. Hence, when a user starts an update from epoch n_1 to epoch n_2, their anonymity set must be restricted to users whose last update brought them to epoch n_1, excluding users who have since updated from that point.

Further, if a user was deleted between n_1 and n_2, but re-added at some $n' < n_2$, the update polynomial does not work. The user must find a new witness from the addition and update it from epoch n' to epoch n_2. This completely deanonymizes them, since this is the only case when users request update polynomials twice in a row, and each epoch adds at most one user.

To obtain anonymity, we thus in practice recommend fixed intervals where users coordinate to update their witnesses; for example, at the start of each day. In fact, a protocol could facilitate this by only allowing updates at certain intervals.

The anonymity set of a verification message will include either all users with IDs currently in the accumulator (if the verification succeeds) or all users not in the accumulator (if it fails). More than that, however, if a user must verify their credentials against epoch n, but they only have a witness valid for epoch $n_0 < n$, they must update their witness before verify. Hence, their anonymity set becomes restricted in the same way as for an update.

Because there is such an anonymity loss from the timing of updates, ALLOSAUR specifies that users *discard* their updated witness after a verification.

In the appendix we prove that these ideas capture all the metadata anonymity loss from ALLOSAUR. In the end, these anonymity sets are intersected with the direct de-anonymizing data the adversary requested. In practice, we interpret this to mean that whatever de-anonymizing information an adversary already has (e.g., certain types of people tend to use their credentials in this certain place), the descriptions above show how much more anonymity they lose from using the accumulator.

We emphasize here that anonymity sets are defined *for each pseudonym*. This means that even if one pseudonym has been mostly de-anonymized to a user with ID y, that same user's anonymity can "heal": future messages from that user with different pseudonyms can have much larger anonymity sets. We define this rigorously in the appendix and note that our extremely complicated anonymity formalisms are the primary reason for the length of this paper.

Rounds. Our anonymity definition requires fairly coarse "rounds", so that users essentially synchronize their messags. This is a necessity given how we model our adversary. Our adversary chooses exactly which user should perform a specific function, so we must synchronize the responses to these requests.

This gives a lot of power to the adversary. In real life, if an adversary knows precisely which user is about to perform a function, then they have *already* de-anonymized that user. The notion we would like to capture is precisely how much extra information an adversary obtains. If the adversary has some distribution over possible users that might perform an action, then the protocol should reveal no extra information (besides certain inevitable information like whether the user has a valid credential, etc.).

Rounds also prevent replay attacks. Verifiers send a random challenge string to be included in each proof, preventing adversaries from storing a proof from an honest user, but verifiers also require the response to return in one round. An adversary cannot execute a person-in-the-middle attack because they would need another round to forward that challenge to an honest prover. This timing requirement may not be realistic, and a practical implementation that needs to prevent replay attacks that occur in a very short time window might need a different approach to thwart replay attacks. We emphasize that *some* replay prevention is necessary for many applications.

5 Performance

We wrote a proof-of-concept implemention of the update protocol of ALLOSAUR in Rust[5]. It uses a curve of order $\approx 2^{255}$ in BLS family [BLS03] with embedding degree 12 over a 381-bit prime field.

Performance of the core accumulator functions will likely depend enormously on the network and the public message board. Moreover, the main online computation of our core accumulator functions – a field inversion – is essentially identical to the Delete operation in [HKRW21], so we we expect similar performance. They report, for 5 servers with a dishonest majority in a simulation of a WAN, an average online time of about 1 second with 370 bytes of communication, and around 2 seconds and 165 kB of communication for the offline precomputation (i.e., of Beaver triples).

Our update protocol has three stages: a user pre-computes powers of their share and divides them into Shamir secret shares, which they send to the servers. The servers then compute the full update polynomial on the shares, and return them to the user. Finally, the user assembles the results into the full update.

We benchmarked each stage separately on an Intel Core i7-8750H 2.20 GHz CPU, as well as the single-server protocol of Section 3. To compare with [VB20], we used the Rust implementation of [Lod21]. Table 1 shows the computation time and communication costs.

Our optimized single-server update barely increases communication but vastly improves computation. ALLOSAUR is a strict improvement over both for updates of 100 or more changes, showing major benefits in user-side computation and communication, especially for larger updates. Communication costs in ALLOSAUR are higher for small updates due to the use of multiple servers. These would increase further if a system chose to use more servers.

Acknowledgements. We thank Arnab Roy for helpful early discussions, and Fujitsu Research of Europe and America for supporting the start of this research.

References

[0xa] 0xaf115b18eE30734f6CeA1C56BE76615df046e010. Welcome to the disco!!

[ABC+12] Jae Hyun Ahn, Dan Boneh, Jan Camenisch, Susan Hohenberger, abhi shelat, and Brent Waters. Computing on authenticated data. In Ronald Cramer, editor, *TCC 2012*, volume 7194 of *LNCS*, pages 1–20. Springer, Heidelberg, March 2012.

[Agi22] Sergey Agievich. Blind accumulators for e-voting. Cryptology ePrint Archive, Report 2022/373, 2022. `https://eprint.iacr.org/2022/373`.

[5] Code available at `https://github.com/sam-jaques/allosaurust`

Protocol	Modifications		Computations		Communication	
	Add	Del	User	Server	User	Server
[VB20]	0	10	4 ms	4 ms	< 0.01	0.81
	10	10	4 ms	4 ms		1.13
	0	100	37 ms	49 ms		8.01
	100	100	37 ms	61 ms		11.21
	0	1000	368 ms	11 196 ms		80.01
	1000	1000	369 ms	21 768 ms		112.01
Single-server (Section 3)	(any)	10	4 ms	4 ms	< 0.01	0.81
	(any)	100	38 ms	41 ms		8.01
	(any)	1000	376 ms	436 ms		80.12
	(any)	10 000	3 934 ms	4 306 ms		801.20
ALLOSAUR	(any)	10	5 ms	4 ms	0.82	0.82
	(any)	100	18 ms	37 ms	2.59	2.43
	(any)	1000	52 ms	371 ms	8.03	8.03
	(any)	10 000	172 ms	3 767 ms	25.58	25.23

Table 1: Performance benchmarks averaged over 50 samples. Communication is in kB. For ALLOSAUR, we assumed 5 servers with a threshold of 3. The communication costs include communication to and from all servers. Servers do not need to communicate with each other during an update.

[BBDE19] Johannes Blömer, Jan Bobolz, Denis Diemert, and Fabian Eidens. Updatable anonymous credentials and applications to incentive systems. In Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, *ACM CCS 2019*, pages 1671–1685. ACM Press, November 2019.

[BBF19] Dan Boneh, Benedikt Bünz, and Ben Fisch. Batching techniques for accumulators with applications to IOPs and stateless blockchains. In Alexandra Boldyreva and Daniele Micciancio, editors, *CRYPTO 2019, Part I*, volume 11692 of *LNCS*, pages 561–586. Springer, Heidelberg, August 2019.

[BCD+17] Foteini Baldimtsi, Jan Camenisch, Maria Dubovitskaya, Anna Lysyanskaya, Leonid Reyzin, Kai Samelin, and Sophia Yakoubov. Accumulators with applications to anonymity-preserving revocation. In *2017 IEEE European Symposium on Security and Privacy (EuroS P)*, pages 301–315, 2017.

[BCF+21] Daniel Benarroch, Matteo Campanelli, Dario Fiore, Kobi Gurkan, and Dimitris Kolonelos. Zero-knowledge proofs for set membership: Efficient, succinct, modular. In Nikita Borisov and Claudia Diaz, editors, *Financial Cryptography and Data Security*, pages 393–414, Berlin, Heidelberg, 2021. Springer Berlin Heidelberg.

[BCMS20] Benedikt Bünz, Alessandro Chiesa, Pratyush Mishra, and Nicholas Spooner. Recursive proof composition from accumulation schemes. In Rafael Pass and Krzysztof Pietrzak, editors, *TCC 2020, Part II*, volume 12551 of *LNCS*, pages 1–18. Springer, Heidelberg, November 2020.

[BCY20] Foteini Baldimtsi, Ran Canetti, and Sophia Yakoubov. Universally composable accumulators. In Stanislaw Jarecki, editor, *CT-RSA 2020*, volume 12006 of *LNCS*, pages 638–666. Springer, Heidelberg, February 2020.

[Bd94] Josh Cohen Benaloh and Michael de Mare. One-way accumulators: A decentralized alternative to digital sinatures (extended abstract). In Tor Helleseth, editor, *EUROCRYPT'93*, volume 765 of *LNCS*, pages 274–285. Springer, Heidelberg, May 1994.

[BDET19] Dmytro Bogatov, Angelo De Caro, Kaoutar Elkhiyaoui, and Björn Tackmann. Anonymous transactions with revocation and auditing in hyperledger fabric. Cryptology ePrint Archive, Report 2019/1097, 2019. `https://eprint.iacr.org/2019/1097`.

[Bea92a] Donald Beaver. Efficient multiparty protocols using circuit randomization. In Joan Feigenbaum, editor, *CRYPTO'91*, volume 576 of *LNCS*, pages 420–432. Springer, Heidelberg, August 1992.

[Bea92b] Donald Beaver. Foundations of secure interactive computing. In Joan Feigenbaum, editor, *CRYPTO'91*, volume 576 of *LNCS*, pages 377–391. Springer, Heidelberg, August 1992.

[BIM00] Amos Beimel, Yuval Ishai, and Tal Malkin. Reducing the servers computation in private information retrieval: PIR with preprocessing. In Mihir Bellare, editor, *CRYPTO 2000*, volume 1880 of *LNCS*, pages 55–73. Springer, Heidelberg, August 2000.

[BLL00] Ahto Buldas, Peeter Laud, and Helger Lipmaa. Accountable certificate management using undeniable attestations. In Dimitris Gritzalis, Sushil Jajodia, and Pierangela Samarati, editors, *ACM CCS 2000*, pages 9–17. ACM Press, November 2000.

[BLS03] Paulo S. L. M. Barreto, Ben Lynn, and Michael Scott. Constructing elliptic curves with prescribed embedding degrees. In Stelvio Cimato, Clemente Galdi, and Giuseppe Persiano, editors, *SCN 02*, volume 2576 of *LNCS*, pages 257–267. Springer, Heidelberg, September 2003.

[BN06] Mihir Bellare and Gregory Neven. Multi-signatures in the plain public-key model and a general forking lemma. In Ari Juels, Rebecca N. Wright, and Sabrina De Capitani di Vimercati, editors, *ACM CCS 2006*, pages 390–399. ACM Press, October / November 2006.

[BOS16] Carsten Baum, Emmanuela Orsini, and Peter Scholl. Efficient secure multiparty computation with identifiable abort. In Martin Hirt and Adam D. Smith, editors, *TCC 2016-B, Part I*, volume 9985 of *LNCS*, pages 461–490. Springer, Heidelberg, October / November 2016.

[BUV21] Alex Biryukov, Aleksei Udovenko, and Giuseppe Vitto. Cryptanalysis of a dynamic universal accumulator over bilinear groups. In Kenneth G. Paterson, editor, *CT-RSA 2021*, volume 12704 of *LNCS*, pages 276–298. Springer, Heidelberg, May 2021.

[Can01] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *42nd FOCS*, pages 136–145. IEEE Computer Society Press, October 2001.

[CGKS95] Benny Chor, Oded Goldreich, Eyal Kushilevitz, and Madhu Sudan. Private information retrieval. In *36th FOCS*, pages 41–50. IEEE Computer Society Press, October 1995.

[CH10] Philippe Camacho and Alejandro Hevia. On the impossibility of batch update for cryptographic accumulators. In Michel Abdalla and Paulo S. L. M. Barreto, editors, *LATINCRYPT 2010*, volume 6212 of *LNCS*, pages 178–188. Springer, Heidelberg, August 2010.

[CHK22] Henry Corrigan-Gibbs, Alexandra Henzinger, and Dmitry Kogan. Single-server private information retrieval with sublinear amortized time. In *EUROCRYPT 2022, Part II*, LNCS, pages 3–33. Springer, Heidelberg, June 2022.

[CJ10] Sébastien Canard and Amandine Jambert. On extended sanitizable signature schemes. In Josef Pieprzyk, editor, *CT-RSA 2010*, volume 5985 of *LNCS*, pages 179–194. Springer, Heidelberg, March 2010.

[CKS09] Jan Camenisch, Markulf Kohlweiss, and Claudio Soriente. An accumulator based on bilinear maps and efficient revocation for anonymous credentials. In Stanislaw Jarecki and Gene Tsudik, editors, *PKC 2009*, volume 5443 of *LNCS*, pages 481–500. Springer, Heidelberg, March 2009.

[CL02] Jan Camenisch and Anna Lysyanskaya. Dynamic accumulators and application to efficient revocation of anonymous credentials. In Moti Yung, editor, *CRYPTO 2002*, volume 2442 of *LNCS*, pages 61–76. Springer, Heidelberg, August 2002.

[Din] Pamela Dingle. Ion – we have liftoff!

[DKL+13] Ivan Damgård, Marcel Keller, Enrique Larraia, Valerio Pastro, Peter Scholl, and Nigel P. Smart. Practical covertly secure MPC

for dishonest majority - or: Breaking the SPDZ limits. In Jason Crampton, Sushil Jajodia, and Keith Mayes, editors, *ESORICS 2013*, volume 8134 of *LNCS*, pages 1–18. Springer, Heidelberg, September 2013.

[DMO00] Giovanni Di Crescenzo, Tal Malkin, and Rafail Ostrovsky. Single database private information retrieval implies oblivious transfer. In Bart Preneel, editor, *EUROCRYPT 2000*, volume 1807 of *LNCS*, pages 122–138. Springer, Heidelberg, May 2000.

[DMS04] Roger Dingledine, Nick Mathewson, and Paul F. Syverson. Tor: The second-generation onion router. In Matt Blaze, editor, *USENIX Security 2004*, pages 303–320. USENIX Association, August 2004.

[DPSZ12] Ivan Damgård, Valerio Pastro, Nigel P. Smart, and Sarah Zakarias. Multiparty computation from somewhat homomorphic encryption. In Reihaneh Safavi-Naini and Ran Canetti, editors, *CRYPTO 2012*, volume 7417 of *LNCS*, pages 643–662. Springer, Heidelberg, August 2012.

[DRS18] David Derler, Sebastian Ramacher, and Daniel Slamanig. Post-quantum zero-knowledge proofs for accumulators with applications to ring signatures from symmetric-key primitives. In Tanja Lange and Rainer Steinwandt, editors, *Post-Quantum Cryptography - 9th International Conference, PQCrypto 2018*, pages 419–440. Springer, Heidelberg, 2018.

[Fou] Sovrin Foundation. Sovrin network now ready for digital credential issuers.

[FVY14] Conner Fromknecht, Dragos Velicanu, and Sophia Yakoubov. A decentralized public key infrastructure with identity retention. Cryptology ePrint Archive, Report 2014/803, 2014. `https://eprint.iacr.org/2014/803`.

[GGM14] Christina Garman, Matthew Green, and Ian Miers. Decentralized anonymous credentials. In *NDSS 2014*. The Internet Society, February 2014.

[GV20] Rishab Goyal and Satyanarayana Vusirikala. Verifiable registration-based encryption. In Daniele Micciancio and Thomas Ristenpart, editors, *CRYPTO 2020, Part I*, volume 12170 of *LNCS*, pages 621–651. Springer, Heidelberg, August 2020.

[Hel] Nader Helmy. Why we're launching mattr vii.

[HKRW21] Lukas Helminger, Daniel Kales, Sebastian Ramacher, and Roman Walch. Multi-party revocation in sovrin: Performance through

distributed trust. In Kenneth G. Paterson, editor, *CT-RSA 2021*, volume 12704 of *LNCS*, pages 527–551. Springer, Heidelberg, May 2021.

[HRMM18] Michael Hölzl, Michael Roland, Omid Mir, and René Mayrhofer. Bridging the gap in privacy-preserving revocation: Practical and scalable revocation of mobile eids. In *Proceedings of the 33rd Annual ACM Symposium on Applied Computing*, SAC '18, page 1601–1609, New York, NY, USA, 2018. Association for Computing Machinery.

[HRMM19] Michael Hölzl, Michael Roland, Omid Mir, and René Mayrhofer. Disposable dynamic accumulators: toward practical privacy-preserving mobile eIDs with scalable revocation. *International Journal of Information Security*, 19(4):401–417, July 2019.

[KB21] Ioanna Karantaidou and Foteini Baldimtsi. Efficient constructions of pairing based accumulators. In Ralf Küsters and Dave Naumann, editors, *CSF 2021 Computer Security Foundations Symposium*, pages 1–16. IEEE Computer Society Press, 2021.

[KOS16] Marcel Keller, Emmanuela Orsini, and Peter Scholl. MASCOT: Faster malicious arithmetic secure computation with oblivious transfer. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *ACM CCS 2016*, pages 830–842. ACM Press, October 2016.

[LHMP21] Magnus Lofstrom, Joseph Hayes, Brandon Martin, and Deepak Premkumar. Racial disparities in law enforcement stops, 2021.

[LLNW16] Benoît Libert, San Ling, Khoa Nguyen, and Huaxiong Wang. Zero-knowledge arguments for lattice-based accumulators: Logarithmic-size ring signatures and group signatures without trapdoors. In Marc Fischlin and Jean-Sébastien Coron, editors, *EUROCRYPT 2016, Part II*, volume 9666 of *LNCS*, pages 1–31. Springer, Heidelberg, May 2016.

[Lod21] Michael Lodder. accumulator-rs. `https://github.com/mikelodder7/accumulator-rs`, 2021.

[Mer88] Ralph C. Merkle. A digital signature based on a conventional encryption function. In Carl Pomerance, editor, *CRYPTO'87*, volume 293 of *LNCS*, pages 369–378. Springer, Heidelberg, August 1988.

[MGGM18] Alexander Mühle, Andreas Grüner, Tatiana Gayvoronskaya, and Christoph Meinel. A survey on essential components of a self-sovereign identity. *Computer Science Review*, 30:80–86, 2018.

[Ngu05] Lan Nguyen. Accumulators from bilinear pairings and applications. In Alfred Menezes, editor, *CT-RSA 2005*, volume 3376 of *LNCS*, pages 275–292. Springer, Heidelberg, February 2005.

[NN98] Kobbi Nissim and Moni Naor. Certificate revocation and certificate update. In Aviel D. Rubin, editor, *USENIX Security 98*. USENIX Association, January 1998.

[oBC] Government of British Columbia. Bc digital trust: Confident, confidential communication online.

[oO] Government of Ontario. Digital id in ontario.

[OWWB20] Alex Ozdemir, Riad S. Wahby, Barry Whitehat, and Dan Boneh. Scaling verifiable computation using efficient set accumulators. In Srdjan Capkun and Franziska Roesner, editors, *USENIX Security 2020*, pages 2075–2092. USENIX Association, August 2020.

[Pol] Polygon Team. Introducing Polygon ID, zero-knowledge identity for web3.

[PS14] Henrich Christopher Pöhls and Kai Samelin. On updatable redactable signatures. In Ioana Boureanu, Philippe Owesarski, and Serge Vaudenay, editors, *ACNS 14*, volume 8479 of *LNCS*, pages 457–475. Springer, Heidelberg, June 2014.

[Res] Brand Essence Research. Digital identity solutions market: Global size, trends, competitive, historical & forecast analysis, 2021-2028.

[SGS20] Gili Schul-Ganz and Gil Segev. Accumulators in (and beyond) generic groups: Non-trivial batch verification requires interaction. In Rafael Pass and Krzysztof Pietrzak, editors, *TCC 2020, Part II*, volume 12551 of *LNCS*, pages 77–107. Springer, Heidelberg, November 2020.

[Sha79] Adi Shamir. How to share a secret. *Communications of the Association for Computing Machinery*, 22(11):612–613, November 1979.

[SLS+] Manu Sporny, Dave Longley, Markus Sabadello, Drummond Reed, Orie Steele, and Christopher Allen. Decentralized identifiers (dids) v1.0.

[Smi] Samuel M. Smith. Key event receipt infrastructure (keri) design.

[Spr] Spruce. Spruce developer update # 1.

[ST99a] Tomas Sander and Amnon Ta-Shma. Auditable, anonymous electronic cash. In Michael J. Wiener, editor, *CRYPTO'99*, volume 1666 of *LNCS*, pages 555–572. Springer, Heidelberg, August 1999.

[ST99b] Tomas Sander and Amnon Ta-Shma. Flow control: A new approach for anonymity control in electronic cash systems. In Matthew Franklin, editor, *FC'99*, volume 1648 of *LNCS*, pages 46–61. Springer, Heidelberg, February 1999.

[Tan] Elizabeth M. Tanner. State of rhode island: Digital government with identity blockchain.

[VB20] Giuseppe Vitto and Alex Biryukov. Dynamic universal accumulator with batch update over bilinear groups. Cryptology ePrint Archive, Report 2020/777, 2020. `https://eprint.iacr.org/2020/777`.

[zCl] zCloak Network. zCloak network: a technical overview.

6 Private Information Retrieval

Following the definition from [DMO00], a *private information retrieval* (PIR) scheme is a triple $(\mathsf{Data}, \mathsf{Query}, \mathsf{Rec})$ of probabilistic algorithms (implicitly parameterized by 1^λ). Imagine that a server runs Data and a user runs Query and Rec. They have the following form:

- $\mathsf{Data}(D)$ takes as input a database D, and interacts with $\mathsf{Query}(n, i)$ which takes in the database size $n = |D|$ and an index i, and they output a hint h_u for the user making the query.
- $\mathsf{Rec}(n, i, h_u) \to \{0, 1\}$ takes the length of the database, the requested index, and the hint(s) from the interactive protocol, and recovers the ith element of the database D.

These algorithms satisfy two security requirements

Correctness : For any $n \in \mathbb{N}$, and any database $D \in \{0,1\}^n$ (represented as $D = (D_1, D_2, \ldots, D_n)$ for $D_i \in \{0,1\}$), then for all $i \in [n]$,

$$\Pr\left[\mathsf{Rec}(n, i, h_u) = D_i \middle| (h_u, h_s, \Lambda) \leftarrow \langle \mathsf{Query}(n, i), \mathsf{Data}(D) \rangle\right] \geq 1 - \mathrm{negl}(\lambda)$$

Privacy : For any $n \in \mathbb{N}$, any database $D \in \{0,1\}^n$, and any two $i, j \in [n]$, then for any PPT algorithm $\mathcal{A}$, we require

$$\begin{aligned} &\Pr\left[b = 1 \middle| (h_u, b, \Lambda) \leftarrow \langle \mathsf{Query}(n, i), \mathcal{A}(D) \rangle\right] \\ -&\Pr\left[b = 1 \middle| (h_u, b, \Lambda) \leftarrow \langle \mathsf{Query}(n, j), \mathcal{A}(D) \rangle\right] \leq \mathrm{negl}(\lambda) \end{aligned}$$

There is a strong similarity between PIR and blind updates. In the following we let C_A denote the cost of each subroutine A. Recall that an additive accumulator can add elements and a negative accumulator allows users to prove non-membership. We assume existence of an efficient, injective function from $[n]$ to the set of possible elements in the accumulator, which exists for all existing accumulator constructions.

Lemma 6.1. *Suppose there is an additive negative accumulator with blind updates (i.e., some maximum advantage $\alpha \leq \mathrm{negl}(\lambda)$), where the communication complexity of a blind witness update for m new additions and deletions is $c(m)$. Then for an n-bit database, there is a PIR with:*

- *Computational cost at most* $\mathsf{C}_{\mathsf{Gen}} + \mathsf{C}_{\mathsf{Add}}(n) + \mathsf{C}_{\mathsf{Upd}_s}(n)$ *for* Data.
- *Computational cost* $\mathsf{C}_{\mathsf{Wit}} + \mathsf{C}_{\mathsf{Upd}_u}(n) + \mathsf{C}_{\mathsf{Ver}}$ *for* Query.
- *Communication cost* $c(n) + |A| + |\mathsf{aux}_0| + |\mathsf{sk}|$ *(i.e., the length of an accumulator, its secret key, and the initial auxiliary data).*
- *Maximum advantage in the user privacy game of* 2α.

Proof. **PIR Construction**: Given a database $D \in \{0,1\}^n$, Data will first call $\mathsf{Gen}(n) \to (\mathsf{sk}, A_0, \mathsf{aux}_0)$ and send all of this output Query. Using an injective PRF f which maps $[n]$ to valid user IDs for the accumulator, Data will then call $\mathsf{Add}(\mathsf{sk}, A_i, f(j), \mathsf{aux}_i)$ to add $y_j := f(j)$ to the accumulator for all $D_j = 0$ in D, eventually producing a final accumulator value A_t with auxiliary update data aux_{upd}.

$\mathsf{Query}(n, i)$ will set $y_i = f(i)$ and locally run the protocol

$$\langle \mathsf{Wit}_u(x_i), \mathsf{Wit}_s(\mathsf{sk}, \mathsf{aux}_0, A_0) \rangle$$

to obtain w_i. It then begins the witness update protocol, running $\mathsf{Upd}_u(y_i, w_i, A_0)$. Data will run $\mathsf{Upd}_s(\mathsf{sk}, A_t, \mathsf{aux}_{upd})$. At the end of this interaction, the user running Query will have a new accumulator A_t, and a new witness w_i'. The hint they output is $h_u = (y_i, w_i', A_t)$. If any of these functions fail, Query halts.

Finally, $\mathsf{Rec}(h_u, n, i)$ will run $\mathsf{Ver}(y_i, w_i', A_t)$ and output the result.

Complexity: Data runs Gen once, runs Add at most n times, and Upd_s for an update of size at most n. Query runs Wit once (both sides), then Upd_u for an update of size at most n, and Ver. The communication is the communication of the blind update, plus Data must send a blank accumulator and its secret key.

Security: The completeness property of the accumulator and the injectivity of f guarantees that if $D_i = 0$, then y_i was added to the accumulator, so the non-membership witness must be invalid, and $\mathsf{Ver}(\mathsf{pk}, x_i, w_i', A') = 0$. Conversely, if $D_i = 1$, then x_i was not added to the accumulator, and the non-membership witness will stay valid, so $\mathsf{Ver}(\mathsf{pk}, x_i, w_i', A') = 1$.

For user privacy, we show how to construct an adversary $\mathcal{A}' = (\mathcal{A}_0', \mathcal{A}_1)$ out of the blindness of the accumulator from an adversary $\mathcal{A}$ against the PIR. Let i_0 and i_1 be a pair of indices for some database D for which $\mathcal{A}$ has the maximum advantage, meaning that $\mathcal{A}$ outputs 1 with probability p for index i_0 and outputs 1 with probability $p + \beta$ for index i_1, where β is the advantage of $\mathcal{A}$.

Since Query halts if given improperly-formatted initial data, we can assume $\mathcal{A}$ sends the expected initial data: an accumulator value A_0, a secret key sk, and auxiliary data aux_0. We construct $\mathcal{A}_0'$ by running the first stage of $\mathcal{A}$, which will output all this data initially. Given the indices i and j, Adv_0' sets

$y_0 = f(i_0)$ and $y_1 = f(i_1)$ and runs Wit_u and Wit_s to recover witnesses for each. Again, assuming Query halts if this fails, we can suppose these functions run successfully on the values provided by $\mathcal{A}$. This gives w_0 and w_1, which $\mathcal{A}'_0$ outputs along with A_0, sk, aux_0, y_0, y_1, and its entire internal state as the first step of the blind update game.

At this point we see that A_0 contains no elements, so $\mathsf{Ver}(y_b, w_b, A_0) = 1$ for both $b \in \{0, 1\}$. Thus, during the second stage of the blind update gate, Upd_u will interact exactly as $\mathsf{Query}(n, i_b)$, which can then be forwarded to $\mathcal{A}$. When $\mathcal{A}$ outputs $b' \in \{0, 1\}$, $\mathcal{A}'_1$ simply outputs thisas its response.

If $b = 0$ then $\mathcal{A}$ interacts with $\mathsf{Query}(n, i_0)$, $\mathcal{A}'$ wins if $\mathcal{A}$ outputs 0, which happens with probability $1 - p$. Instead if $b = 1$, $\mathcal{A}'$ wins when $\mathcal{A}$ outputs 1, which has probability $p + \beta$. The total probability of $\mathcal{A}'$ winning is then

$$\frac{1}{2}(1 - p) + \frac{1}{2}(p + \beta) = \frac{1}{2} + \frac{\beta}{2}.$$

and thus Adv' has advantage $\frac{\beta}{2}$ in the distinguishing game against the blind update, meaning $\beta \leq 2\alpha$. □

Corollary 6.2. *The same result holds for subtractive positive accumulators.*

Proof. A subtractive positive accumulator implies an additive negative accumulator by initializing the accumulator to contain all possible future elements. For PIR this is a well-known set: Data will add $f(i)$ to the accumulator for all $i \in [n]$ before sending it to the user. □

While the definition of a PIR we used does not distinguish pre-computation and online computation, we point out that this PIR can construct the accumulator as an offline precomputation, and only perform Upd during the online phase.

7 Multi-Party Formalism

Our formalism involves three types of program:

- the *control program*, which coordinates execution of the other players, relays messages, and maintains a representation of an ideal accumulator.
- the *players*, which represent honest servers and users.
- the *adversary*, which performs arbitrary, bounded computations.

A true UC formalism would cleanly separate the functionality of, e.g., digital signatures from the rest of the protocol. The control program should send data back and forth from players and adversary to other programs, which act independently. As much as is practical we follow this philosophy, but strict separation would bog down the description.

For example, authenticated channels with digital signatures prevent impersonation of messages, but adding even a formal functionality of digital signatures

is unnecessarily complex. Instead we break the control flow slightly and permit the control program and all non-accumulator funcationality access to a global variable representing the currently active player, to correctly decide who has sent each message.

We start by describing these other functionalities, which have well-known realizations. Throughout we use **coloured_text** for global variables.

7.1 Pseudonyms

To model anonymous internet connections, such as TOR [DMS04], we introduce *pseudonyms*.Pseudonyms in our formalism are unique IDs which a *pseudonym program* manages, assigning each pseudonym exclusively and irrevocably to one player. Players can request new pseudonyms at any time. Pseudonyms are managed through a dictionary **identities**, mapping each pseudonym to the player which owns it. When players post messages, they add one of their pseudonyms as a "sender" argument, and the pseudonym program assigns that sender to the message.

Though we assume messages are private, we also assume they are authenticated. This is why the pseudonyms are exclusive: a pseudonym represents a session with a particular signing key.

PSEUDONYM_SETUP (num_players)

1: **identities** $\leftarrow$ new empty dictionary
2: **for** $i = 1$ to num_players **do**
3: **identities**$[i] \leftarrow i$
4: **end for**

Program 1: Establishes a number of initial players (the servers) who will be given non-anonymous pseudonyms.

PSEUDONYM_GIVE (player)

1: $p \leftarrow_{\$} 2^{2\lambda} \setminus$ **identities**.keys
2: **identities**$[p] \leftarrow$ player
3: **RETURN**(p)

Program 2: Chooses an unused pseudonym, assigns it to the player index given as argument, and returns it. This function is not accessible to players.

PSEUDONYM_NEW ()

1: **return** **PSEUDONYM_GIVE**(**ACTIVE_PLAYER_ID**)

Program 3: Draws a random new unused pseudonym to assign to the currently active player, and returns it.

```
PSEUDONYM_VERIFY (pseudonym)
1: RETURN(identities[pseudonym] == ACTIVE_PLAYER_ID)
```

Program 4: Ensures the active player owns this pseudonym.

```
PSEUDONYM_GET ()
1: P ← {pseud : identities[pseud] = ACTIVE_PLAYER_ID }
2: RETURN(P)
```

Program 5: Returns all pseudonyms belonging to the active player.

```
PSEUDONYM_CHECK_CORRUPTED (pseudonym, corrupted_players)
1: RETURN(identities[pseudonym] ∈ corrupted_players)
```

Program 6: Checks whether a pseudonym belongs to a corrupted set.

For tracking anonymity, the pseudonym program will also return a set of pseudonyms belonging to a set of player IDs.

```
PSEUDONYMS_FROM_SET (players)
1: pseudonyms ← ∅
2: for (pseud, id) ∈ identities do
3:    if id ∈ players then
4:       pseudonyms.add(pseud)
5:    end if
6: end for
7: RETURN(pseudonyms)
```

Program 7: Returns all pseudonyms belonging to a set of players.

We will also allow the observer program to de-anonymize pseudonyms at will.

```
PSEUDONYM_GET_IDENTITY (pseudonym)
1: RETURN(identities[pseudonym])
```

Program 8: De-anonymize a single pseudonym.

7.2 Message Boards

We model a public, authenticated, consensus channel with guaranteed delivery as a *message board*. In our formalism, we allow all players to post to the message board and to read from it. However, the message board program does not immediately add each posted message to the board. Instead we allow the adversary to arbitrarily re-order messages between rounds. After each round, the order of the messages is fixed and all players are able to read them. This creates a somewhat artifical "round". In real terms, a round is the maximum delay that a message might face before reaching being added to the channel.

A player attempting to post on a message board must include a pseudonym to use. The message board program first requests that the pseudonym program verifies that this pseudonym belongs to the active player before attaching it to the message as metadata.

MESSAGE_BOARD_SETUP ()

1: // Initialize a dynamic array to represent committed values.
2: **BOARD** ←empty dynamic array
3: // Initialize a dynamic array to represent uncommitted values.
4: **TEMP_BOARD** ←empty dynamic array.

Program 9: Initialize public message board.

MESSAGE_BOARD_POST (message, sender)

1: // Ensure the sender (a pseudonym) is the current active player
2: **if** not **PSEUDONYM_VERIFY**(sender) **then**
3: **RETURN**
4: **end if**
5: // Add message to the temporary array.
6: **TEMP_BOARD**.Add(message, sender)

Program 10: Send a message to the public bulletin board, which may be reordered before the end of the round.

MESSAGE_BOARD_DELIVER ()

1: $\pi \leftarrow$ **ADVERSARY.reorder** (**TEMP_BOARD**)
2: Rearrange the messages in **TEMP_BOARD** by π
3: Add the contents of **TEMP_BOARD** to **BOARD**, in order
4: **TEMP_BOARD** ← empty dynamic array

Program 11: Asks the adversary for a permutation, permutes the messages, then posts them to the permanent message board.

MESSAGE_BOARD_PULL (index)

1: **if** $|\mathbf{BOARD}| \leq$ index **then**
2: **RETURN**(**BOARD**.index).
3: **else**
4: **RETURN**($\perp$)
5: **end if**

Program 12: Returns a message from the public bulletin board. Messages are requested by an index i, requiring players to track the last message they saw, but since they can download the entire message board, this is equivalent to other data structures. Since users can process data locally, this is also equivalent to more complicated requests, such as "retrieve all messages matching some criteria", which a real-world host might allow.

Private message boards. The message board program maintains a dictionary of *private message boards*, indexed by players' pseudonyms. As with the public board, any player can post to the private message board for any pseudonym, but only the player which owns that pseudonym can read the messages on that board, and they do so via a similar interface to the public board. Since players may have many pseudonyms, they may have many private message boards. Private message boards represent encrypted, available, authenticated channels. Critically, we assume all messages are signed, so that players can reveal the private messages if they need to prove dishonest behaviour from another player. The **PRIVATE_MESSAGE_BOARD_REVEAL** function represents this

When the adversary is allowed to reorder *private* messages, they are allowed to see the senders and the lengths of each message, but not the contents. The ordering in which messages are posted provides some de-anonymizing information to the adversary. In some cases this gives no advantage; in others we need to account for this. To avoid these issues, the message board program shuffles private messages before sending to the adversary. This is more of a relic of the formalism than a real-world capability: in practice, the adversary might have access to far more fine-grained timing information, but the timing of messages sent will also be much less informative, since different players will run asynchronously. Moreoever, our formalism does not allow the adversary to correlate timings between different message boards, which would be possible (to some extent) in real life.

PRIVATE_MESSAGE_BOARD_SETUP ()

1: // Represents committed values.
2: **PRIVATE_BOARDS** $\leftarrow$empty dictionary
3: // Represents uncommitted values.
4: **PRIVATE_TEMP_BOARDS** $\leftarrow$empty dictionary

```
5: // Represents metadata of each message.
6: MESSAGE_HEADERS ← empty dictionary
```

Program 13: Initialize private message boards.

```
PRIVATE_MESSAGE_BOARD_POST(
        message, sender, receiver)
1: // Ensure the sender (a pseudonym) is the current active player
2: if not PSEUDONYM_VERIFY(sender) then
3:    RETURN
4: end if
5: // Create new message boards for new pseudonyms.
6: if receiver ∉ PRIVATE_TEMP_BOARDS.keys then
7:    PRIVATE_TEMP_BOARDS.add(receiver, empty queue)
8:    MESSAGE_HEADERS.add(receiver, empty queue)
9: end if
10: // Add m to the temporary array.
11: PRIVATE_TEMP_BOARDS[receiver].Add(
        message, sender).
12: MESSAGE_HEADERS[receiver].Add(
        round_num, sender, length(message)).
```

Program 14: Posts a message to a pseudonymous private message board. It maintains a list of headers so that the adversary can later view metadata of each message, but not the internal data.

```
PRIVATE_MESSAGE_BOARD_DELIVER ()
1: for i = 0 to |PRIVATE_BOARDS| do
2:    σ ← random shuffle
3:    Rearrange elements of MESSAGE_HEADERS[i] by σ
4:    π ← ADVERSARY.reorder (MESSAGE_HEADERS[i])
5:    Rearrange the messages in PRIVATE_TEMP_BOARDS[i] by π∘σ
6:    Add the contents of PRIVATE_TEMP_BOARDS[i] to
      PRIVATE_BOARDS[i], in order
7:    PRIVAET_TEMP_BOARDS[i] ← empty dynamic array
8:    MESSAGE_HEADERS[i] ← empty dynamic array
9: end for
```

Program 15: Delivers private messages from temporary boards to permanent boards, with adversarial ordering based on metadata. Messages are shuffled before delivered to the adversary; otherwise, there are some timing vulnerabilities to anonymity.

PRIVATE_MESSAGE_BOARD_PULL(index, receiver)

1: **if** not **PSEUDONYM_VERIFY**(receiver) **then**
2: **RETURN**
3: **end if**
4: **RETURN**(**PRIVATE_BOARDS**[receiver][index]).

Program 16: Retrieve messages from a private message board, ensuring that the requesting player has read access to the board.

PRIVATE_MESSAGE_BOARD_REVEAL(receiver, index)

1: **if** **PSEUDONYM_VERIFY**(receiver) **then**
2: **RETURN**
3: **end if**
4: **BOARD**.append(**PRIVATE_BOARDS**[receiver][index]).

Program 17: Reveal private messages. Models the function of digital signatures

Observer Functions. We will include some extra functions beyond the normal function of a message board that the accumulator control program will use to manage anonymity. One will return the set of all pseudonyms that have sent any messages in the last round, from both private and public message boards, and the other returns all posted messages of a specific type (e.g., function messages).

MESSAGE_BOARD_GET_SENDERS()

1: $P \leftarrow$ all senders of all messages in **TEMP_BOARD**
2: **for** private_board in **PRIVATE_TEMP_BOARDS** **do**
3: $P \leftarrow P \cup$ all senders of all messages in private_board
4: **end for**
5: **RETURN**(P)

Program 18: Returns the pseudonyms of all messages sent in the current round.

The next function returns all messages of a certain type

MESSAGE_BOARD_GET_FUNCTIONS(type)

1: $M \leftarrow$ all messages in **TEMP_BOARD** such that the type is type
2: **for** private_board in **PRIVATE_TEMP_BOARDS** **do**

3: M.insert all messages in private_board such that the type is type
4: **end for**
5: **RETURN**(M)

Program 19: Returns all function messages

7.3 Clock

Our protocol requires a global clock. This ensures that players run in fairly large discrete rounds to prevent fine-grain timing attacks. We imagine this is accomplished by some broadcast signal and agreement on the duration of a single "round".

CLOCK_INITIALIZE()
1: **clock_time** $\leftarrow 0$

CLOCK_GET_TIME()
1: **RETURN**(**clock_time**)

CLOCK_INCREMENT()
1: **clock_time** $\leftarrow$ **clock_time** $+ 1$

Program 20: Basic functions of the clock object.

7.4 Random Oracle

We also require a random oracle.

RANDOM_ORACLE (x)
1: **if** $x \in$ **RO** **then**
2: **return** **RO**$[x]$
3: **else**
4: $r \leftarrow_{\$} \{0,1\}^{2\lambda}$
5: Add $\{x : r\}$ to **RO**
6: **return** r
7: **end if**

Program 21: A standard random oracle function.

8 Accumulator Formalism

Our asynchronous multi-party specification for an accumulator intends to replicate the functionality of the definitions in Section 2, but most functions we cannot define as step-by-step functions, since they may run in parallel, across several rounds and across several players. Rather, the functions are implicitly defined by the expected behaviour of the accumulator, and the messages the control program expects to see. We treat messages as objects here, having fields such as type, which can be implemented via any formatting.

Throughout this section, we say that players "should" perform some behaviour. This means that the control program looks for behaviour like this, and the honest players will lose one or more security games if they do not do the expected behaviour.

8.1 Player Functionality

Initialization. An accumulator protocol must implement

$$\textbf{SETUP\_PLAYER\_HONEST}(i),$$

which creates player i, and

$$\textbf{RUN\_PLAYER\_HONEST}(i),$$

which runs player i. When a player runs, they should process and respond to all posted messages in a round, but this may differ by protocol. In our formalism, players have access to the API of the message boards, the pseudonyms, and the random oracle. In practice these may be parameters or arguments to the setup.

Core Accumulator Functions. Any function that we expect server players to run, which manages the behaviour of the accumulator, we refer to as a *core accumulator function*. The control program never calls these directly; rather, any player (typically the adversary) can post a request for these functions to the message board, and other players will respond. The expected messages and the expected responses are:

- **GEN** $\rightarrow$ accumulator: Prompts the server players to initialize the accumulator. They should post a message of type ACC at some point after this request, which will contain this first value of the accumulator.
- **ADD**$(y, \text{proof}) \rightarrow$ accumulator: Prompts the server players to add the element y to the accumulator, so they should post a message of type ACC representing the new accumulator. The second component of proof is optional; the first time it is present indicates that the player who sent this Add function owns the element y. The server players should provide a long-term blind signature to the user based on this proof.
- **DEL**$(y) \rightarrow$ accumulator: Prompts the server players to delete the element y from the accumulator, and post a new accumulator via a ACC-type message.

User Functions. Similar to the core accumulator functions, there are also several functions that user players must implement in response to function messages, again mostly from the adversary.

- **WIT**$(y) \to \mathsf{sk}, \mathsf{witness}$, which takes as input a user ID y and prompts the protocol to generate a new user player who will have ID y, a long-term secret key sk, and a witness for their secret key and ID witness. This requires interaction with the server players, so the new player should send an Add message with their user ID, and a proof of knowledge of sk that will become part of their witness.
- **VER**(prover, epoch) starts an interactive protocol between a prover and a verifier (the prover is specified by a pseudonym), where the prover proves they have a valid witness or not for the accumulator of the provided epoch. At the end of the protocol the verifier player should call

  OBSERVER_CHECK_PROOF and provide:

  - an epoch of a valid accumulator
  - a boolean result (did the proving player have a valid witness for that accumulator?)
  - the pseudonym of the prover.
- **PROVE**(epoch) is the other half of the prove-and-verify protocol, where a user will provide a proof for this epoch.
- **UPDATE**(epoch) $\to$ witness updates a user's witness to epoch. In ALLOSAUR this is interactive, though it need not be (e.g., [VB20] uses public update data).

Messages of type ACC should have a value for the accumulator, and an *epoch* value, an increasing value to synchronize which accumulator is which. The epoch should increment with every Add and Del call.

Blames. Messages of type BLAME are meant to inform all users that some player is not trustworthy. These have a second field data, which can take one of three values: START, indicating that something has gone wrong and the servers will attempt to blame someone; END indicating that the servers have finished attempting to blame someone, and (user, i), where is the player ID of a player to be blamed.

8.2 Accumulator Functionality

Because of the complex nature of the accumulator and the messages it sends, we do not define the ideal functionality in the functions above, but specify it implicitly by "observer" functions. Our formalism will treat these as functions of the control program, but we stress that they do not *control* any user or adversary behaviour, but only observe it.

OBSERVER_INITIALIZE ()

1: **user_IDs** ← empty reflexive dictionary // Maps user IDs to accumulator elements y
2: **round_limit** ← ∞// time limits for verification
3: $\mathcal{S}$ ←empty dynamic array //the set of accumulated elements
4: $\mathcal{A}$ ←empty dynamic array // accumulators
5: **ver_time_limit** ← **ACCUMULATOR_VERIFY_TIME**(λ, N)// a time limit defined by the protocol
6: // Each array in **blames** is the ID of a server who blamed another server
7: **blames** ←array of N empty sets
8: **challenges** ←empty priority queue of $\mathbb{N}^4$, sorted by the last entry.// verification challenges
9: **anon_sets** ←empty dictionary// anonymity sets
10: **used_pseudonyms** ← empty set // pseudonyms already used for some function request

Program 22: Initialize all global variables for the accumulator.

ACCUMULATOR_VERIFY_TIME (λ, N)

1: **RETURN**(a time limit that the protocol specifies)

Program 23: Give a time limit for verifications.

Message Checks. The observer must maintain a notion of an "ideal accumulator", representing an accumulated set of which elements have been added or deleted, as well as the actual accumulator values. It does this by iterating through all new public messages at the end of every round, and updating accordingly.

First it checks for ACC messages, and it simply stores all such messages in an array $\mathcal{A}$, indexed by both the epoch and the player which posted it. It does not enforce any structure or consistency, and is agnostic to consensus.

Then it looks for FNC-type messages of Del or Add, each of which changes the ideal accumulated set in the expected way. The observer tracks this with a list of sets $\mathcal{S}$, and with every Del or Add the observer appends a new set to the list, equal to the last set plus or minus the element added or deleted.

Finally, it checks blame messages. The servers themselves manage consistency by posting public blame messages. A blame message simply gives the index of another server to be blamed, and the observer tracks these. If enough players blame a specific server, the observer triggers an abort. For this reason the abort function takes a single argument, representing a server. In any secu-

rity game, if the game aborts with the index of a corrupted server, the adversary loses.

```
OBSERVER_UPDATE_IDEAL(message)
// Users cannot modify the accumulator
if message.sender ∉ servers then
    RETURN
end if
if message is formatted as an ACC-type message then
    A ← message.data
    // We only consider the first accumulator value each server sends
    if 𝒜[message.epoch][message.sender] ≠ ⊥ then
        𝒜[message.epoch][message.sender] ← A
    end if
else if message is formatted as a FNC-type message then
    if message is formatted as a Del message with ID y then
        𝒮.append(𝒮.last \ {y})
        𝒜.append array of size |servers| initialized with ⊥
    else if message is formatted as an Add message with ID y then
        𝒮.append(𝒮.last ∪ {y})
        𝒜.append array of size |servers| initialized with ⊥
        // If an add request contains a proof, it signals a new user to track

        if message is the first message to add y with a proof attribute then
            user_IDs.set(y :
                PSEUDONYM_GET_IDENTITY(message.sender))
        end if
    end if
else if message is a BLAME-type message then
    blames[m.user].add(m.sender)
    if size(blames[m.user]) ≥ BLAME_THRESHOLD then
        ACCUMULATOR_ABORT(m.user)
    end if
end if
```

Program 24: Check if a message carries important accumulator information, and update the ideal accumulator as required.

Starting User Behaviour. When new user behaviour is requested (via messages), the observer logs what the user is requested to do, to ensure they correctly perform this.

```
OBSERVER_CHECK_FUNCTIONS(fncs)
// fncs is a list of function-type messages
for (reciever, function, data) in fncs do
    id ← PSEUDONYM_GET_IDENTITY(reciever)
    y ← user_IDS[id] // ⊥ if user has no ID yet
    // Check that the request is well-formed
    if function ∉ {Wit_u, Upd_u, Ver_u} then
        Skip this iteration of the loop
    end if
    if fncs contains more than one message to receiver or
            receiver ∈ old_pseudonyms then
        //Users ignore multiple function requests to the same pseudonym
        Skip this iteration of the loop
    end if
    // Disallow user functions before the accumulator is generated
    if |A| = 0 then
        RETURN
    end if
    if fun = Wit then
        // Assign the ID y to this player, if they have no ID
        if y = ⊥ then
            user_IDs.add(y : id)
        end if
    else if id = ⊥ then
        // For all other functions, the player must exist
        Skip this iteration of the loop
    end if
    // When a proof is requested, this records the arguments.
    if fun = Prove then
        OBSERVER_START_PROOF(data.verifier, data.epoch, id)
    else if fun = Upd then
        // Nothing to track with an update
    end if
end for
// Update anonymity sets
RESTRICT_ANONYMITY(fncs)
```

Program 25: Prompts users to perform specific functions. If it is Prove, this starts a proof challenge, which the observer program manages.

This allows the observer to track three key features of these messages: when users are added, when users should verify credentials, and anonymity.

Anonymity. When any function is queued for a user, the observer adds it to a list of functions to be executed, which it uses to manage anonymity sets. This is done after all messages are sent, but before they are re-ordered and committed. We do not define the anonymity sets in the general formalism, and leave it to each protocol to decide how anonymity sets are defined and managed.

This does not mean that a real implementation of the protocol must implement such a function. Instead, specifying a protocol and proving its security must involve specifying a theoretical observer function

RESTRICT_ANONYMITY, which gives a precise accounting of how much anonymity each user has in each round.

8.3 Verification

Sending a Prove message instructs a user to prove their identity, so the observer creates a "challenge" for each request, to check whether the user correctly verifies its credentials within some time period. A user requested to prove their credentials to a player pseudonym verifier should convince that verifier to call a special function **OBSERVER_CHECK_PROOF** with an argument indicating whether the verification succeeded.

OBSERVER_START_PROOF (verifier, epoch, prover)

1: **if** verifier has posted no messages on the message board, or prover $== \perp$ **then**
2: **RETURN**
3: **end if**
4: // Ensure this accumulator will eventually be posted
5: **if** $|\mathcal{S}| <$ epoch **then**
6: **RETURN**
7: **end if**
8: // Do not create a challenge if either party is corrupt
9: **if** **PSEUDONYM_CHECK_CORRUPTED**(verifier) or **PSEUDONYM_CHECK_CORRUPTED**(prover) **then**
10: **RETURN**
11: **end if**
12: // Servers do not have IDs, so ignore challenges to servers
13: **if** prover $\in$ **servers** **then**
14: **RETURN**
15: **end if**
16: $T \leftarrow$ maximum index of $\mathcal{A}$ with |**servers**| non-empty entries
17: // Compute the numbered of queued changes to the accumulator
18: changes $\leftarrow$ length($\mathcal{A}$) $- T$
19: time_limit $\leftarrow$ round_num + **ver_time_limit** + **time_per_change** $*$ changes
20: **challenges**.add(prover, verifier, epoch, time_limit)

```
21: round_limit ← min{time_limit, round_limit}
```

Program 26: When a proof request is posted, this keeps a record of the verifier, prover, and the start time, to later check that the verification was correctly performed within a specific amount of time.

The purpose of creating a challenge for each proof is to ensure progress. Each challenge creates a deadline for the verifier to post a valid response; if the response is not posted in time, the control program will exit with a failure.

When a user calls **OBSERVER_CHECK_PROOF**, the observer program compares the output to the previously-recorded challenges. If the challenge was answered correctly, it deletes the challenge, and otherwise it calls a failure function and ends the accumulator game.

This function ignores any calls from corrupt players. An adversary can choose to simply give nonsense answers at any time, and since only the observer program sees this function, there is no need to respond to the adversary.

```
OBSERVER_CHECK_PROOF (verifier, prover, result, epoch)
 1: // Adversary's can call this function, so we ensure the verifier is honest
 2: if not PSEUDONYM_VERIFY(verifier) then
 3:   RETURN
 4: end if
 5: // Ignore responses from corrupt players
 6: if ACTIVE_PLAYER_ID ∈ C then
 7:   RETURN
 8: end if
 9: // A proof for a non-existent accumulator is obviously incorrect
10: if |A[epoch]| < |server| then
11:   ACCUMULATOR_FAIL
12: end if
13: // If an honest player every validates an adversarial proof, record the
    fact
14: if PSEUDONYM_CHECK_CORRUPTED(prover) then
15:   if result then
16:     adv_verified ← TRUE
17:   end if
18:   RETURN
19: end if
20: // If the user passes but is not supposed to pass (resp. fail), the accu-
    mulator fails
21: y ← user_IDs[PSEUDONYM_GET_IDENTITY(prover)]
22: if (y ∈ S[N] and b == 0) or (y ∉ S[N] and b == 1) then
```

```
    ACCUMULATOR_FAIL
end if
// Reaching this point implies success
// Remove the corresponding challenge(s), so the accumulator does not time out
for (p, v, n, t) in challenges do
    if p = prover and v = verifier and n ≤ epoch then
        Remove (p, v, n, t) from challenges
    end if
end for
if challenges non-empty then
    // challenges is sorted so the first element has smallest time value
    round_limit ← challenges[0].time
else
    round_limit ← ∞
end if
```

Program 27: Checks that a previously-started verification passes as it should, based on the set of accumulated elements.

9 Control Flow

The basic structure of the control program is to initialize all users and allow the adversary to corrupt some subset of them. It then iterates through "rounds", and in each round, it sequentially and atomically runs each player (including adversarially corrupted players). The order in which the players run in each round is dictated by the adversary. Here we divide the players into two types: *servers* and *users*. The control program initializes server players at the beginning of the game, and they maintain the accumulator. The control program initializes user players at the adversary's direction, and user players maintain only their own credentials. This structure is a static corruption model.

After each round, the adversary can produce some output in an attempt to win the security game.

We use **purple text** to denote global variables of each user.

9.1 Main Control Flow

```
ACCUMULATOR_GAME (λ, N, k):
OBSERVER_INITIALIZE (λ, N)

// Give control of at most k allowable players to the adversary.
C ← ADVERSARY.corrupt(λ, N, k).
```

```
5: if |C| > k or C ⊈ [N] then
6:    ACCUMULATOR_ABORT(any index in C)
7: end if
8: adv_id ← any index in C
9: for i = 1 to N, i ∉ C do
10:   SETUP_PLAYER_HONEST(i)
11: end for
12:
13: // Run the main protocol.
14: loop
15:   //Run a round of the protocol.
16:   GAME_ROUND ().
17:   b ← GAME_NEXT_ROUND()
18:   // Stop if the adversary is done.
19:   if b == 0 then
20:      answer ← ADVERSARY.finalize()
21:      Output: ADVERSARY: answer
22:      Output: all global variables
23:      HALT
24:   end if
25: end loop
```

Program 28: Accumulator protocol (the starting point of execution). This initializes variables, players, and the adversary, who then corrupts some players. After this, it runs accumulator rounds until it either aborts, fails, or the adversary decides to stop execution and output some answer. The desired stopping conditions depend on the particular security game.

ACCUMULATOR_ABORT (player i):

```
1: if i ∈ C then
2:    Output: ADVERSARY FAILS
3: else
4:    Output: FAIL
5: end if
6: Output: all global variables
7: HALT
```

ACCUMULATOR_FAIL ():

```
1: Output: FAIL
2: Output: all global variables
3: HALT
```

Program 29: Two alternative ways to halt the accumulator. "Abort" represents a case where a player i has done something wrong; "Fail" represents a case where the accumulator has failed in its functionality.

INITIALIZE_GAME (λ, num_servers)

```
1:  // Initialize pseudonyms, with servers having public pseudonyms
2:  PSEUDONYM_SETUP(num_servers)
3:  // Initialize empty message boards.
4:  MESSAGE_BOARD_SETUP().
5:  PRIVATE_MESSAGE_BOARD_SETUP()
6:  CLOCK_INITIALIZE()
7:  ACTIVE_PLAYER_ID ← 0
8:  num_players ← num_servers
9:  servers ← [num_servers]
10: C ← ∅ // Set of corrupted players
11: message_index ← 0// Index of last public message board
12: OBSERVER_INITIALIZE()
```

Program 30: Initialize all subroutines and global variables for the control program to manage players.

GAME_ROUND ():

```
1:  P ← ∅ // Used to track which players already ran
2:  σ ← random permutation of [num_players] such that σ(p) = p for all
    p ≤ |servers|
3:  loop
4:    id ← ADVERSARY.choose_player(num_players)
5:    // Abort if incorrect player chosen.
6:    if id ∉ {1, ..., num_players} then
7:      ACCUMULATOR_ABORT(adv_id);
8:    end if
9:    id ← σ(id)
10:   P.insert(id)
11:   GAME_RUN_PLAYER (id)
12:   // Adversary decides whether to end the round
13:   if ADVERSARY.finish_round() then
14:     // End round only if all players have run
15:     if {1, ..., num_players} ⊆ P then
16:       RETURN
17:     end if
```

```
    end if
end loop
```

Program 31: One round of the accumulator. The adversary dynamically chooses players to run, potentially running some players many times. Once all players have run, the adversary is allowed to end the round.

GAME_NEXT_ROUND ():

```
// Find function messages to track anonymity
fncs ← MESSAGE_BOARD_GET_FUNCTIONS(FNC)
RESTRICT_ANONYMITY(fncs)
// Deliver messages
MESSAGE_BOARD_DELIVER()
PRIVATE_MESSAGE_BOARD_DELIVER()
// Track accumulator functionality
loop
    m ← MESSAGE_BOARD_PULL(message_index)
    if m = ⊥ then
        BREAK
    end if
    OBSERVER_UPDATE_IDEAL(m)
    message_index++
end loop
CLOCK_INCREMENT()
// Check for timeout
if CLOCK_GET_TIME() > round_limit then
    ACCUMULATOR_FAIL
end if
// Get adversarial response
b ← ADVERSARY.round()
RETURN(b).
```

Program 32: Finalize a round of the accumulator. Calls the observer to restrict anonymity and check the status of the accumulator, then increments the round..

GAME_RUN_PLAYER (*i*)

```
Track who sends messages
Set //ACTIVE_PLAYER_ID ← i
// Check if i is not in the corrupted set.
if i ∉ C then
    RUN_PLAYER_HONEST (i);
```

```
6: else
7:    ADVERSARY.run_player (i);
8: end if
9: Default player ID is adversarial
10: ACTIVE_PLAYER_ID ← adv_id
```

Program 33: Run a player, either honestly or as the adversary.

To prompt user behaviour, the adversary will post function messages. This means the control program need not concern itself with requesting user functions besides setting up a player and running them once per round.

De-anonymizing. Messages can only be sent to pseudonyms. Thus, for an adversary to request some user behaviour, they must have a pseudonym for that user. We would instead like to model a scenario where an adversary can prompt a *specific* user to perform some behaviour, and leave the user to select a new pseudonym if they need to. To do this, we give the adversary the ability to send a set of user IDs $y_1, \ldots, y_\ell$, and the control program will request a new pseudonym for each user, and return the pseudonyms to the adversary, albeit shuffled. This gives the adversary a fresh set of pseudonyms, where the anonymity set for each pseudonym is $\{y_1, \ldots, y_\ell\}$. From there, the adversary chooses to send function messages to the pseudonyms as it wishes.

```
GAME_DEANONYMIZE (𝒴)
1: P ← ∅
2: // All honest user IDs
3: 𝒴 ← 𝒴 \ {y : user_ids[y] ∈ 𝒞}
4: if 𝒴 contains any duplicate elements then
5:    RETURN
6: end if
7: for y ∈ 𝒴 do
8:    // Create a new honest player as needed
9:    if y ∉ user_IDs then
10:      new_index ← num_players
11:      SETUP_PLAYER_HONEST(new_index)
12:      user_IDs[y] ← new_index
13:      num_players + +
14:   end if
15:   // This assigns a new pseudonym for y, and adds it to P
16:   P.add(PSEUDONYM_GIVE(user_IDs[y]))
17: end for
18: // Restricts anonymity sets of these pseudonyms to the input users
19: for p ∈ P do
```

```
    if p ∉ anon_sets then
      anon_sets[p] ← 𝒴
    else
      anon_sets[p] ← anon_sets[p] ∩ 𝒴
    end if
  end for
  P.sort() // decorrelates P from 𝒴
  RETURN(P)
```

Program 34: Create new pseudonyms for a set of players and return them.

In our protocol definition, only this function can add new users. Since the adversary is the only player that can request this, this means the adversary controls which users are added, and also controls which ID they have by sending Wit messages. Once an adversary creates a user in this way, the user then behaves honestly and outside of the adversary's control.

For dishonest users, the adversary can simply request a new pseudonymous ID and simulate such a user "internally", i.e., the control flow does not need a separate program to act as a corrupted user.

9.2 Public Functions

Certain control program functions are available for the users to call; they are as follows:

- **PSEUDONYM_NEW**
- **PSEUDONYM_GET**
- **MESSAGE_BOARD_POST**
- **MESSAGE_BOARD_PULL**
- **PRIVATE_MESSAGE_BOARD_POST**
- **PRIVATE_MESSAGE_BOARD_PULL**
- **PRIVATE_MESSAGE_BOARD_REVEAL**
- **RANDOM_ORACLE**
- **CLOCK_GET_TIME**
- **OBSERVER_CHECK_PROOF**

9.3 Adversaries

The adversary can be any collection of PPT interactive Turing machines. They must implement the following functions, which the control program will call at various points during execution. The adversary has an internal persistent state that all of these functions share access to.

1. **ADVERSARY.corrupt**$(\lambda, N, k) \to C$: Returns a subset of indices between 1 and N, representing server players to be corrupted and under the adversary's control.
2. **ADVERSARY.choose_player**$(P) \to i$: This returns an index between 1 and P, representing the next player to run.
3. **ADVERSARY.finish_round**$() \to b$: Returns a boolean value, of whether the adversary has decided to finish the current round in order to start the next one.
4. **ADVERSARY.reorder**(**DYNAMIC_ARRAY**) $\to \pi$: Given an array of messages (or message headers), output a permutation π, from which they will be re-ordered.
5. **ADVERSARY.run_player**(**i**): Run a (corrupted) player i.
6. **ADVERSARY.round**$() \to b$: Finish any calculations at the end of a round, and output a boolean value $b \in \{0, 1\}$ indicating whether to stop the game.
7. **ADVERSARY.finalize**$() \to A$: Once the adversary calls for the game to stop, it gets this chance to finalize its calculations and produce an answer A (whose form depends on the security game).

During the execution of any of the above functions, the adversary can call any of the public functions, as well as **GAME_DEANONYMIZE**.

10 Security

Unlike the single server case, where we presented the functions an accumulator must execute and then defined security in terms of these functions, the formalism we have so far described, which has no rigorous definition of the behaviour of any accumulator function, is sufficient to define security. That is, a specific protocol need not implement a specific Gen or Wit function; rather, the desired functionality of maintaining an accumulator and its credentials is part of the observer program's internal state, and it is up to the protocol to ensure it meets the required functionality.

As an example, an empty protocol (where all honest players do nothing) would fit into our formalism (imagine that the formalism "compiles"), but it

would fail correctness. The adversary would request additions to the accumulator which would never be fulfilled, the user would never verify their credentials, and the observer program would reach its internal time limit.

By the structure of our formalism, all security games involve the same interactions of the adversary and the control program. The only difference between them is the adversary's goals. For this reason we do not need to write a formal security "game"; in all cases, we consider

$$\mathsf{answer}, \mathsf{state} \leftarrow \textbf{ACCUMULATOR\_GAME}(\lambda, N, k)$$

and define security in terms of the probability that answer (the result of execution; possibly an answer from the adversary) and state (the internal state of the observer program) satisfy certain properties. We access variables in state with the same syntax as the observer program's global variables in the accumulator functionality.

Informally, our three goals are to ensure that honest users are able to verify their credentials, dishonest users cannot verify credentials they do not have, and messages from different pseudonyms are not correlatable.

10.1 Correctness

The goal of the correctness game is to ensure that the accumulator progresses as expected and all honest users are able to produce the correct proofs of their credentials.

Definition 10.1. An accumulator is *correct* if, for all PPT adversaries $\mathcal{A}$,

$$\Pr\left[\mathsf{answer} = \text{FAIL} \middle| \begin{matrix}\mathsf{answer},\\ \mathsf{state}\end{matrix} \leftarrow \textbf{ACCUMULATOR\_GAME}(\lambda, N, k)\right] \leq \text{negl}(\lambda)$$

where **ACCUMULATOR_GAME** interacts with $\mathcal{A}$.

Inspecting the pseudocde, there are conditions that can cause answer =FAIL:

1. When a verifier calls **OBSERVER_CHECK_PROOF**, the control program finds the ID y associated to the prover's pseudonym and checks whether y is supposed to be in the accumulator for the provided epoch. This tells it what the result *should* be; if the result is different, the control program calls **ACCUMULATOR_FAIL**. If this happens, the adversary wins. The specifics of this are detailed in the formalism of

   OBSERVER_START_PROOF and **OBSERVER_CHECK_PROOF**.

2. Every time any user requests a proof from an honest user, indicating that a user should verify their identity, the control program sets a round limit. If the corresponding verification is not called before that limit, the accumulator also fails.

3. If a majority of server players post blame messages towards an honest player.

10.2 Commitment Soundness

Here the goal is that an adversary cannot produce fake credentials. Because credentials are blinded, this complicates the definition in two ways:

1. An adversary may find a way to pass the verification protocol without any credentials at all;

2. An adversary may find a way to create credentials it should not have.

To address the first issue, we do not require a soundness adversary to ouput a witness, but rather the observer program tracks (Line 16 of **OBSERVER_CHECK_PROOF**) whether an adversarial player has ever successfully passed a verification protocol. To ensure that the adversary needs a witness to do this, we define an *extractor* function which can call the adversarial functions (including modifying internal state) a polynomial number of times and access the transcript of the accumulator game. The extractor attempts to outputs a tuple (y, n, w) consisting of an ID y, an epoch n, and a witness w.

To decide whether the extracted witness is valid or not, we need two functions: Acc and Ver.

Acc takes as input an array of accumulator values and outputs either $\perp$ (for no consensus) or another accumulator. This represents some sort of consensus; for example, it could take the majority value of the array (if it exists). In ALLOSAUR this function outputs $\perp$ unless there are $|\mathbf{servers}|$ values in the array which are all identical, in which case it outputs that value.

The second function is $\mathsf{Ver}(A, y, w) \to \{0, 1\}$ which takes as input an accumulator A, a user ID y, and a witness w. Intuitively, this should ouput 1 if and only if w is a "valid" witness.

Thus, if an adversary ever passes the verification protocol, there should be an extractor that can successfully extract a valid witness.

For the second goal of our definition, to ensure an adversary only possesses credentials they "should" have, we require that for *any* extractor, the extracted witness belongs to the adversary and matches the accumulated set. The observer program tracks which player owns which ID y via **user_IDs**, so once an extractor outputs (y, n, w), the ID y must belong to a corrupted player. Second, the value y should belong to the accumulated set $\mathcal{S}$ at epoch n.

Formally,

Definition 10.2. An accumulator is *blind commitment sound* if, for all PPT adversaries $\mathcal{A}$ such that

$$\Pr\left[\mathbf{adv\_verified} = \mathsf{TRUE} \,\middle|\, \begin{matrix}\mathsf{answer}, \\ \mathsf{state}\end{matrix} \leftarrow \mathbf{ACCUMULATOR\_GAME}(\lambda, N, k)\right] \geq \mathrm{negl}(\lambda)$$

then there exists an extractor $\mathcal{E}$ such that, given access to $\mathcal{A}$, its internal state, and the public message board **BOARD** (as part of the output state),

$$\Pr\left[\mathsf{Ver}(A, y, w) = 1 \,\middle|\, \begin{matrix}(y, n, w) \leftarrow \mathcal{E}(\mathcal{A}, \mathbf{BOARD}) \\ A \leftarrow \mathsf{Acc}(\mathcal{A}[n])\end{matrix}\right] \geq \mathrm{negl}(\lambda)$$

Further, for all such extractors, it must also hold that

$$\Pr\left[\begin{array}{c} \mathsf{Ver}(\mathcal{A}[n], y, w) = 1 \\ \text{and} \begin{cases} \mathbf{user\_IDs}[y] \notin \mathcal{C}, \text{ or} \\ \forall m \geq n : \mathsf{Acc}(\mathcal{A}[m]) = \bot \text{ or } y \notin \mathcal{S}[m] \end{cases} \end{array} \middle| (y, n, w) \leftarrow \mathcal{E}(\mathcal{A}, \mathbf{BOARD}) \right] \leq \mathrm{negl}(\lambda)$$

10.3 User indistinguishability

To clarify our notation, a user has 3 "identities" in the accumulator:

- An internal ID i that only the control program knows or uses.
- A user ID y that is given to the user on creation and remains constant throughout execution.
- Multiple pseudonyms p, which are used to address messages to and from the user.

In practical terms, the user ID y might already be pseudonymous (in ALLOSAUR it is a random element of a finite field); however, the privacy notion we want to capture is that different interactions with the accumulator cannot be correlated. At certain points, a user's *real* identity is correlated with certain interactions (for example, when they verify their credentials), so we must ensure that these interactions are not correlated with each other.

To that end, our goal is to prevent correlation between distinct pseudonyms. This cannot be perfectly achieved, since there is implicit data available from just the metadata of different messages: for example, if a pseudonym posts a message which passes a verification, then we know the ID associated to that pseudonym must be in the accumulated set.

Thus, the observer program tracks *anonymity sets*. The anonymity sets are subsets of all user IDs y, and they are indexed by a pseudonym. An anonymity set represents all possible user IDs which might belong to a specified pseudonym.

Definition 10.3. An accumulator is *indistinguishable* if, for all PPT adversaries $\mathcal{A}$ and all integers n_{12}, n_1, n_2:

$$\Pr\left[\begin{array}{c} \mathbf{identities}[p_1] \\ = \\ \mathbf{identities}[p_2] \end{array} \middle| \begin{array}{c} \mathsf{answer}, \\ \mathsf{state} \end{array} \leftarrow \mathbf{ACCUMULATOR\_GAME}(\lambda, N, k) \\ \mathsf{answer} = \text{ADVERSARY:}(p_1, p_1) \\ |\mathbf{anon\_sets}[p_1] \cap \mathbf{anon\_sets}[p_2]| = n_{12} \\ |\mathbf{anon\_sets}[p_1]| = n_1 \\ |\mathbf{anon\_sets}[p_2]| = n_2 \right] \leq \frac{n_{12}}{n_1 n_2} + \mathrm{negl}(\lambda)$$

The target probability, which the adversary must exceed, is the probability that if we draw two random IDs from the anonymity set of the pseudonyms p_1 and p_2 that the adversary provides, that these IDs will match. However,

this definition means that the target probability depends on the specific state of a single execution. This creates a slight paradox: we must define security statistically, in terms of all possible executions, but the threshold for security depends on the execution-specific anonymity sets. We solve this by conditioning the size of the anonymity sets.

To justify this definition, users must disguise their messages using some internal randomness to have any hope of anonymity. This means that many different executions (either based on randomness of the users, or the control program's randomness in the order of execution of user players) will create indistinguishable messages. We thus expect, in an indistinguishable accumulator, many executions where the adversary outputs the same pair of pseudonyms, even though different users are given those pseudonyms.

Our definition captures the notion that the specifics of the protocol should not reveal any more information than they need to. However, since the construction of anonymity sets is left to the protocol, one could define a protocol which is "indistinguishable" by having completely non-anonymous messages. While this could be a problem, it instead forces the definition of a protocol which claims any anonymity to *precisely* quantify the anonymity lost from the metadata of each message.

Notice also how the definition forces an indistinguishability adversary to grant some anonymity to users in the game. If all pseudonyms are totally de-anonymized and contain only a single user ID, then the target probability is either 0 (the pseudonyms belong to different users) or 1 (the pseudonyms belong to the same user), and the adversary can never exceed probability 1.

11 Common Functionality

The functionality we described so far is enough to define the relevant security games, and the rest could be left up to each protocol. However, there is a basic flow of user execution we describe here that could form a robust basis for any protocol.

11.1 User control flow

Our basic control for a user is to first iterate through all received messages and add any function calls to one of two functions to execute: a queue of *core accumulator functions*, which are public functions that should be run in-order, and a queue of *user functions*, which are meant to be run simultaneously.

At many points during execution, these functions may need to wait for other players to post messages. Thus, we need some functionality to pause and resume execution. The formalism here does this by defining a data type for a function stack: it should contain all local variables and a record of the point of execution of every function in the stack. Users call it via GetCallStack(line number). During execution, if a player must wait for the next round, the player will return this stack to **RUN_PLAYER_HONEST**, which stores it for the next round.

A real-world implementation could use any mechanism to wait for messages to be posted.

For functions, they are run in a given order for anonymity reasons. The ordering of user functions is as follows:

$$\mathsf{Ver}_s > \mathsf{Upd}_s > \mathsf{Wit}_u > \mathsf{Upd}_u > \mathsf{Ver}_u$$

where multiple calls to Upd_u or Ver_u are sorted so that (a) calls that *started* in earlier rounds are first, and (b) among those that started in the same round, those with the greatest accumulator number are first.

We use **purple text** to represent variables which are local to each player, but persistent between rounds. Implicitly, these are actually arrays of data that are indexed by the player's ID. We will be a bit sloppy and allow users access to the control program variable **servers**; this is public data since the number of servers will be a global parameter of the scheme and the server players' IDs are numbered sequentially.

```
SETUP_PLAYER_HONEST (id)
// Recall that users have access to a pseudonym
// manager, public and private message boards,
// and a random oracle. We do not provide these
// as input here, but each player is given access
// to these objects.
if id ∈ servers then
   role ← server
else
   role ← user
end if
core_function_queue ← ∅// core accumulator functions
user_function_queue ←empty priority queue// user functions
data_set ← ∅// all public messages
private_data_set ← ∅// all private messages
Function ← ⊥// current core accumulator functions
public_index ← 0// index into public message board
private_index ← empty dictionary// indices into private message boards
acc_epoch ← −1// current epoch
accs ← [] // all posted accumulators
wait_counter ← 0 // tracks timeout of WAIT
wait_number ← 0 // synchronizes calls to WAIT
open_number ← 0 // synchronizes calls to OPEN
random_share_count ← 0 // used for secret sharing
triple_share_count ← 0
curr_round ← CLOCK_GET_TIME() // tracks last round
pseudonyms ← ∅ // tracks all used pseudonyms
```

Program 35: Initialize local player variables

User Functions. Users are prompted to start functions by messages posted to the public or private message boards. We want core accumulator functions to run sequentially; other functions are run simultaneously.

```
RUN_PLAYER_HONEST (i)
1:  // Ensures a user only runs once per round
2:  if curr_round ≥ CLOCK_GET_TIME() then
3:    RETURN
4:  end if
5:  HONEST_PLAYER_PROCESS_MESSAGES(i)
6:  loop
7:    // If no function is currently being executed
8:    if Function == ⊥ then
9:      if core_function_queue not empty then
10:       m ← core_function_queue.pop()
11:       fun ← m.fun_type
12:       fun_data ← m.data
13:       Function ← (fun, fun_data, line = 0)
14:     end if
15:   end if
16:   if Function ≠ ⊥ then
17:     // Run another round of Function
18:     // This is defined by the actual implementation
19:     Function ← RUN(Function)
20:   end if
21:   // If the previous function hasn't finished, we must wait for the next
      round
22:   if Function ≠ ⊥ then
23:     BREAK_LOOP
24:   end if
25: end loop
26: // Run all user functions
27: current_user_funs ← user_function_queue
28: user_function_queue ← empty queue
29: for User_Function in current_user_funs do
30:   New_Function ← RUN(User_Function)
31:   if New_Function ≠ ⊥ then
32:     user_function_queue.push(New_Function)
33:   end if
```

```
34: end for
```

Program 36: A single round for an honest player. This retrieves all new messages from the last round, then continues execution of the accumulator functions requested, before posting any new messages this player creates.

When users process messages, they do a quick check for formatting. This ensures that all functions contain all the necessary data, e.g., that an Add function contains a user ID, etc.

```
HONEST_PLAYER_PROCESS_MESSAGES (i)
 1: // Process messages since last round
 2: loop
 3:   message MESSAGE_BOARD_PULL(public_index)
 4:   if message == null then
 5:     Exit loop
 6:   else if message is not correctly formatted then
 7:     Skip to next loop iteration
 8:   end if
 9:   public_index++
10:   if message.type is FNC then
11:     if message.fun_type ∈ {Gen, Add, Del, Wit_s} and role = server then
12:       core_function_queue.add(message)
13:     else
14:       fun ← message.fun_type
15:       fun_data ← message.data
16:       user_function_queue.add(fun, fun_data, line = 0)
17:     end if
18:   else if message.type is ACC then
19:     if |servers| ACC messages are posted with the same counter and
        same value of accumulator then
20:       accs[message.counter].append(message.accumulator)
21:     end if
22:   else if message.type is DAT then
23:     data_set.add(message)
24:   end if
25: end loop
26: // Users have one private message board for all their pseudonyms
27: for p in PSEUDONYM_GET() do
28:   new_functions ← empty list
29:   loop
30:     message ← PRIVATE_MESSAGE_BOARD_PULL(
              private_index[p], p)
31:     if message == null then
```

```
32:         Exit loop
33:       end if
34:       private_index[p] + +
35:       if message is not correctly formatted then
36:         Skip to next loop iteration
37:       end if
38:       if message.type is FNC then
39:         // Ignore function requests to used pseudonyms
40:         if p ∉ pseudonyms then
41:           new_functions.add(message)
42:         end if
43:       else
44:         private_data_set.add(message)
45:       end if
46:     end loop
47:     // Ignore multiple messages to one pseudonym
48:     if new_functions contains only one message then
49:       user_function_queue.add(new_functions)
50:     end if
51:     pseudonyms.add(p)
52:   end for
```

Program 37: Processing messages for an honest player. This retrieves all new messages from the last round, queueing up any functions necessary to run.

11.2 Wait

When a player runs a particular function, it likely requires communication with other players. A common functionality is a **WAIT** function, which will pause execution of a particular function until a specified message is received.

A function calls **WAIT** with a sender and a condition as argument. **WAIT** checks if the required message is in the players **data_set** list, and if not, it returns execution to **RUN_PLAYER_HONEST**, with a call stack of the current point in execution in order to resume in the next round (when hopefully the desired message is posted). If the message is present, **WAIT** returns the value.

In this way, in every round where the data does not appear, the function will resume at this particular call to **WAIT**, and if the data is still not there, it immediately returns and continues waiting. If the data is there, it will continue execution.

```
WAIT (sender, condition, pseudonym)

1: wait_counter ← 0
```

```
2: // Checks the local data set for the message
3: for m in data_set (iterated in order received) do
4:    //Takes the first matching message, ensuring consensus
5:    if condition(m) and m.sender = sender then
6:       RETURN(m) to calling function
7:    end if
8: end for
9: // Track number of rounds, to blame players that do not post the
   required messages
10: wait_counter + +
11: if wait_counter == WAIT_THRESHOLD then
12:    MESSAGE_BOARD_POST(message = (type = BLAME, user =
       sender), sender = pseudonym)
13: end if
14: Function ← GetCallStack(3)
15: RETURN(Function) to RUN_PLAYER_HONEST
```

Program 38: Halts execution during another function until some data is received (checked via "condition") from a particular sender. The argument Function is the function that called **WAIT**, and counter is the counter of where this **WAIT** appeared in the function's logic.

PRIVATE_WAIT (sender, condition, pseudonym)

```
1: local_wait_counter ← 0
2: // Checks the local data set
3: for m in private_data_set (iterated in order received) do
4:    Takes the first matching message, ensuring consensus
5:    if condition(m) and m.sender = sender then
6:       RETURN(m) to calling function
7:    end if
8: end for
9: local_wait_counter + +
10: if local_wait_counter == WAIT_THRESHOLD then
11:    MESSAGE_BOARD_POST(message = (type = BLAME, user =
       sender), sender = pseudonym)
12: end if
13: Function ← GetCallStack(3)
14: RETURN(Function) to RUN_PLAYER_HONEST
```

Program 39: The same functionality as **WAIT**, except this waits for a message on a private message board.

12 Secret Sharing

Our protocol requires access to shared secrets for the secure multi-party computation (MPC). While we could adopt a fully-formed scheme for this (e.g., SPDZ), we instead use smaller pieces, as great portions of MPC protocols govern only integrity, which the accumulator can provide for free. For our MPC, we will use the classic Beaver triple technique [Bea92a]. We alsowant security against malicious adversaries who may try to stop the protocol, so we will enforce the *identifiable abort* property [BOS16].

In this section we define both the control program and ideal functionality (in the style of [BOS16]), as well as useful user functions to deal with these shared secrets.

The accumulator control program will call **MPC_INITIALIZE** during **OBSERVER_INITIALIZE**. This creates a new player and saves the ID and pseudonym of this player, as well as posting a single initialization message to the message board. The trusted third party will use this pseudonym to post secret shares and their commitments to other players.

MPC_INITIALIZE $(\cdot)$

1: **random_share_count** $\leftarrow 0$
2: **triple_share_count** $\leftarrow 0$
3: // Create player ID who will be a random secret trusted third party
4: **secret_ttp_id** $\leftarrow$ **num_players** $+ 1$
5: **SETUP_PLAYER_HONEST**(**secret_ttp_id**)
6: **secret_ttp_pseudonym** $\leftarrow$ **PSEUDONYM_NEW**()
7: **MESSAGE_BOARD_POST**(message $=$ ("secret_shares"), sender $=$ **secret_ttp_pseudonym**)

Program 40: Sets up the necessary infrastructure for secret sharing.

Here the control program manages shared secrets, privately sending the shares to each server while posting commitments to the public message board. We use our random oracle commitment scheme, but any commitment scheme will work. In certain cases the Beaver triple generation may produce commitments as a byproduct. We require the adversary to implement a function **provide_shares**, through which the adversary will provide its shares.

BRACKET (x, id)

1: Initialize a vector $\mathbf{x} \in \mathbb{F}_q^{\text{num\_servers}}$.
2: // Obtain arbitrary inputs for the adversarial shares.
3: $\mathbf{y} \leftarrow$ **ADVERSARY.provide_shares**().
4: // Incorporate the adversarial shares

```
for i ∈ servers do
    if i ∈ C then
        Set x_i = y_i.
    end if
    // Set the remaining shares randomly.
    if i ∉ C then
        Set x_i ← F_q.
    end if
end for
// Fix one of the non-adversarial shares so the sum is x.
Choose some j such that j ∉ C.
x_j ← x − Σ_{i∈servers−j} x_i.
// Generate "commitments" to each x_i.
previous_active_player ← ACTIVE_PLAYER_ID
ACTIVE_PLAYER_ID ← secret_ttp_id
Σ ← empty array
for i ∈ servers do
    (σ_{x_i}, o_i) ← COMMIT(x_i).
    message ← (type = DAT, data = (type + "triple_share", {x_i, o_i, id})
    PRIVATE_MESSAGE_BOARD_POST(
        message, receiver = i, sender = secret_ttp_pseudonym)
    Σ[i] ← σ_{x_i}
end for
message ← (type = DAT, data = ("triple_share_commit", id, Σ))
MESSAGE_BOARD_POST(message, sender = secret_ttp_pseudonym)
ACTIVE_PLAYER_ID ← previous_active_player
```

Program 41: Linearly shares a secret in an n out of n way.

Given this functionality for creating and posting secrets, we define a function that all players can call at any time, which creates a Beaver triple.

INPUT_TRIPLE $(\cdot)$

```
Sample a, b ← F_q uniformly at random.
Set c = a · b.
BRACKET(a, triple_share_count)
BRACKET(b, triple_share_count + 1)
BRACKET(c, triple_share_count + 2)
triple_share_count ← triple_share_count + 3
```

Program 42: Creates and posts secret shares for a Beaver triple.

User functions. We assume all players can call **GET_SHARED_RANDOM** and **INPUT_TRIPLE** at will. For convenience, we define the following functions for players, which track the shared secrets they have already received and request more as needed.

We define a global player variable **secret_ttp**, which they will set to be the sender pseudonym of the first message of "secret shares" which is posted to the accumulator.

We use the following commitment scheme, whose main purpose is for random oracle proofs to be easier.

COMMIT (x)

1: $r \leftarrow_\$ \{0,1\}^{2\lambda}$
2: $c \leftarrow$ **RANDOM_ORACLE**(x, r)
3: **RETURN**$(c, o = (x, r))$

Program 43: Hash-based commitment.

OPEN_COMMIT (c, o)

1: $(x, r) \leftarrow o$
2: $c' \leftarrow$ **RANDOM_ORACLE**(x, r)
3: **if** $c' = c$ **then**
4: **RETURN**(x)
5: **else**
6: **RETURN**$(\perp)$
7: **end if**

Program 44: Hash-based commitment opening.

GET_SHARED_RANDOM (pseudonym)

1: $r \leftarrow_\$ \mathbb{F}_p$
2: $(c, o) \leftarrow$ **COMMIT**(r)
3: **MESSAGE_BOARD_POST**(type = DAT, data = ("random", **random_share_counter**, c), sender = pseudonym)
4: commits $\leftarrow$ empty array of size **num_servers**
5: **for** $j \in$ **servers** **do**
6: commits$[j] \leftarrow$ **WAIT**$(j,$ m such that m.data = ("random", **random_share_counter**, c_j) for some c_j)
7: **end for**

8: $R \leftarrow (\mathsf{share} = r, \mathsf{open} = o, \mathsf{id} = \textbf{random\_share\_counter},$
$\mathsf{commit} = \mathsf{commits})$
9: $\textbf{random\_share\_counter} + +$
10: $\textbf{RETURN}(R)$

Program 45: Samples a random value and commits to it on the public message board.

GET_SHARED_TRIPLE ()

1: $T \leftarrow$ empty array of size 3
2: // If no unused Beaver triples are left, request a new one
3: **if** data_set has no message from secret_ttp with data ("triple_share_commit", triple_share_counter, Σ) for some Σ **then**
4: **INPUT_TRIPLE**()
5: **end if**
6: **for** $i \in \{0, 1, 2\}$ **do**
7: $m_1 \leftarrow \textbf{WAIT}(\textbf{secret\_ttp},$
("triple_share_commit", triple_share_counter $+ i, \Sigma$))
for some Σ
8: $m_2 \leftarrow \textbf{PRIVATE\_WAIT}(\textbf{secret\_ttp},$
("triple_share", x, σ_x, triple_share_counter $+ i$))
for some x and σ_x
9: $\textbf{wait\_number} \leftarrow \textbf{wait\_number} + 1$
10: // Shared secret data types have a share, ID, and commitments to other player's shares
11: $T[i] \leftarrow (\mathsf{share} = m_2.x, \mathsf{id} = \textbf{triple\_share\_counter}, \mathsf{commit} = m_1.\Sigma)$
12: $\textbf{triple\_share\_counter} \leftarrow \textbf{triple\_share\_counter} + 3$
13: **end for**
14: $\textbf{RETURN}(T[0], T[1], T[2])$

Program 46: Retrieves the next secret-shared Beaver triple from the message board. If none is present, requests the control program to post another.

12.1 Identifiable Abort

To accomplish identifiable aborts, ALLOSAUR will have a destructive blame where all players will reveal their secrets (hence rendering the accumulator insecure) in order to assign blame to a specific player. For this they need a function to check whether a shared secret was used correctly, given as follows.

CHECK_SHARED_VALUE $(y, \text{pseudonym})$

```
1:  // This player posts an opening of their secret share
2:  m ← (type = DAT, data = ("open_share", y.id, y.open))
3:  MESSAGE_BOARD_POST(m, sender = pseudonym)
4:  os, ys ←empty array of size |servers|
5:  for j ∈ servers do
6:      // Get an opening for each player's share
7:      opening ← WAIT(j, (type = DAT, data = (open_share, id, o)))
              for some o
8:      os[j] ← opening.o
9:      // Open the share
10:     ys[j] ← OPEN_COMMIT(y.commit[j], os[j])
11:     // If the opening is invalid, blame the player
12:     if ys[j] == ⊥ then
13:         MESSAGE_BOARD_POST(type = BLAME, user = j, sender =
            pseudonym)
14:     end if
15:     wait_number ← wait_number + 1
16: end for
17: RETURN(ys)
```

Program 47: Reveals a shared secret, then compares the values posted by other players to the commitments obtained from the public message board when that secret was first established. If anything doesn't match, it blames that player. Then it returns all shares.

12.2 Open

Opening shared secrets takes some care to prevent adversarial players from controlling the final value. We use a naive approach and simply post commitments to the share, then output the share once all other players have posted a commitment.

Our shared secret functionality gives an ID to every shared secret. **OPEN** will open a particular secret and and returns both the reconstructed value and all the shares. Most calls will ignore the shares.

OPEN $(x, \text{pseudonym})$

```
1:  // Post a commitment to this player's share
2:  c, open ← COMMIT(x, open_number)
3:  m ← (type = DAT, data = ("open_commit", open_number, c))
4:  MESSAGE_BOARD_POST(m.sender = pseudonym)
```

```
commits ← empty array of size N
// Wait for other player's commitments
for j ∈ servers do
    commit ← WAIT(j, m.data = ("open_commit", open_number, c_j))
          for some c_j
    commits[j] ← commit.c_j
    wait_number ← wait_number + 1
end for
// Post the opening for the share
m ← (type = DAT, data = ("open_reveal", open_number, open_i))
MESSAGE_BOARD_POST(m, sender = pseudonym)
xs ← empty array of size N
// Wait for all shares
for j ∈ servers do
    open ← WAIT(j, m.data = ("open_reveal", open_number, open_j)
    xs[j] ← OPEN_COMMIT(commits[j], open)
    // Blame any invalid openings
    if xs[j] == ⊥ then
        MESSAGE_BOARD_POST(type = BLAME, user = j, sender =
        pseudonym)
    end if
    wait_number ← wait_number + 1
end for
open_number ← open_number + 1
x ← the value x constructed from the shares {x_1, ..., x_|servers|}
RETURN(x, (x_1, ..., x_|servers|))
```

Program 48: Function for one user to open a specific shared secret.

In certain cases we will need to back-track and ensure that an opening matches a value revealed later, in order to perform blames. The following function covers this case:

```
CHECK_OPEN (X, id, pseudonym)

// X is an array of secret values
check ← True
// Find the original opening
for j ∈ servers do
    Find the first m_1, m_2 in data_set such that for some c_j, o_j:
        sender == j for both m_1 and m_2
        m_1.data = ("open_commit", id, c_j)
        m_2.data = ("open_reveal", id, o_j)
    x_j ← OPEN_COMMIT(c_j, o_j)
```

```
    if x_j ≠ X[j] then
      check ← False
      MESSAGE_BOARD_POST(type = BLAME, user = j, sender = pseudonym)
    end if
  end for
  // Abandon all execution if the check failed
  if not check then
    RETURN ⊥ to RUN_PLAYER_HONEST
  end if
  RETURN
```

Program 49: Function to check that a set of opened values provided as input (X) match what was previously posted during an Open indexed by id.

12.3 Invert

Our only use for Beaver triples will be inverting shared secrets in a finite field, which we with the following standard technique:

INVERT $(x, r, (a, b, c), \text{pseudonym})$

```
1:  // x, r, a, b, c are all secret shares; (a, b, c) is a Beaver triple
2:  ε_share ← x − a
3:  δ_share ← r − b
4:  ε ← OPEN(ε_share, pseudonym)
5:  δ ← OPEN(δ_share, pseudonym)
6:  z_share ← c + ε · r + δ · x − ε · δ
7:  z ← OPEN(z_share, pseudonym)
8:  if z == 0 then
9:    y ← 0
10: else
11:   y ← z^{-1} r  mod q
12: end if
13: RETURN([y])
```

Program 50: Inverts a shared secret x.

During a destructure blame, we will need to check that an inverse was performed correctly, and that is done as follows.

CHECK_INVERSE (open_start, shares_x, r, a, b, c, pseudonym)

1: $\mathsf{shares}_r \leftarrow$ **CHECK_SHARED_VALUE**(r, pseudonym)
2: $\mathsf{shares}_a \leftarrow$ **CHECK_SHARED_VALUE**(a, pseudonym)
3: $\mathsf{shares}_b \leftarrow$ **CHECK_SHARED_VALUE**(b, pseudonym)
4: $\mathsf{shares}_c \leftarrow$ **CHECK_SHARED_VALUE**(c, pseudonym)
5: $\mathsf{shares}_\epsilon[j] \leftarrow \mathsf{shares}_x[j] - \mathsf{shares}_a[j]$ for all j
6: $\mathsf{shares}_\delta[j] \leftarrow \mathsf{shares}_r[j] - \mathsf{shares}_b[j]$ for all j
7: **CHECK_OPEN**(shares_ϵ, open_start, pseudonym)
8: **CHECK_OPEN**(shares_δ, open_start + 1, pseudonym)
9: $\epsilon \leftarrow$ reconstruct from shares_ϵ
10: $\delta \leftarrow$ reconstruct from shares_δ
11: $\mathsf{shares}_z[j] \leftarrow \mathsf{shares}_c[j] + \epsilon \cdot \mathsf{shares}_r[j] + \delta \cdot \mathsf{shares}_x[j] + \epsilon \cdot \delta$ for all j
12: **CHECK_OPEN**(shares_z, open_start + 2, pseudonym)
13: $z \leftarrow$ reconstruct from shares_z
14: **if** $z == 0 \mod q$ **then**
15: $\quad \mathsf{shares}_y[j] \leftarrow 0$ for all j
16: **else**
17: $\quad \mathsf{shares}_y[j] \leftarrow z^{-1}\mathsf{shares}_r[j] \mod q$ for all j
18: **end if**
19: **RETURN**(shares_y)

Program 51: Checks that an inverse was performed correctly, assuming opens for that inverse start at open_start, using the randomness r and Beaver triple (a, b, c) (where each of these inputs is a tuple of $(\mathsf{share}, \mathsf{id}, \Sigma)$).

13 ALLOSAUR

At its core, our multi-server construction is identical to the single-server construction, except that the secret values α and s_m are shared among all the servers. This means that all computations using these values require multi-party computation, as well as mechanisms to achieve consensus on the updated value of the accumulator.

13.1 Differences with previous accumulators

No non-membership witnesses. Because of security issues, we assume that applications needing non-membership witnesses will use a second accumulator tracking elements *not* in the first accumulator.

The accumulator does not change with additions. When an element y is added to the accumulator, the value of the accumulator (say, V) does not change. However, the servers compute $\frac{1}{y+\alpha}V$ anyway, since this will be the witness that the user can obtain from the public messages.

Servers maintain a witness for everyone. When y is added to the accumulator, the servers store the value $\frac{1}{y+\alpha}V$. They update this value every time the accumulator changes (i.e., with deletions). When an element y is deleted, the servers do not do any MPC, they simply set the accumulator value to equal the witness they have stored for y.

This may be sub-optimal in practice. Since updates can be batched, or MPC used, we leave this as a later design decision.

Servers compute user updates. When a user y requests an update, the servers compute the batch polynomial of [VB20]. To do this anonymously, the user divides all powers of y, as $1, y, y^2, y^3, \ldots$ with Shamir secret sharing. The servers apply the polynomial to their shares of the powers of y, which the user reassembles.

Trapdoor secret shares are linear; user secret shares are randomized Shamir shares. In our implementation, the trapdoors of the accumulator (e.g., α) are shared so that if server i has a share α_i, then $\alpha = \alpha_1 + \cdots + \alpha_N$. In contrast, the user uses a t-out-of-n Shamir secret sharing scheme to split their ID y during updates.

13.2 Parameters

ALLOSAUR needs the same parameters as the single-server case: two elliptic curve groups G_1 and G_2 of prime order q, with a Type 2 pairing $e : G_1 \times G_2 \to G_T$. There are public generators P, $\tilde{P}$, and $e(P, \tilde{P})$ of the three groups, as well as another three points $K, K_0 \in G_1$ and $\tilde{K} \in G_2$. Finally, we have points X, Y, and Z in G_1, used for verification. All points must be random, i.e., no one should know the discrete logarithm of any point with respect to any other.

We could start with only two generators and use the multi-party assumptions to create the other points in a trusted set-up phase. Instead we assume they are pre-determined, for example by hashing "ALLOSAUR: Point K" onto a BLS curve [BLS03].

13.3 Core Accumulator Functions

The servers run these functions to maintain the accumulator. As the servers are not anonymous, we omit the sender argument to all posted messages, since it will always equal **self_id**.

The first function that the servers should run is **GEN**, which initializes the shared secrets and the first value of the accumulator. We also define

POST_ACCUMULATOR, which posts the value of the accumulator and waits for all other servers to post the same value, blaming any server who posts a different value. As long as a majority of players agree, then either there will always be consensus or some player will be blamed and the accumulator will abort.

GEN ()

1: // Blame any player that calls this setup after it is already started
2: **if** acc_counter ≥ 0 **then**
3: **for** messages gen on the messages board of type FNC with fun_type == Gen besides the first **do**
4: **MESSAGE_BOARD_POST**(type = BLAME, user = gen.sender)
5: **end for**
6: **RETURN**($\perp$)
7: **end if**
8: **loop**
9: // Create public keys randomly
10: $\alpha \leftarrow$ **GET_SHARED_RANDOM**()
11: $s_m \leftarrow$ **GET_SHARED_RANDOM**()
12: $r \leftarrow$ **GET_SHARED_RANDOM**()
13: sk $\leftarrow (\alpha, s_m)$
14: $\tilde{Q}$_share $\leftarrow \alpha$.share $\cdot \tilde{P}$
15: V_share $\leftarrow r$.share $\cdot P$
16: $\tilde{Q}_m$_share $\leftarrow s_m$.share $\cdot \tilde{K}$
17: $\tilde{Q}, (\tilde{Q}_{com}) \leftarrow$ **OPEN**($\tilde{Q}$_share)
18: $\tilde{Q}_m, (\tilde{Q}_{m,com}) \leftarrow$ **OPEN**($\tilde{Q}_m$_share)
19: // Random initial accumulator value
20: $V \leftarrow$ **OPEN**(V_share)
21: // If public values are trivial, check that they were produced honestly and either repeat or blame
22: **if** $\tilde{Q} = \mathcal{O}$ or $\tilde{Q}_m = \mathcal{O}$ **then**
23: shares$_\alpha \leftarrow$ **CHECK_SHARED_VALUE**(α)
24: shares$_s \leftarrow$ **CHECK_SHARED_VALUE**(s_m)
25: **for** $j \in$ **servers** **do**
26: **if** $\tilde{Q}_{com}[j] \neq$ shares$_\alpha[j]\tilde{P}$ or $\tilde{Q}_{m,com}[j] \neq$ shares$_s[j]\tilde{K}$ **then**
27: **MESSAGE_BOARD_POST**(type = BLAME, user = j)
28: **end if**
29: **end for**
30: **else**
31: Break loop
32: **end if**
33: **end loop**
34: aux, aux$_{\text{upd}} \leftarrow \emptyset$// list of elements currently in accumulator
35: acc $\leftarrow (V, \tilde{Q}, \tilde{Q}_m)$
36: acc_counter $\leftarrow 0$
37: gen_commit $\leftarrow (\tilde{Q}_{com}, \tilde{Q}_{m,com})$
38: wits $\leftarrow$empty dictionary// dictionary of valid witnesses
39: Y $\leftarrow \emptyset$// list of all elements ever added
40: **POST_ACCUMULATOR**(acc, acc_epoch)

41: **RETURN**$(\perp)$

Program 52: Accumulator generator function. Creates the two constant "public-key" components of the accumulator and picks a random starting value.

POST_ACCUMULATOR (A, epoch)

1: $\mathsf{acc\_mess} \leftarrow (\mathsf{type} = \mathsf{ACC}, m.\mathsf{epoch} = \mathsf{epoch}, m.\mathsf{accumulator} = A)$.
2: **MESSAGE_BOARD_POST**$(\mathsf{acc\_mess})$
3: **for** $j \in \mathsf{servers}$ **do**
4: $m \leftarrow$ **WAIT**$(j, (m.\mathsf{type} == \mathsf{ACC}, m.\mathsf{epoch} == \mathsf{epoch}))$
5: **if** $m.\mathsf{accumulator} \neq A$ **then**
6: **MESSAGE_BOARD_POST**$(\mathsf{type} = \mathsf{BLAME}, m.\mathsf{user} == j)$
7: **end if**
8: wait_number $\leftarrow$ wait_number $+ 1$
9: **end for**

Program 53: Updates the value of the accumulator on the bulletin board, and ensure that other players post a matching value.

To avoid unnecessary MPC, we perform one inversion every time an element is added to the accumulator, and one extra inversion the first time an element is added. Since the inversion during this addition is posted publicly, this avoids the need for any Wit_s function: users who want their witness simply read the public messages posted during the inversion. In practice, servers could condense this data by posting a list of current witnesses.

ADD posts a new accumulator, which is technically unnecessary as the value does not change. Doing so makes it simpler for our proofs to match changes to the accumulator with posted epochs.

ADD (y, proof)

1: // Blame any player that tries to add before the accumulator is initialized.
2: **if** acc_epoch < 0 **then**
3: Let sender be the sender ID of the message that started this **ADD** function
4: **MESSAGE_BOARD_POST**$(\mathsf{type} = \mathsf{BLAME}, \mathsf{user} = \mathsf{sender})$
5: **end if**
6: // If y has never been added before, attempt to create a long-term signature.
7: **if** $y \notin$ Y **then**

```
    SIGN(y, proof)
end if
Y ← Y ∪ {y}
Add the element if it is not already in the accumulator
if y ∉ aux then
    aux ← aux ∪ {y}
    aux_upd ← aux_upd ‖ {(y, 1)}
    acc_epoch ← acc_epoch + 1
    // This message tells the user that their first/new witness is about
    to be posted
    MESSAGE_BOARD_POST(
        type = DAT, data = ("witness", y, open_number, acc_epoch))
    // Produces a witness V' for y
    V' ← AFF_INV_ACC(y)
    wits[y] ← V' // y would not be in the accumulator previously
    // Ensure other servers posted the messages necessary for y to find
    their witness
    for j ∈ servers do
        if data_set does not contain a message with data
        ("witness'', y, open_number − 4) from player j then
            MESSAGE_BOARD_POST(type = BLAME, user = j)
        end if
    end for
end if
POST_ACCUMULATOR(acc, acc_epoch)
RETURN(⊥)
```

Program 54: Adds an element to the accumulator.

When a user is first added, they need a signature (separate from their witness) that will be static throughout execution. This is done via SIGN.

```
SIGN (y, proof)

// Check that the user knows the discrete log of R_ID via a Schnorr
proof
(h, s, R_ID) ← proof
K' ← sK + hR_ID
if RANDOM_ORACLE(R_ID, K') ≠ h then
    RETURN(⊥)
end if
// Compute the signature with MPC, implicitly posting it on the public
message board
AFF_INV_SIGN(y, R_ID + K_0)
```

9: **RETURN**($\perp$)

Program 55: Creates the long-term component of a user's witness, if they send a valid proof

Both **SIGN** and **ADD** require affine inversion (i.e., $(y, P) \mapsto \frac{1}{y+s}P$ for a secret s). There are two near-identical functions defined to do this,

AFF_INV_ACC and **AFF_INV_SIGN**.

Here we see that because the resulting value can immediately be verified using pairings, there is no need for extra MPC verification. If the pairing check fails, then servers know that the MPC was not performed correctly and they can open all secrets to decide who did not do the correct computation.

An extension to our protocol could likely do this non-destructively with t-out-of-n secret sharing for $t < n$, and some homomorphic commitment to shared secrets. We leave this for future work.

AFF_INV_ACC (y)

1: // Compute a single finite field inverse with a Beaver triple
2: $\alpha, s \leftarrow$ sk
3: // Adding y to one share creates a share of $y + \alpha$
4: **if** $i == 1$ **then**
5: $(y + \alpha)$_share $\leftarrow y + \alpha$.share
6: **else**
7: $(y + \alpha)$_share $\leftarrow \alpha$.share
8: **end if**
9: $r \leftarrow$ **GET_SHARED_RANDOM**()
10: $(a, b, c) \leftarrow$ **GET_SHARED_TRIPLE**()
11: w_share $\leftarrow$ **INVERT**$((y + \alpha)$_share$, r$.share$, (a$.share$, b$.share$, c$.share$))$
12: V'_share $\leftarrow w$_share $\cdot V$
13: $V' \leftarrow$ **OPEN**(V'_share)
14: // Check that the computation worked correctly; if not, find the malicious server
15: **if** $e(V', y\tilde{P} + \tilde{Q}) \neq e(V, \tilde{P})$ **then**
16: Start a blame process
17: // Break the accumulator so the revealed secret is useless
18: acc_epoch $\leftarrow$ acc_epoch $+ 1$
19: **POST_ACCUMULATOR**($\perp$, acc_epoch)
20: **MESSAGE_BOARD_POST**(BLAME, data = START)
21: Check that accumulator was properly generated
22: $(\text{shares}_\alpha, \text{shares}_s) \leftarrow$ **CHECK_GEN**()
23: Check the inversion itself

```
    shares_α[1] ← shares_α[1] + y
    shares_w ← CHECK_INVERSE(open_number − 4, shares_α, r, a, b, c)

    shares_v[j] ← shares_w[j]V for all j
    CHECK_OPEN(shares_v, open_number − 1)
end if
RETURN(V′)
```

Program 56: Affine inversion of accumulator value.

AFF_INV_SIGN (y, X)

```
// Compute a single finite field inverse with a Beaver triple
α, s ← sk
if i == 1 then
    (y + s)_share ← y + s.share
else
    (y + s)_share ← s.share
end if
r ← GET_SHARED_RANDOM()
(a, b, c) ← GET_SHARED_TRIPLE()
w_share ← INVERT((y + s)_share, r.share, (a.share, b.share, c.share))
R_m_share ← w_share · X
R_m ← OPEN(R_m_share)
//Check that the computation worked correctly; if not, find the malicious server
if e(R_m, K̃) ≠ e(X, yK̃ + Q̃_m) then
    acc_epoch ← acc_epoch + 1
    POST_ACCUMULATOR(⊥, acc_epoch)
    MESSAGE_BOARD_POST(BLAME, data = START)
    (shares_α, shares_s) ← CHECK_GEN()
    shares_s[1] ← shares_s[1] + y
    shares_m ← CHECK_INVERSE(open_number − 4, shares_s, r, a, b, c)

    shares_R[j] ← shares_m[j]X for all j
    CHECK_OPEN(shares_R, open_number − 1)
end if
RETURN(R_m)
```

Program 57: Affine inversion of second part of public key. The logic is identical to **AFF_INV_ACC**.

If the check fails and the servers start a blame, they will need to verify that **GEN** was computed correctly, as follows:

CHECK_GEN ()

```
// Check that secrets were initially well-formed
blames ← ∅
α, s ← sk
shares_α ← CHECK_SHARED_VALUE (α)
shares_s ← CHECK_SHARED_VALUE(s_m)
// Ensure the public keys matched the secrets
(Q̃_com, Q̃_m,com) ← gen_commit
for j ∈ servers do
    if shares_α[j] · P̃ ≠ Q̃_com[j] then
        blames.add(j)
    end if
    if shares_s[j]K̃ ≠ Q̃_m,com[j] then
        blames.add(j)
    end if
end for
if blames ≠ ∅ then
    for j ∈ blames do
        MESSAGE_BOARD_POST(type = BLAME, user = j)
    end for
    RETURN ⊥ to RUN_PLAYER_HONEST
end if
RETURN(shares_α, shares_s)
```

Program 58: Reveals all accumulator secrets and checks that they were posted honestly.

The servers maintain a set **wits** of witnesses for each user, and during deletions, they update each of these witnesses. These updates do not require any secret data. Since Nguyen's accumulator has the property that the value of the accumulator after deleting an element y is the same as the value of a witness for y, then the servers can effectively delete an element from the accumulator without any MPC by simply using the witness that they have stored.

DEL (y)

```
// Blame if this function comes before initialization of the accumulator
if acc_epoch < 0 then
    Let sender be the sender ID of the message that started this DEL
    function
    MESSAGE_BOARD_POST(type = BLAME, user = sender)
    RETURN
```

```
6: end if
7: if y ∈ aux then
8:    aux ← aux \ {y}
9:    aux_upd ← aux_upd‖{(y, V)}
10:   // V′ is the new accumulator, the deleted element's witness
11:   V′ ← wits[y]
12:   // Remove y from wits and update all other witnesses
13:   for y′ ∈ wits do
14:      wits[y′] ← 1/(y−y′) (wits[y′] − V′)
15:   end for
16:   acc ← (V′, Q̃, Q̃_m)
17:   acc_epoch ← acc_epoch + 1
18: end if
19: POST_ACCUMULATOR(acc, acc_epoch)
20: RETURN(⊥)
```

Program 59: Accumulator Delete function

13.4 Private User Functions

Witnesses. The first function every user runs is WIT. Here the user selects a random long-term secret and masks it, then asks for their ID y (given to them by the adversary) to be added to the accumulator, along with a signature. Once this is done, they wait for their new witness to be posted to the public message board, after which they read and store it.

```
WIT (y)
1: if user_id is already initialized or role = server then
2:    RETURN
3: end if
4: // Wait for the first accumulator
5: while |accs| = 0 do
6:    RETURN(5)
7: end while
8: pseud ←pseudonym of initial Wit message
9: witness ← (0, O, (O, O))
10: next_witness ← 0
11: // Long-term secret
12: sk ←$ F*_q
13: R_ID ← skK
14: k ←$ F*_q
15: h ← RANDOM_ORACLE(R_ID, kK)
```

16: proof $\leftarrow (h, k - h\mathsf{sk} \mod q, R)$ // basic Schnorr proof
17: // Request the servers to add them to the accumulator
18: **MESSAGE_BOARD_POST**(type = FNC, fun_type = Add, data = $(y$, proof$)$, sender = pseud)
19: // Wait for a long-term signature
20: **WAIT** for messages from all server players of type open_commit and open_reveal such that:
21: - the ID numbers all equal
22: - they assemble to a value R_m
23: - R_m satisfies $e(R_m, y\tilde{K} + \tilde{Q}_m) = e(R_{ID} + K_0, \tilde{K})$
24: // Find their first witness
25: $(n, C) \leftarrow$ **FIND_WITNESS**$(y, 0, \infty)$
26: **witness** $\leftarrow (n, C, R_m)$
27: **next_witness** $\leftarrow n$
28: // tracks in-progress witness epochs
29: **user_id** $\leftarrow y$
30: **RETURN**

Program 60: Initializes a user. Creates a Schnorr proof of their secret sk and posts the proof as part of an Add message, so they obtain both a witness and a long-term signature.

FIND_WITNESS $(y$, min_epoch, max_epoch$)$
1: // Look for the message the servers post which indicates that they are starting the witness computation
2: **if** **data_set** contains a data message from any player with data = ("witness", $y, n \in \mathbb{N}, m \in \mathbb{N}$), $m \leq$ max_epoch **then**
3: open_id, epoch $\leftarrow n, m$ from the latest message of this type with $m \leq$ max_epoch
4: **else**
5: **RETURN**$(\perp)$
6: **end if**
7: $Cs \leftarrow$ empty array
8: // Retrieve the appropriate messages. The wait returns immediately, since the above check ensures the messages are already posted
9: **for** $i \in$ servers **do**
10: c $\leftarrow$ **WAIT**$(i, (\text{type} == \text{DAT}, \text{data} = (\text{open\_commit}, \text{open\_id} + 3, c)))$
11: o $\leftarrow$ **WAIT**$(i, (\text{type} == \text{DAT}, \text{data} = (\text{open\_reveal}, \text{open\_id} + 3, o)))$
12: $Cs[i] \leftarrow$ **OPEN**(c, o)
13: **end for**
14: $C \in G_1 \leftarrow$reconstruct shares in Cs
15: **RETURN**(i, C)

Program 61: Finds a witness posted on the public message board as a result of an MPC affine inversion.

Updates. Update messages request users to update to a specific epoch. Users first ensure that this update makes sense (e.g., they do not have a witness for a later epoch) and then (mainly for readability) they call subroutines to return the updated value. We allow users to compute several updates at once. By the nature of the update process, a user can start the update process from epoch n_1 to epoch n_2 before they have a valid witness for epoch n_1; they simply wait for the previous update to finish before continuing. Importantly, once a user *starts* an update to an epoch n, they will not start any more updates to epochs less than n, even if they have not yet finished the update.

```
UPDATE(new_epoch)
1:  if next_witness ≥ new_epoch
        or |accs| < new_epoch
        or next_witness == 0 or role = server
        or user_id = ⊥ then
2:      RETURN(⊥)
3:  end if
4:  // Ensures that users only update once per round, and update greedily

5:  if there is a function message of type Upd_u posted in the current round
    with epoch > new_epoch then
6:      RETURN(⊥)
7:  end if
8:  // Signals to other functions that a new witness update is in progress
9:  pseudonym ← psuedonym of update request message
10: last_epoch ← next_witness
11: next_witness ← new_epoch
12: // Start protocol to obtain the new witness (takes several rounds)
13: witness ← RETURN_UPDATE(
        witness, last_epoch, new_epoch, pseudonym)
14: // Give the new witness to any new updates that started
15: for all entries in wit_waits with key new_epoch do
16:     wit_waits[new_epoch] ← witness
17: end for
18: RETURN(⊥)
```

Program 62: Witness update function. This mainly manages epochs, ensuring users update to a more recent epoch.

To update witnesses, we use the batch update protocol of [VB20] but offload this computation to the servers. Since the batch update only requires evaluating a polynomial based on *public* data (a list of deleted users and previous accumulator values) the servers can compute this using only this public data, carrying no risk of revealing secret information to the user. Instead, the security goal here is that the user keeps their ID secret. Thus, a user first splits their ID according to a Shamir secret-sharing scheme.

Shamir secret-sharing has two main benefits, besides anonymity. Choosing a t-out-of-n scheme means that if there are at least t honest servers, corrupt players cannot enact a denial-of-service attack, as the honest servers provide enough data for the user to continue. For a user to verifiably blame a corrupt server for *not* posting something, especially when all messages are private, is a formidable challenge, and this lets us avoid the problem entirely.

The second benefit is that if the users receives $t+1$ shares, they can validate the result by checking that all shares lie on the same polynomial. Again, if there are at least t honest servers, if the results fit the same interpolated polynomial then the user knows it is the *correct* polynomial. In fact a user has a more direct integrity check – whether the update gave them a valid witness – but in the edge case where a user is deleted and their update *should* fail, an adversary could de-anonymize slightly by sending incorrect shares that produce a valid witness. While this case is somewhat pathological, the polynomial interpolation integrity check avoids it.

In case the returned witness is not valid, then either a server was dishonest or the user was genuinely deleted. One option for the user here is to wait for the remaining update messages from other servers, since (by assumption) there are enough honest servers to ensure a valid update. However, the user has no method to check *which* response was invalid. Instead, here we assume the user accesses the public message board for an expensive update, where they download all accumulator update data and compute the update themselves.

Once the user downloads this data, they can compute the batch update polynomials and compute what each server *should* have returned, via $\mathbf{BLAME_U}$. If any server deviates from the expected protocol, the user will notice and will post the server's messages to the public message board, requesting the remaining servers trigger a blame.

If the user downloads this information and finds that servers all behaved correctly, this assures the user that their ID was deleted from the accumulator (they could also determine this by checking the sequence of deletions posted to the message board). In this case, they check the message board to see if they were re-added to the accumulator. If they were, that means the servers posted a new witness for this user. The user retrieves this witness (the latest witness available, for the rare case in which the user was deleted and re-added more than once between updates), and then must update this witness to the desired epoch by recursing the entire procedure.

If the user was deleted but not re-added, they return with the identity point on the curve as their witness, which is always invalid.

In practice we expect users will not be deleted often, and they will know when

they are deleted from out-of-band data. This means the expensive download will happen rarely, if at all, so we do not expect it to be an excessive burden. This also means the remaining edge cases may never happen, but we want ALLOSAUR to handle them gracefully anyway.

```
RETURN_UPDATE (witness, old_epoch, new_epoch, pseudonym)
// This checks if there is already an update in progress. If so we add a request for that update to return its result here
if witness.epoch ≠ old_epoch then
    wit_waits.add(old_epoch, ⊥)
end if
// Perform the interactive protocol with the servers
(𝒲, 𝒟, Ws, Ds, ys) ← GET_UPDATE(
        witness, old_epoch, new_epoch, pseudonym)
// Get the posted accumulator for this epoch
(V, Q̃, Q̃_m) ← accs[new_epoch]
// Check if the MPC failed
if any element of 𝒲 or 𝒟 is ⊥ then
    // Download and verify the servers' behaviour
    for i over all senders of messages in Ws or Ds do
        BLAME_U (i, old_epoch, new_epoch, ys[i], Ws[i], Ds[i], pseudonym)
    end for
    RETURN(new_epoch, 𝒪, R_m)
else if any element of 𝒟 is 0 or witness invalid then
    // In this case the player's ID was deleted
    // They must now find a new witness, if it existss
    witness ← FIND_WITNESS(y, witness.epoch, new_epoch)
    // If they cannot find a new witness, they are no longer part of the accumulated set. They return an invalid witness.
    if witness == ⊥ then
        RETURN(new_epoch, 𝒪, R_m)
    else
        // If they find a new witness, they may need to update the new witness recursively
        (epoch, C, R_m) ← witness
        (V, Q̃, Q̃_m) ← accs[epoch]
        if epoch < new_epoch and e(C', yP̃ + Q̃) == e(V, P̃) then
            RETURN(RETURN_UPDATE(witness, epoch, new_epoch))
        end if
    end if
else
    // Valid non-zero points mean the update should work directly
    // Obtain the previous witness value C, whether directly or from a previous update
```

```
    if old_epoch ≠ witness.epoch then
      // Waits for previous update to finish
      while wit_waits[old_epoch] == ⊥ do
        RETURN(GetCallStack(36))
      end while
      (old_epoch, C, R_m) ← wit_waits[old_epoch]
      wit_waits.delete(old_epoch) // delete only one
    else
      (old_epoch, C, R_m) ← witness
    end if
    // Final steps of the batch update
    C' ← PROCESS_UPDATE(D, W, C)
    RETURN((new_epoch, C', R_m))
  end if
```

Program 63: Finds the latest valid witness. This checks validity and recurses as necessary.

CREATE_SHARE $([y_1, \ldots, y_k], N, t)$

```
p ← random array of size k, with entries as arrays of F_q of size t
for i in 1 to k do
  p[i][0] ← y_i
end for
// Create random points; not strictly necessary
x_1, ..., x_N ←$ F_q^N
shares ←empty array of size N
for j from 1 to N do
  // Treat p as coefficients of a polynomial
  shares[j] ← p(x_j)  mod q
end for
RETURN([x_1, ..., x_N], shares)
```

Program 64: Shamir secret share creation. Produces N shares of x such that t shares can reconstruct x.

OPEN_SHARE $([x_1, \ldots, x_N], \mathsf{shares}, t, \mathsf{indices})$

```
V ← t × t matrix
for i = 1 to t do
  j ← indices[i]
  ith row of V ← (1, x_j, x_j^2, ..., x_j^{t-1})
```

```
5: end for
6: p ← V^{-1} · shares
7: for i = t + 1 to |indices| do
8:    j ← indices[i]
9:    if p(x_j) ≠ shares[i] then
10:      RETURN(⊥)
11:   end if
12: end for
13: RETURN(p(0))
```

Program 65: Reconstructs a secret from shares. This is not an efficient method, but demonstrates the principal.

```
GET_UPDATE (old_epoch, new_epoch, pseudonym)
1: d ← new_epoch − old_epoch
2: // Instructs servers to expect update data
3: MESSAGE_BOARD_POST(type = FNC, fun_type = Upd_s, data =
   (new_epoch, old_epoch), sender = pseudonym)
4: k ← ⌊√d⌋
5: ys ← array of length k − 1, each entry initialized to an array of length
   N
6: // Take t as the number of honest players. Users do not actually have
   access to C but in practice the protocol should decide on a threshold
   expected to be greater than the number of dishonest servers
7: t ← N − |C|
8: (ys, xs) ← CREATE_SHARE([y, y^2, ..., y^{k−1}], t, N)
9: // Give each server shares of powers of their ID y
10: for j from 0 to N − 1 do
11:    PRIVATE_MESSAGE_BOARD_POST(type = DATA, data =
       (ys[0][j], ys[1][j], ..., ys[k − 1][j])), dest = j), sender = pseudonym)
12: end for
13: Ws ← array of length N initialized with ⊥
14: Ds ← array of length N initialized with ⊥
15: // Expect server responses in 2 rounds
16: for i = 1 to 2 do
17:    RETURN(GetCallStack(16))
18: end for
19: for j ∈ servers do
20:    if private_data_set contains a message m from j to pseudonym
          with m.data ∈ wit × G_1^m × F_q^m × h
          such that h = RANDOM_ORACLE(ys[j], new_epoch, old_epoch)
       then
```

```
      Let (wit, Ws, D) = m.data
      Ws[j] ← W
      Ds[j] ← D
    end if
  end for
  Reconstruct update information
  W ←empty list
  D ←empty list
  for ℓ from 0 to m do
    // Notation to use the second index of Ws and Ds
    W[ℓ] ← OPEN_SHARE(xs, Ws[all][ℓ], t, j : Ws[j][ℓ] ≠ ⊥)
    D[ℓ] ← OPEN_SHARE(xs, Ds[all][ℓ], t, j : Ds[j][ℓ] ≠ ⊥)
  end for
  RETURN(W, D, Ws, Ds, ys)
```

Program 66: Interactive part of witness update (user side).

PROCESS_UPDATE $(\mathcal{D}, \mathcal{W}, \mathsf{witness})$

```
new_witness ← witness
for (d, W) ∈ (D, W) do
  new_witness ← d^{-1}(new_witness − W)
end for
RETURN(new_witness)
```

Program 67: Given an array of tuples to update a witness, repeatedly applies them to finish an update.

UPD_HELP $(V, Y, y\_\mathsf{shares}, d, k, m)$

```
Y_1, ..., Y_m ← split Y into sub-lists of length k
V_1, ..., V_m ← split V into sub-lists of length k
Ds, Ws ← empty arrays
for i in 1 to m do
  d(x) ← ∏_{t=1}^{k} (Y_i[t] − x) as a polynomial mod q
  w_s ← ∏_{t=1}^{s} (Y_i[t] − x)  mod q for all s from 1 to k − 1 (w_0(x) = 1)
  // The inner product means to treat the coefficients of d as a vector
  // This multiplies the share of y^i by the coefficient of x^i in d(x), for
  all i
  D ← ⟨d, (1, y_shares)⟩
  v_s ← ⟨w_s, (1, y_shares)⟩ // include trailing 0s in w_s
  W ← ∑_{s=0}^{k−1} v_s · V_{i·k−s}
```

12: Ds.append(D)
13: Ws.append(W)
14: **end for**
15: **RETURN**(Ws, Ds)

Program 68: Computes the public update polynomial on the shares of some user ID provided. Servers use this for regular updates, and users call this during blames to verify a server computed it correctly. Our implementation uses slightly more efficient methods to perform the same computations.

When a user's update fails, they call **BLAME$_U$**. First it checks whether they were honestly deleted, and stops blaming if they were. Otherwise, they can check precisely which server failed, because they have all the public data the server was supposed to use. Once they find the dishonest server, they ask for the private messages to be publicly revealed, so that honest server players can blame the malicious server. In practice, this would involve just posting the signed responses from the malicious server.

BLAME$_U$ (id, old_epoch, new_epoch, y_shares, W, D, t, pseudonym)

1: Create an ordered list $\mathcal{L}$ of $(y_i, \{\text{Add}, \text{Del}\})$ of all addition and deletion function calls posted to the message board
2: Find all accumulator values for these changes
3: $\mathcal{V} \leftarrow$ empty dynamic array
4: **for** i from old_epoch to new_epoch **do**
5: **if** $\mathcal{L}[i] = (y, \text{Del})$ **then**
6: $\mathcal{V}$.append($(y, \textbf{accs}[i])$)
7: **end if**
8: **end for**
9: Let $Y = [y_1, \ldots, y_d]$ be the values of y added to $\mathcal{V}$
10: Let $V = [V_1, \ldots, V_d]$ be the first argument of the accumulator values added to $\mathcal{V}$
11: $k \leftarrow$ length(y_shares), $m \leftarrow$ length(W)
12: $W_{check}, D_{check} \leftarrow$ **UPD_HELP**(id, V, Y ,y_shares, d, k, m, t)
13: **if** $W_{check} \neq W$ or $D_{check} \neq D$ **then**
14: // Any mismatched server responses get revealed publicly to show that the server misbehaved.
15: Let n_1 be the index of the first messages to server id for the update
16: Let n_2 be the index of the last messages to server id for the update
17: **for** $n = n_1$ to n_2 **do**
18: **PRIVATE_MESSAGE_BOARD_REVEAL**(pseudonym, n)
19: **end for**
20: **MESSAGE_BOARD_POST**(type = FNC, fun_type = Blame, data = (id, old_epoch, new_epoch, n_1, n_2, t, y_shares),

```
            sender = pseudonym)
21: end if
```

Program 69: Accumulator Blame (user-side), used for failed witness verification.

Proofs. For verification, a user must ensure they have an up-to-date witness. Since this may require an update, the user pauses execution for as long as an update might take before continuing, to obscure whether or not they needed an update.

We decided that the updates performed during verifications should be ephemeral and the updated witness is deleted after verification. This is because we imagine a real-world implementation will update in regular, predictable intervals, but having an update that *precisely* matches the epoch of a specific verification would de-anonymize a user.

If the user has already updated their witness to a later epoch, they will prove membership with respect to that later epoch.

Once all of the anonymity is taken care of, we use exactly the same zero-knowledge proof of the original accumulator [Ngu05].

```
PROVE (verifier, epoch_request)

 1: if role = server
        or user_id = ⊥ then
 2:   RETURN()
 3: end if
 4: // If there is no consensus accumulator that epoch, return immediately

 5: if |accs| < epoch_num then
 6:   RETURN()
 7: end if
 8: pseudonym ← pseudonym that recevied original function call
 9: // Maximum witness epoch the user has or will have
10: start_epoch ← next_witness
11: ver_round ← current_round + 4
12: // Update witness if the requested epoch is too high
13: if epoch_request > start_epoch then
14:   witness ← RETURN_UPDATE(
          witness, start_epoch ,epoch_request, pseudonym)
15: end if
16: // Wait to ensure verifications happen at the same round, regardless of
    whether an update was necessary.
17: while current_round < ver_round do
```

```
    RETURN(GetCallStack(18))
end while
// Wait for the verifier to send a challenge
challenge_message ← PRIVATE_WAIT(sender = verifier,
        data = ("proof_challenge", c, epoch),
          round ≥ ver_round − 1, dest = pseudonym)
        for some c
challenge ← challenge_message.data.c
// Ensure that this player only responds to each challenge once
Delete challenge_message from private_data_set
// If the user has a more recent witness, verify for that epoch
epoch_request = max{epoch_request, start_epoch)
if e(C, yP̃ + Q̃) = e(V, P̃) then
    // This zero-knowledege proof is from Nguyen [Ngu05]
    r1, r2, r3, k0, ..., k7 ←$ Zq
    U1 = Rm + r1Z
    U2 ← C + r2Z
    R ← r1X + r2Y + r3Z
    T1 ← k1X + k2Y + k3Z
    T2 ← k4X + k5Y + k6Z − k7R
    Π1 = e(K, K̃)^k0 e(U1, K̃)^−k7 e(Z, K̃)^k4 e(Z, Q̃m)^k1
    Π2 = e(U2, P̃)^−k7 e(Z, P̃)^k5 e(Z, Q̃)^k2
    c ← RANDOM_ORACLE(challenge, V, U1, U2, R, T1, T2, Π1, Π2)
    s0 ← k0 + c · sk
    si ← ki + cri for i ∈ {1, 2, 3}
    si ← ki + cri−3y for i ∈ {4, 5, 6}
    s7 ← k7 + cy
    proof ← (U1, U2, R, c, s1, ..., s7)
else
    // Send empty proof if user has no valid witness
    proof ← ⊥
end if
m ← (type = DATA)
m.data ← ("proof", epoch_request, proof)
PRIVATE_MESSAGE_BOARD_POST(m, dest = verifier,
        sender = pseudonym)
RETURN(⊥)
```

Program 70: Prove knowledge of a valid witness.

Upon receiving a verification proof, any other player (user or server) can verify it. If the verification is malformed, including referencing an accumulator epoch that has yet to be posted, the verifier treats this as a failed verification.

VER (prover, epoch)

1: // Wait for a consensus accumulator for this epoch
2: **while** **accs**[epoch] contains $\perp$ **do**
3: **RETURN**(GetCallStack(2))
4: **end while**
5: pseudonym $\leftarrow$pseudonym that sent original message
6: **PRIVATE_MESSAGE_BOARD_POST**(type = FNC, fun_type = Prove, data = (epoch), dest = prover, sender = pseudonym)
7: // Wait 3 rounds for the user to all but one round of any necessary updates
8: **for** $i = 1$ to 3 **do**
9: **RETURN**(GetCallStack(8))
10: **end for**
11: // Post ephemeral challenge
12: challenge $\leftarrow_{\$} \{0,1\}^{2\lambda}$
13: **PRIVATE_MESSAGE_BOARD_POST**(type = DAT, data = ("proof_challenge", challenge, epoch), sender = pseudonym)
14: // Wait exactly one round for the user to respond
15: **RETURN**(15)
16: **for** $m \in$ **private_data_set** such that:
m.recipient = pseudonym, m.sender = prover,
m.type = DAT, m.data = ("proof", epoch, proof),
m.round = **CLOCK_GET_TIME**() $- 1$ **do**
17: // If the proof is malformed, treat it as failure
18: **if** m.data.proof $\notin G_1^3 \times (\mathbb{Z}_q)^8$ or epoch $>$ max index(**accs**) **then**
19: **OBSERVER_CHECK_PROOF**(
pseudonym, sender, result $= 0$, epoch)
20: Skip to next loop iteration
21: **end if**
22: $V \leftarrow$ **accs**[m.epoch]
23: $(U_1, U_2, R, T_1, T_2, c, s_1, \ldots, s_7) \leftarrow m$.data.proof
24: $T_1 \leftarrow s_1 X + s_2 Y + s_3 Z - cR$
25: $T_2 \leftarrow s_4 X + s_5 Y + s_6 Z - s_7 R$
26: $\Pi_1 \leftarrow e(K, \tilde{K})^{s_0} e(U_1, \tilde{K})^{-s_7} e(Z, \tilde{K})^{s_4} e(Z, \tilde{Q}_m)^{s_1} e(K_0, \tilde{K})^{c} e(U_1, \tilde{Q}_m)^{-c}$
27: $\Pi_2 \leftarrow e(U_2, \tilde{P})^{-s_7} e(Z, \tilde{P})^{s_5} e(Z, \tilde{Q})^{s_2} e(V, \tilde{P})^{c} e(U_2, \tilde{Q})^{-c}$
28: **if** $c \neq$ **RANDOM_ORACLE**(challenge, $V, U_1, U_2, R, T_1, T_2, \Pi_1, \Pi_2$) **then**
29: // Do nothing; we post failure at the end if necesssary
30: **else**
31: // Any success almost certainly matches the challenge, so note success and return
32: **OBSERVER_CHECK_PROOF**(
pseudonym, sender, TRUE, m.epoch)

```
    RETURN
  end if
end for
// Ending the loop means no message passed the proof. Record this as failure.
OBSERVER_CHECK_PROOF(pseudonym, sender, FALSE, m.epoch)

RETURN
```

Program 71: Accumulator Verification function (verifier)..

13.5 Private Server Functions

Servers receiving an update request will immediately compute the result using their public data and return it.

UPD_S (sender, curr_epoch, last_epoch)

1: **if** role = user **then**
2: **RETURN**()
3: **end if**
4: $d \leftarrow$ curr_epoch $-$ last_epoch
5: $k \leftarrow \lfloor \sqrt{d} \rceil$
6: $m \leftarrow \lceil d/k \rceil$
7: Let aux′ be all the tuples in aux_{upd} from last_epoch to curr_epoch
8: Let Y be an array of $y_1, y_2, \ldots, y_d$, the values of y corresponding to deletions in aux′
9: Let V be an array of $V_1, \ldots, V_d$, the values of V corresponding to deletions in aux′
10: y_share_message $= (y_1, \ldots y_{k-1}) \leftarrow$ **WAIT**(sender, (m.type == DATA, m.data $\in \mathbb{F}_p^{k-1}$))
11: y_shares $= (y_1, \ldots, y_{k-1}) \leftarrow y$_share_message.data
12: $W, D \leftarrow$ **UPD_HELP**$(V, Y, y_{\text{shares}}, d, k, m)$
13: // The tag is necessary to ensure that these values are associated to this request
14: // Without this tag, malicious users could claim the response is a malformed response to a different request.
15: tag $\leftarrow$ **RANDOM_ORACLE**(y_shares, curr_epoch, last_epoch)
16: $m \leftarrow$ (type = DATA, data = ("wit", W, D, tag))
17: **PRIVATE_MESSAGE_BOARD_POST**$(m$, sender$)$
18: **RETURN**$(\perp)$

Program 72: Server side of witness update.

When a user posts a blame message from a bad update, the servers execute this function which checks for all update messages to look for one which was malformed.

BLAME (sender, id, upd_end, upd_start, y_shares)

1: $W, y, D \leftarrow \perp$
2: // Find all revealed messages matching the update
3: Let blames be all messages $m \in$ **data_set** such that:
   m.sender = sender
   m.type = DATA
   m.data = ("wit", W, D, h) such that
   $W \in G_1^k$, $D \in \mathbb{F}_q^k$, and $h \in \{0,1\}^{2\lambda}$
   $h =$ **RANDOM_ORACLE**(y_shares, upd_end, upd_start)
4: **if** blames has length 0 **then**
5: **RETURN**($\perp$)
6: **end if**
7: $d \leftarrow$ upd_end $-$ upd_start
8: $k \leftarrow \lfloor \sqrt{d} \rfloor$
9: $m \leftarrow \lceil d/k \rceil$
10: Let aux′ be all the tuples in aux_{upd} from upd_start to upd_end
11: Let Y be an array of $y_1, y_2, \ldots, y_d$, the values of y corresponding to deletions in aux′
12: Let V be an array of $V_1, \ldots, V_d$, the values of V corresponding to deletions in aux′
13: // Replicate the computations of the accused server to check if they were done correctly
14: **for** all messages m in blames **do**
15: $W, D, h \leftarrow m$.data
16: $W_{check}, D_{check} \leftarrow$ **UPD_HELP**(V, Y, y_shares, d, k, m)
17: **if** $W_{check} \neq W$ or $D_{check} \neq D$ **then**
18: **MESSAGE_BOARD_POST**(type = BLAME, user = id)
19: **end if**
20: **end for**

Program 73: Accumulator Blame (server-side), used for failed witness verification.

13.6 Anonymity Control Programs

After a round is finished and players have posted all their messages, the observer program updates the anonymity sets. Recall that anonymity sets are already restricted from adversarial calls to **GAME_DEANONYMIZE**. Besides that, anonymity is reduced because:

- Users requesting a new witness are completely non-anonymous, since they

include their ID in the request!

- Users posting a verification proof must be in the set of users either in the accumulator or not, depending on the validity of the proof.
- Users requesting an update from epoch n_1 to epoch n_2 must be in the set of users which last updated to epoch n_1.

To manage this, whenever a function request is sent to a specific pseudonym p belonging to a user y, the observer program modifies the anonymity sets as follows:

- On a Wit request, the anonymity set for p is reduced to just $\{y\}$.
- On an Upd request, the anonymity set of p is reduced to all all users whose current witness epoch matches the current witness epoch of y, *and* who are also request to update to the same epoch as p. Further:
  - If the user is sent multiple Upd requests, they will only act on the one with the largest valid epoch
  - If y receives multiple update requests to *different* pseudonyms, the user will ignore most of them. To avoid doubt, the anonymity set is restricted to $\{y\}$.
  - The user's "current witness epoch" might be the the update of a previous but unfinished update call
  - If the user ID y was deleted and re-added before the epoch of the update, they will send another message 2 rounds after the first, whose start epoch matchs the epoch that y was re-added. Because only one element is added in each epoch, this completely de-anonymizes y; the anonymity set is reduced to $\{y\}$
- On a Prove request, the anonymity set is restricted similarly to updates. However, since **PROVE** does not change the user's internal state (witness updates during **PROVE** are ephemeral), the observer does not require other users y' in the anonymity set of p to have also been requested to Prove.
- A Ver request causes no restrictions, as it neither reads from nor writes to any user-specific variables.
- If the pseudonym sends a message with a proof, the anonymity set for that pseudonym is restricted to only users that are either in the accumulated set or not, depending on whether the proof is valid.

These anonymity set restricts are conservative. Multiple update requests sent to different pseudonyms of the same user might be difficult for an adversary to correlate; however, deciding precisely *how* much correlation the adversary gets is also difficult. By completely reducing the anonymity set in this case, an

adversary who does this has no hope of winning the indistinguishability game using such messages, because there are no two users among the anonymity set that the adversary can distinguish.

RESTRICT_ANONYMITY (fncs)

// For users without a witness yet, check if they will retrieve one from the message board this round
$\mathcal{Y}_0 \leftarrow \{y : \textbf{user\_epochs}[y] = 0\}$
for $y \in \mathcal{Y}_0$ **do**
  // A consensus accumulator containing y implies that **ADD** made a witness for y
  $n \leftarrow \min\{m : y \in \mathcal{S}[m], |\mathcal{A}[m]| = |\textbf{servers}|\}$
  if n exists **then**
    $\textbf{user\_epochs}[y] \leftarrow n$
    $\mathcal{Y}_0$.del(y)
  end if
end for
// All honest players
$\mathcal{Y} \leftarrow$ all user IDs y belonging to players not in $\mathcal{C}$
// Process proof messages
$M_{proofs} \leftarrow$ **MESSAGE_BOARD_GET_FUNCTIONS**(DAT)
Restrict M_{proofs} to all messages with m.data = ("proof", ...)
// Restrict based on whether the proof should be valid or not
for $m \in M_{proofs}$ **do**
  if not **PSEUDONYM_CHECK_CORRUPTED**(m.sender, $\mathcal{C}$) **then**
    $\textbf{anon\_sets}[m.\text{sender}] = \textbf{anon\_sets}[\text{m.sender}] \cap \mathcal{Y} \cap \mathcal{S}[m.\text{epoch}] \setminus \mathcal{Y}_0$
  end if
end for
$M \leftarrow$ **MESSAGE_BOARD_GET_FUNCTIONS**(FNC)
// Find multiple function requests to the same pseudonym, since users will also do so
for tuples $(m_1, m_2, \dots)$ of more than one message in M with the same recipient **do**
  $\textbf{used\_pseudonyms}$.add(m_1.recipient)
end for
// Ignore all messages to used pseudonyms
Restrict M to messages with recipients not in **used_pseudonyms**
// Witness issuance is completely de-anonymizing
for $m \in M$ such that m.type = FNC and m.fun_type = Wit **do**
  // If the user already exists, they will ignore witness requests
  if $y \notin \textbf{user\_epochs}$ **then**
    $y \leftarrow m$.data.user_id
    $\textbf{anon\_sets}[m.\text{recipient}] = \{y\}$

```
        user_epochs[y] ← 0
    end if
end for
// Process update requests
// Removing used pseudonyms means each m is sent to a unique pseudonym
M_upd ← {m ∈ M : m.type = FNC and m.fun_type = Upd_u}
// MY_upd will match update messages to the user ID for the update
MY_upd ← ∅
// Restrict to honest user updates to epochs with consensus
for m ∈ M_upd do
    if |A[m.epoch]| < |servers| then
        Skip to next loop iteration
    end if
    if not PSEUDONYM_CHECK_CORRUPTED(m.recipient, C) then
        MY_upd.add(m,
            user_IDs[PSEUDONYM_GET_IDENTITY(m.receiver)])
    end if
end for
// Find any user given multiple updates to different pseudonyms
MY_dup ← {(y, m) : there is more than one tuple (y, ∗) in MY_upd}
// Only the remaining users in MY_upd have any anonymity
MY_upd ← MY_upd \ MY_dup
for MY ⊆ MY_upd with the same value of m.epoch do
    Partition MY into M_1, ..., M_ℓ based on m such that:
        there are some n_1, ..., n_ℓ such that for all (y, m) ∈ MY_i,
        user_epochs[y] = n_i
    // Find users that were deleted and re-added
    Y_del,i ← {y' ∈ Y_i : y' ∈ S[n + 1] \ S[n], user_epochs[y'] ≤ n < m.epoch − 1}
    for i = 1 to ℓ do
        // Users without valid witnesses yet are ignored
        if user_epochs[y] = ⊥ or 0 then
            Skip to next loop iteration
        end if
        for (m, y) ∈ MY_i do
            if y ∈ Y_del,i then
                anon_sets[m.recipient] ← {y}
            else
                anon_restriction ← ∩{y : (∗, y) ∈ MY_i} \ (Y_del,i ∪ Y_0)
                p ← m.recipient
                anon_sets[p] ← anon_sets[p] ∩ anon_restriction
            end if
```

```
    end for
  end for
  // Update the maximum witness each user is expected to have
  for (m, y) ∈ MY do
    user_epochs[y] ← m.epoch
  end for
end for
// De-anonymize users with multiple update messages
for (m, y) ∈ MY_dup do
  anon_sets[m.recipient] ← {y}
  user_epochs[y] ← max{user_epochs[y], m.epoch}
end for
// Restrict anonymity based on proof requests
M_ver ← {m ∈ M : m.type = FNC and m.fun_type = Prove}
Restrict M_ver to only m with m.epoch such that
      A[m.epoch] = |servers|
for m ∈ M_ver do
  y ← user_IDs[PSEUDONYM_GET_IDENTITY(m.receiver)]
  // Ignore uninitialized users (who will ignore this request)
  if user_epochs[y] = 0 or ⊥ then
    Skip to next loop iteration
  end if
  // Get users with same current witness epoch
  Y_upd ← {y' : user_epochs[y'] = user_epochs[y]}
  Y_del,i ← {y' ∈ Y_i : y' ∈ S[n + 1] \ S[n], user_epochs[y'] ≤ n <
  m.epoch − 1}
  // Users that were deleted and re-added will be completely de-
  anonymized
  if y ∈ Y_del then
    anon_sets[m.receiver] = {y}
  else
    anon_sets[m.receiver] = anon_sets[m.receiver] ∩ Y_upd \ (Y_del ∪ Y_0)
  end if
end for
used_pseudonyms ← used_pseudonyms ∪
      {all recipient pseudonyms in M}
```

Program 74: Updates the anonymity sets based on a posted message.

14 Proof Outline

In the next sections we prove security. Since our formalism executes functions asynchronously except for the round structure, our first task is to show that the server players proceed through the same steps in the same order. Section 15

shows that honest server players execute the same functions in the same order as each other during all core accumulator functions (Gen, Add, Del, and all of their subroutines), which follows straightforwardly from the fixed ordering of the message board. Section 16 shows that all honest server players execute the same steps in the same rounds, mainly by showing that when server players wait for each other (e.g., to open a secret share), this should take only one round.

From here we need to show that the adversary must execute the same steps as honest server players or they will be blamed. Section 17 shows this based on binding property of the commitment scheme, which means that if the MPC arithmetic goes wrong, the honest players can reliably determine who misbehaved by opening various commitments to the secret data. Once this is done, we can talk about "the accumulator", and we show that it is maintained in the same way as in the single-server case in Section 18. This includes showing that the internal database of witnesses that each honest server player maintains, **wits**, contains valid witnesses for each ID y that was added to the accumulator.

Then Section 19 shows that the servers correctly issue witnesses during the Add operations and that the users will be able to find these witnesses. Then we must show that user updates work correctly. Section 20 shows that when a user updates its witness, whether as part of **UPDATE** or **PROVE**, it always completes between 2 and 4 rounds. We also show in this section that if a user has a witness for some epoch, it is correct: it is a valid witness if the user is in the accumulated set, and invalid otherwise. This allows us to state that verifications work correctly: an honest user will produce a valid proof during verification if and only if they are in the accumulated set. While the basic correctness follows readily from inspecting the polynomial arithmetic introduced in [VB20], the bulk of this section deals with how **UPDATE** handles different edge cases, like users being deleted between updates.

Then Section 21 uses the results on progress to show that requests to Add and Del proceed in a timely fashion. This implies that verification requests finish within the required time to prevent the accumulator from timing out. With all of this groundwork in place, we can then show correctness, Definition 10.1, in Theorem 21.9.

For soundness and indistinguishability, we require some basic results on simulating the accumulator for an adversary. In Section 22 we show that, we can force an adversary to be the first to open its shares when opening any shared secret, allowing a simulator to indistinguishably open any secret share to any value it needs by reprogramming the random oracle.

To show soundness, we first need to construct an extractor. Our extractor comes from the soundness proof of [Ngu05], and works by replaying an adversary and reprogramming the random oracle. This proof is only correct thanks to our single-round response to verifier challenges, which we suspect highlights that there is a genuine security threat in our model if there are no defenses against replay attacks.

We must then show that all extracted witnesses belong to the adversary. This relies on our new group problem, which we define in Section 24 and prove secure later in Section 28. In Section 24 we use the hard group problem on two different

aspects of the scheme to show that extracted witnesses must match a valid accumulator *and* belong to the adversary. This shows soundness, Theorem 24.6, as defined in Definition 10.2.

Finally we must prove indistinguishability, Definition 10.3. It is straightforward in Section 25 to show that we can simulate the messages of both updates and proofs, since these are based on secret-sharing and zero-knowledge proofs, respectively. More challenging is addressing all metadata, in Section 26. To do this we define the notion of what messages a pseudonym might send in response to other messages, and show that if two pseudonyms are in the same anonymity set, they will respond in exactly the same way to all future messages. Further, we show that for the two functions which modify users' internaal state – **WIT** and **UPDATE** – all pseudonyms receiving such requests receive the same request in the same round. This ensures that the modifications to user state are the same through anonymity sets. All of this allows us to show that we can swap the pseudonyms of two users in the same anonymity set without the adversary noticing. As this demonstrates that there is no correlation between the adversary's behaviour and which pseudonyms are assigned to which user (at least within the same anonymity set), indistnguishability follows readily.

This proves all required security definitions. However, our definition of indistinguishability does not make any guarantees about anonymity. We thus define a pattern of adversarial inputs (and hence of user behaviour) in Section 27, which creates a non-trivial level of anonymity.

14.1 Notation

Throughout the proofs, we use the following terminology:

- "honest server player" refers to one of the players with id in **servers** that is not in the set of players corrupted by the adversary.
- "core accumulator functions" refers to **GEN**, **ADD**, and **DEL**.
- "user functions" refers to **WIT**, $\mathbf{UPDATE}_{s,u}$, **PROVE**, and **VER**.
- "consensus accumulator" refers to an accumulator for an epoch n such that all server players have posted an accumulator value, and all their accumulator values match.

14.2 Assumptions

We make the following assumptions through all the following proofs.

Every proof reasons about the execution of subroutines during execution of

$$\begin{matrix}\mathsf{answer},\\ \mathsf{state}\end{matrix} \leftarrow \mathbf{ACCUMULATOR\_GAME}(\lambda, N, k).$$

The value k (the maximum number of corrupt players) is chosen so that $k < \frac{1}{2}N$, and the global constant **BLAME_THRESHOLD** is greater than k and less than $N - k$.

The value $\perp$ can be used without error in any operation, but composing $\perp$ with any other data returns $\perp$, i.e., $x \leftarrow x + \perp$ means $x \leftarrow \perp$.

15 Ordering Proofs

15.1 Outline

In this section we prove that honest server players follow the same steps in the same order throughout core accumulator functions. Since these functions branch at several points, we must show that honest server players follow the same branches.

First, Proposition 15.1 shows that all players run the same core accumulator functions in the same order, and Proposition 15.2 shows that their data sets (i.e., public messages) are the same. Both follow straightforwardly from the append-only public message board. Then we show, Proposition 15.3, that if **GEN** does not run first, then an adversary gets blamed; the checks in every other function ensure this.

To show that the branching is the same, we must show that the variables that the branching depends on will be the same. First it is straightforward to show that **aux** and **Y** are the same (Proposition 15.4), since they depend only on arguments to the core accumulator functions. The only remaining branching is in the "blame branches" of the affine inversion computations, which rely on the outputs of multi-party computations, so we must prove consistency of these computations.

For this we start by showing in Proposition 15.5 that **WAIT** returns the same value for all honest players, which follows because if players wait for the same kind of message posted to a public message board, the first satisfying message must be the same for all of them. Since **OPEN** is just many calls to **WAIT**, this implies (Proposition 15.6) that **OPEN** also returns the same value, at least if it waits for the same shares.

At this point we need to ensure that the arguments to **OPEN** are consistent, so that players are not waiting for different shared secrets to open. This is fundamentally inductive, since the inputs to **OPEN** depend on previous branches of computation, which depend on previous *outputs* of **OPEN**. We first argue that if no blame branch is taken, then the numbering of each call to **OPEN** is consistent (Proposition 15.7), which is clear because it is incremented sequentially. This means that if no blame occurs, the indices of the shared randomness and Beaver triples are consistent (Proposition 15.8), again since they are incremented sequentially. Together this gives the final result, Proposition 15.9 all the arguments to **OPEN** will have the same indices, so they return the same outputs, so the variables used to decide whether to enter the blame branch are consistent.

We summarize the section with Theorem 15.10: all honest server players follow the same branching. This follows from the previous logic.

15.2 Proofs

To start, we show basic properties of the order of execution.

Proposition 15.1. *Let $\mathcal{L}_i(n)$ be an ordered list of all the core accumulator functions that an honest server player i has started to run, including the arguments to the function, by the end of round n. Then for any two honest server players i and j, either $\mathcal{L}_i(n)$ is a substring of $\mathcal{L}_j(n)$ or vice verse.*

Proof. Honest server players only call core accumulator functions that were posted to the message board. The message board functionality provides a consistent ordering, and honest players execute functions from this board via a first-in-first-out queue. □

Proposition 15.2. *In each round, all honest players have the same data in* **data_set** *in the same order. Moreover, if* **data_set** $= \mathsf{D}$ *in round i, in all future rounds* **data_set** $= \mathsf{D}\|\mathsf{D}'$ *for some* D'*, i.e., it is append-only.*

Proof. The execution of **RUN_PLAYER_HONEST** is the same for all honest players, and adds data to **data_set** in the order it is posted on the message board, which is the same data in the same order for all players. Moreover, from the start of each round to the end, the message board does not change, thus each player has the same data. Append-only is trivially true by construction. □

Note that **private_data_set** will likely be different.

Proposition 15.3. *If* **ADD** *or* **DEL** *is posted to the message board before* **GEN***, the adversary will be blamed and the accumulator will abort.*

Proof. All of the stated functions begin by checking if **acc_counter** < 0, blaming the sender of the function message if it is. Since this variable is initialized to -1, and only incremented in **GEN**, the result follows. □

Proposition 15.4. *All honest server players have the same data in* **aux** *and* **Y** *at the start of the execution of the same accumulator function.*

Proof. The object **aux** is only modified in **ADD** and **DEL**. While a partial execution of either function will push these variables out of sync across honest server players, this proposition only states that they are the same at the start of the execution of one of these functions, so we need only show that once honest server players *finish* these functions, they are in sync. It will take more work to show that the players are in sync during execution.

The only branch in the value of **aux** is based on the value of **aux** itself. Honest players execute the **ADD** and **DEL** functions in the same order as each other. Thus, they will perform the same modifications to **aux**, and hence they will always have the same value when they start **ADD** and **DEL**, so they will keep the same value by induction.

For **Y** the same logic applies, with the slight caveat that the execution depends also on the argument to proof – but since this is the same for all players, their execution follows the same branches. □

We now argue that when players wait for messages from each other, they will always return with the same message. We do not yet show that this message is correct or even well-formed. We call a message that satisfies the conditions of a **WAIT** a *satisfying message*.

PRIVATE_WAIT does not have this property, as one player may send a different satisfying message to each other player waiting for a private message from them.

Proposition 15.5. *All honest server players that call* **WAIT** *with the same argument either will return the same argument, if they return an argument at all.*

Proof. Since each player's **data_set** is the same and append-only, the first instance of a message matching some condition is well-defined and consistent across players. Therefore, if two players return a message, they must select the same message. □

We next argue that **OPEN** produces consistent values.

Proposition 15.6. *Each honest server player calls* **OPEN** *with sequentially increasing values of* **open_number**. *Any honest server players which return with a value from a call to the* **OPEN** *with the same value of* **open_number** *will all return the same value. Similarly, if honest server players call* **CHECK_SHARED_VALUE** *with shares y that have the same* id *and return, they will also return the same value.*

Proof. The first statement is clear by the incrementing of **open_number** during calls to **OPEN**.

By Proposition 15.5, all honest players will obtain the same value from each call to **WAIT**. By the structure of **OPEN**, all honest players will wait for the same messages, with one exception: each honest player i will not wait for a message from itself.

For such an honest player i, i will post the commitment and opening of their message, and only post one message for each value of **open_number**. This satisfies the requirements of the **WAIT** call of the other honest players. Thus, all honest players will end up with the same values of commits and open. Here, if an adversarial player posted a malicious commitment and/or opening, it may corrupt the output of **OPEN_COMMIT**; however, since **OPEN_COMMIT** is deterministic, each honest server player will obtain the *same* corrupted output. That is, no matter what is posted, all honest server players obtain all the same shares, and thus return the same value.

The exact same logic applies to **CHECK_SHARED_VALUE**: all honest server players post the value that matches the **WAIT** for other honest server players. □

This shows that all honest server players have the same data, except for individual secrets. We refer to a "blame" as any time an honest server player

starts to execute the branches in AFF_INV_ACC or AFF_INV_SIGN based on a malformed result.

These values are consistent, but there is no guarantee they are correct. If the players get out of sync, they may attempt to open different shares. This does not occur either; because they always open the same values, they will always branch the same way.

Proposition 15.7. *If no honest server player enters the "blame" branch of* AFF_INV_ACC *or* AFF_INV_SIGN*, then all honest server players calling* OPEN *with the same value of* open_number *will call it in the same core accumulator function, with the same arguments, with same call stack and line numbers.*

Proof. We have already shown that honest server players execute the same core accumulator functions in the same order. Thus we must only show that, within such a function, that all branching leading to different calls to OPEN will branch the same way for all players. We thus argue by induction. For the base case, the first call to OPEN occurs in the first call to GEN, which will branch the same way because it is the first call to GEN.

For the inductive case, we ignore all branching that does not lead to different calls to OPEN. All remaining branching in core accumulator functions depends on either aux or (in DEL) the outputs of previous calls to OPEN. By Proposition 15.6, all players obtain the same values for previous calls to OPEN, and inductively, these correspond to the same values in the code (e.g., they will obtain the same value for $\tilde{Q}$). Thus, they will branch the same way in each function, proving the statement. □

At this point we have not yet shown that any call to OPEN will necessarily complete, we have only given guarantees on its output if it does. We now argue that shared randomness is dealt with sensibly.

Proposition 15.8. *If no honest server player enters the "blame" branch of* AFF_INV_ACC *or* AFF_INV_SIGN*, then all calls to* GET_SHARED_RANDOM() *and* GET_SHARED_TRIPLE() *by honest server players that occur at the same lines in the same core accumulator functions with the same arguments will return secret shares with the same ID.*

Proof. The previous arguments about branching apply here as well; all calls to these functions occur within branching that we argued is identical across all honest server players. Thus, we can sequentially number all calls to these functions and these numbers will be consistent across honest server players. Since each one will increment either random_share_counter or triple_share_counter each time, these IDs will increment with the calls to these functions. □

During GEN, each player posts their share of $\tilde{Q}$ and $\tilde{Q}_m$, equal to $\tilde{P}$ or $\tilde{K}$ multiplied by their share of α and s_m (respectively), and honest server players store these shares in gen_commit. During GET_SHARED_RANDOM, each player posted a commitment to these shares. Thus, when CHECK_GEN

opens these commitments, it should immediately show if a player misbehaved during this step. We show later that this works; for now we only show that CHECK_GEN returns consistent results for all honest server players.

Proposition 15.9. *Any honest server players that enter the "blame" branch of* AFF_INV_ACC *or* AFF_INV_SIGN *will follow the same branches within this execution.*

Proof. We show this for AFF_INV_ACC, but the same logic applies to AFF_INV_SIGN. First, if any honest server player gets "stuck" (i.e., permanently waiting) at any point, then the proposition statement holds vacuously.

Otherwise, the first branch point is in CHECK_GEN, but the logic of Proposition 15.8 shows that all honest server players will follow the same branches in it, and return with the same set. They then call CHECK_INVERSE. By Proposition 18.2 the V argument is the same; from previous propositions open_number is the same, and by Proposition 15.8, all honest server players have the same IDs for r, a, b, and c.

This means that within CHECK_INVERSE, they will all return with the same shares from each call to CHECK_SHARED_VALUE. From this, we can see that each call to CHECK_OPEN will branch the same way as well. Finally, this implies that CHECK_INVERSE, if it returns, will return the same values.

This means that shares_v will have the same value for all honest server players, and the next call to CHECK_OPEN will also have the same values and thus branch in the same way. This applies to both AFF_INV_SIGN and AFF_INV_ACC. □

This leads to our main consistency theorem:

Theorem 15.10. *All honest server players follow the same branches during execution of all core accumulator functions, except for possibly posting differing blames within* WAIT.

Proof. We must show this inductively over the branch points of execution, which could occur during GEN, DEL, and ADD or any of their subroutines. These are executed in the same order (Proposition 15.1), so our inductive hypothesis on branching also implies that all incremented variables will stay consistent across all honest server players.

By Proposition 15.3, all honest server players execute GEN first. The first branch in this is during GET_SHARED_RANDOM.

Since each call to GET_SHARED_RANDOM and GET_SHARED_TRIPLE sequentially increases random_share_counter or triple_share_counter, honest server players will wait for the same messages during calls to WAIT during these functions.

From Proposition 15.5, honest server players return with the same values from WAIT if it is called with the same arguments. Here we do not yet have a guarantee that some honest server players will not post the same blame as others

if they were waiting for much longer (we later show this does not happen), but this does not affect any other branching.

This means GEN branches in the same way until it reaches POST_ACCUMULATOR. Since the accumulator epoch is incremented the same for all honest server players, they again wait for the same messages, so they branch in the same way.

Next we reason about ADD. Players follow the same branch at Line 7 because Proposition 15.4 shows Y is the same. During SIGN, the value of proof is the same for all honest server players (it was part of a public message) and AFF_INV_SIGN follows the same branches during OPEN and INVERT by the same logic of incrementing the indices of random shares and Beaver triples. It then follows the same blame branch by Proposition 15.9. Thus, SIGN follows the same branches.

Then at Line 12 the honest server players again follow the same branches by Proposition 15.4, and during AFF_INV_ACC the same logic applies and they branch in the same way. Finally, they reach POST_ACCUMULATOR, which follows the same branches by the same logic as in GEN.

DEL has the same subroutines as ADD, so the proof is identical. □

16 Progress Proofs

16.1 Outline

In this section we prove that the adversary cannot stall progress of the accumulator without getting blamed. Mainly this will involve proving that during any WAIT calls, honest server players wait for the same data and do not become deadlocked.

Recall that WAIT is given a player ID and a condition, and it waits for that player to post a message satisfying the given condition, and then returns that message. All calls to WAIT during core accumulator functions (during OPEN, POST_ACCUMULATOR, or GET_SHARED_RANDOM) occur in a loop that iterates over all server players, waiting for a message from each. We refer to such a loop as a "WAIT loop".

Implicitly, throughout we use Proposition 16.1, which states that the conditions are always evaluable in polynomial time and satisfiable. Then we show that the numbering of OPEN, where many WAIT calls occur, is consistent among honest server players (Proposition 16.2). This allows us to sequentially number all calls to WAIT, Proposition 16.3, such that honest server players will call them in order, which is clear by Theorem 15.10 since execution follows the same branches. This means the WAIT loops are the same among the honest server players, so honest server players will post messages satisfying the requirements of the other players in the loop before starting the loop Proposition 16.4, and thanks to the numbering, these messages are unique.

All of this shows the consistency of the WAIT calls, so since the honest server players wait for the same message, they will return from each WAIT

in the same round (Proposition 16.5). Since users proceed through execution without interruption between calls to **WAIT**, this gives us our main Theorem 16.6, stating that honest server players will both call and return from the same **WAIT** in the same round.

That gives us two helpful corollaries: Corollary 16.7, honest users will not blame each other in a **WAIT** for not posting the required message in time, and Proposition 16.8, that a **WAIT** loop finishes in a bounded number of rounds or some corrupted player has delayed it and will be blamed.

16.2 Proofs

We implicitly use the follow proposition throughout.

Proposition 16.1. *All calls to* **WAIT** *by honest server players use a condition that is evaluable in polynomial time and for which it is possible for a satisfying message to exist.*

Proof. Clear by inspection of all calls to **WAIT**. □

Proposition 16.2. *All honest server players calling* **OPEN** *with the same value of* **open_number** *will call it in the same core accumulator function, with the same arguments, with the same call stack and line numbers.*

Proof. The proof is exactly the same as Proposition 15.7, except Proposition 15.9 guarantees that *all* branching is the same. □

We need to order the calls to **WAIT**. For this, we use **wait_number**.

This counter ensures each player numbers calls to **WAIT** sequentially. We have honest server players call **WAIT** for messages from themselves, which is technically unnecessary but makes the following proofs much cleaner. Since all the calls to **WAIT** occur in loops over messages from all server players (either in **OPEN**, **GET_SHARED_RANDOM**, or **POST_ACCUMULATOR**), in the same order, there exists one server player that all honest server players will wait on first. That player, if honest, could skip waiting on their own message, which would cause a vast array of off-by-one errors in these proofs, which would be inconsequntial since that player would immediately be forced to wait for a different honest player's message in the same wait loop). Instead we simply force all players to wait for their own messages.

Proposition 16.3. *All players which call* **WAIT** *with the same value of* **wait_number** *will return with the same message, and if* $n_1 < n_2$*, all players which execute* $\textbf{WAIT}_{n_1}$ *and* $\textbf{WAIT}_{n_2}$ *will call* $\textbf{WAIT}_{n_1}$ *before* $\textbf{WAIT}_{n_2}$*.*

Proof. Since honest server players follow the same branching by Proposition 15.9, they will reach the same **WAIT** in the same order. □

This justifies numbering the calls to **WAIT** sequentially, and using $\textbf{WAIT}_n$ to refer to the **WAIT** that each player will call when **wait_number** $= n$.

Proposition 16.4. *Before entering a loop of calls to* **WAIT** *for all server players, any honest server player i will post a message that will satisfy the requirements of all other honest server players in that loop waiting for a message from player i. Moreover, this will be the first and only message satisfying that* **WAIT**.

Proof. The first is clear by inspection. For the second, since calls to **WAIT** are only in **OPEN** and **POST_ACCUMULATOR**, they are indexed by **open_number** or an accumulator epoch, which both increment with each call, so no other message will match these indices. **WAIT** also checks the sender of a message and the message board functionality ensures this is added to each message, so no other player could post a satisfying message. □

We are now ready to show that wait calls must progress in a fixed number of rounds.

Proposition 16.5. *If at least one honest server player calls (not necessarily starts)* $\textbf{WAIT}_k$ *in round n_1, all honest server players will complete execution of* $\textbf{WAIT}_k$ *in the same round as each other, or by at most round $n_1 +$* **WAIT_THRESHOLD** *they will all have blamed the same player.*

Proof. Suppose more than one player calls a **WAIT** for a message from player i in round n_1. **WAIT** checks messages in **data_set**, but by Proposition 15.2, this is the same for all honest server players in each round. This means in each round, all honest server players will reach the same conclusion about whether **data_set** contains a message satisfying the conditions of the **WAIT**. Thus, if a player posts a message satisfying these conditions in round $n_2 - 1$, then in round n_2 it will be posted to the message board and all honest server players will return from **WAIT**. If player i does not post a satisfying message within **WAIT_THRESHOLD** rounds, all players still waiting will post a blame of player i. Since every player's wait counter is at least 0 in round n_1, by $n_1 +$ **WAIT_THRESHOLD** they will have all blamed player i. □

Notice here that we haven't restricted to whether the satisfying message must be posted by a corrupted or honest player. It doesn't matter at this point, thanks to Propositions Proposition 15.5 and Proposition 16.5: honest server players return at the same time with the same values from the same **WAIT**. This will let us show in the next theorem that honest server players progress at almost exactly the same rate. That is, they are at most round out of sync with each other.

Theorem 16.6. *For any n, all honest server players that call* $\textbf{WAIT}_n$ *in a core accumulator function call it in the same round as each other, and complete it in the same round as each other.*

Proof. We prove by induction, which is well-defined by Proposition 16.3. Theorem 15.10 shows that all branching is the same, so the statement is true in the first round, since all honest server players will reach $\textbf{WAIT}_1$ in the same round.

Then suppose it is true up to the nth call to **WAIT**. By induction, all honest server players complete this call in the same round (say, k). There are two cases here: In the first case, they complete execution of all queued core accumulator functions. They will not call another **WAIT** until a new message is posted to the message board; however, since they see all the same messages in the same round, they will all start this **WAIT** in the same round.

The second case is that they return from **WAIT** and proceed immediately to the execution of another wait. Since all honest server players execute the same functions in the same order, they will all have the same **WAIT** in their function queue. Thus they will all start this call in the same round.

By Proposition 16.4 they will all post a satisfying message for their wait in that round and only that round. Thus, if the honest server players wait for a message from anohter honest server player, then none of them will complete that **WAIT**, since none of the messages will be posted until the next round. If they are waiting for a message from a corrupt player, then it doesn't matter when the corrupt player posts the same message, since we just showed that the honest players all start waiting for it in the same round and then Proposition 16.5 shows that they return from the **WAIT** in the same round. □

Corollary 16.7. *For any* **WAIT_THRESHOLD** ≥ 2*, honest server players will never blame each other during a* **WAIT**.

Proof. Following the logic in Theorem 16.6, if all honest server players start **WAIT** in round m, then they will all post a satisfying message in the same round. Thus, they will all have posted a satisfying message before the wait_counter variable reaches 2. □

This corollary means we can set **WAIT_THRESHOLD** $= 2$. This is short because we have pushed a lot of the synchronization into the structure of the public message board itself.

Proposition 16.8. *All honest server players will complete execution of any* **WAIT** *loop over server players within at most* $|\mathcal{C}|$**WAIT_THRESHOLD** $+ 1$ *rounds, or they will blame a corrupted player.*

Proof. By Theorem 16.6, all honest server players will enter the first **WAIT** in the same round as each other. Once all the honest server players have entered the loop, they will all have posted satisfying messages, by Proposition 16.4. Thus, all honest server players will finish all subsequent **WAIT** calls for messages from honest server players in the same round that they are called. For each corrupted player, the honest server players will enter the **WAIT** in the same round, so the corrupted player must post a satisfying message within **WAIT_THRESHOLD** rounds or all honest server players will blame them. Thus, each corrupted player adds at most **WAIT_THRESHOLD** rounds to the execution.

Adding up the maximum number of rounds gives the total number of rounds. □

From this point forward we define $T_{wait} = |\mathcal{C}|\mathbf{WAIT\_THRESHOLD} + 1$.

We could have a tighter bound by slightly modifying execution to have one wait threshold for all players in such a **WAIT** loop. We opted not to, to make the specification and proofs simpler.

17 Consistency proofs

17.1 Outline

We now reason that the adversarial server player must act like an honest server player, or else they will be blamed. This would be guaranteed by more complicated MPC protocols, but here we argue from the nature of the accumulator arithmetic.

Our blame procedure works by revealing all the secret shares used for the multi-party computation, re-computing the expected messages from each party, and blaming anyone whose messages deviated from this.

To show that this works, recall that in our protocol, all random shares and Beaver triples post commitments to the shares on the public message board before use. Since these are binding, it means that revealed secret shares must be genuine. We start by showing that honest server players can connect the commitments to the shares which were used, by showing that all honest server players will store the same commitments for the same ID of share or triple, Proposition 17.1. This allows us to argue in Proposition 17.2, that **CHECK_SHARED_VALUE**, which reveals a committed secret share and checks the commitments, will not blame any honest players.

Once the secret shares are revealed, the blame process works by stepping through the multi-party computation and comparing the expected output. This first involves checking that the accumulator parameters were generated correctly, and so Proposition 17.3 shows that **CHECK_GEN** does not blame honest players, mainly because it only checks the openings of shared values. Then Proposition 17.4 shows that once the shared secrets are revealed, the expected messages calculated in the blame will match what honest players post. Thus, no honest players will be blamed.

It remains to show that adversarial players *will* be blamed. To do that, we use the binding properties of the commitment to argue that if an adversary passes **CHECK_SHARED_VALUE** without being blamed, then for random shares the resulting value must be random (Proposition 17.5) and for Beaver triples the resulting value must be a valid Beaver triple (Proposition 17.6). This then shows that **GEN** will successfully pass the first loop and produce valid parameters, Proposition 17.7, and it will not blame any honest party (Proposition 17.8). By showing that an adversary must post values that are the same as what an honest user would post, we can show in Proposition 17.9 that the only way for the affine inversion functions to enter the blame branch is if the adversary caused the multi-party computation to fail, and in this case the honest players can find unexpected messages from the adversary and successfully blame

them.

17.2 Proofs

Proposition 17.1. *If two honest server players i and j have values y_i and y_j (respectively) in* **shared_triples** *or* **shared_random**, *if* $y_i.\mathsf{id} = y_j.\mathsf{id}$, *then*

$$y_i.\mathsf{commit} = y_j.\mathsf{commit}.$$

Proof. The control program only posts shared triple values with unique IDs. Thus, honest server players will only have values with the same ID if they came from the same message, which means they found the same value for commit on the public message board. Other players post commitments to randomness publicly, and by Proposition 15.5 the honest players will store commitments from the same messages. Since honest players only post messages with a label of "random" during **GET_SHARED_RANDOM** and they increment **random_share_counter** every time, honest players only post one such message, and thus the commitment a player posts, which other players see, will match what they store internally. □

Proposition 17.2. *During execution of* **CHECK_SHARED_VALUE**, *no honest server player will blame another honest server player, and all honest server players will return with all the shares from other honest server players.*

Proof. We know that the honest server players will not blame each other during the **WAIT**; further, this guarantees that they execute the same **WAIT** and execute **CHECK_SHARED_VALUE** with a share with the same id.

Every honest server player posts a message which contains their share of the secret y, proving the second part of the statements. Since this share corresponds one-to-one with the commitments posted to the public message board (Proposition 17.1), then $y.\mathsf{commit}[j]$ will commit to the share posted by any honest server player j. Thus, they will not post a blame message for that honest server player. □

Proposition 17.3. *During execution of* **CHECK_GEN**, *no honest server player will blame another honest server player. If no blame occurs, all honest server players return with* $\mathsf{shares}_\alpha, \mathsf{shares}_s$ *that match the shares of* α *and* s_m *belonging to all other honest server players.*

Proof. Recall that honest players form $\tilde{Q}$ and $\tilde{Q}_m$ by multiplying their shares of α and s_m by $\tilde{P}$ and $\tilde{K}$ respectively, then opening the resulting shared elliptic curve point. Honest server player save the points output during this opening in $\tilde{Q}_{com}$ and $\tilde{Q}_{m,com}$.

During **CHECK_GEN**, the honest server players will execute **CHECK_SHARED_VALUE**. Since the shared random values α and s_m from **GEN** are obtained in a fixed order, they will have the same ID, and no honest server player will blame another. Further, they will return with the the same shares by Proposition 17.2.

This means that for any honest server player i, if j is the index of any other honest server player, then during i's execution, $\mathsf{shares}_\alpha[j]$ equals the value of $\alpha.\mathsf{share}$ that j used, proving the second part of the statement. This means $\mathsf{shares}_\alpha[j] \cdot \tilde{P}$ will be precisely the value that player j output during **OPEN**($[\tilde{Q}]$), which means that will be the value that player i has in $\tilde{Q}_{com}$. Thus, the first check will pass.

The same logic applies with the second check for $\tilde{Q}_{m,com}$. □

Proposition 17.4. *During execution of* **DEL** *or* **ADD**, *no honest server player will blame another honest player.*

Proof. Our proof will argue over each of the lines where blame messages could be posted: in the first check to see if the accumulator exists when the function was called, during a **WAIT**, during the check in **ADD** for messages with a "witness" label, and during the blame branches of **AFF_INV_ACC** and **AFF_INV_SIGN**.

First, only users post Add messages and, thanks to Line 5 of **WIT**, they wait until the first accumulator is posted before doing so.

Next, Proposition 16.8 shows that honest server players will not blame each other during **WAIT**.

Then we argue that the loop at Line 17 in **ADD**, checking for messages with a "witness" label, will not blame an honest server player. All honest server players post a message matching the criteria of this loop in Line 17 before executing **AFF_INV_ACC** in Line 19. We know that **AFF_INV_ACC** will take at least one round to complete, since all honest server players must wait for a message from other honest server players. Thus, the "witness" messages will be posted and added to **data_set** by the time this finishes, which finishes before executing the loop at Line 17. Thus, the required messages will be posted from all honest server players during this loop.

Finally we argue about the blames posted from **AFF_INV_ACC**, noting that the logic is identical for **AFF_INV_SIGN**. For this we argue line-by-line through the blame branch, using previous propositions to show that each line does not blame an honest player.

The statement is certainly true if no honest server players take the blame branch at Line 15 in **AFF_INV_ACC**. Thus, assume $e(V', y\tilde{P} + \tilde{Q}) \neq e(V, \tilde{P})$ so that the honest players enter the blame branch. All honest server players will notice this and begin the blame branch, starting by posting $\perp$ as the new accumulator in Line 19. Since all honest server players do this, they will not blame each other in this step.

Proposition 17.3 shows that the honest server players do not blame each other during **CHECK_GEN**(), and they return with the shares of α and s. This means in Line 24, adding y to $\mathsf{shares}_\alpha[1]$ will mean that $\mathsf{shares}_\alpha[i] = [y+\alpha]_i$ (with the latter value defined in the local execution for player i). We then step through **CHECK_INVERSE**.

As the shared values r, a, b, and c are all called with the same ID, they pass **CHECK_SHARED_VALUE** and reveal all the shares. Since all these shares

are the same for honest players, each honest server player will have the same value for $\mathsf{shares}_{\epsilon,\delta}[j]$ if j is the index of another honest server player.

When they call the first **CHECK_OPEN** at Line 7, the ID will correspond to the first **OPEN** that was called during **INVERT** at Line 4. Stepping through the logic of **INVERT** and **CHECK_INVERSE**, we see that shares_ϵ will precisely match the values of $[\epsilon]$ that were opened. Thus, during **CHECK_OPEN**, the value x obtained from the commitments will precisely match the value it is checked against. We can see that no honest server player will post more than one message with label open_commit or open_reveal with the same ID.

The same logic applies as we step through the rest of **CHECK_INVERSE**. Since all honest server players are comparing the same values, they will all blame the same player j if they blame any player at all, and thus they will either all exit or none will exit. Thus, they will all return from **CHECK_INVERSE** with the same values of shares_w without blaming each other, and with shares_w matching the local values of $[w]$.

From there, we see that $\mathsf{shares}_v[j]$ match the local values of $[V']$. Thus, when the honest server players call the final **CHECK_OPEN**, they will also not blame each other. □

We must next show that the blame branch must blame someone, and since we just showed it cannot be an honest player, it must blame a corrupt player. To do this, we need two propositions showing notions of soundness of the shared random values and Beaver triples.

Proposition 17.5. *If an honest server player returns with r from* **CHECK_SHARED_VALUE**(X) *without posting a blame, where X is a single random shared value, then with probability at least $1 - O(\frac{T^2}{2^{2\lambda}})$, the shares in r add up to a uniformly random value.*

Proof. We first consider when the commitments to X are first posted in Line 3 of **GET_SHARED_RANDOM**. Let $c_1, \ldots, c_N$ be the commitments. For each c_i posted by a corrupted player, let (r_i, o_i) be an input to the random oracle that produces this output. The probability that there are two such queries matching this output is $O(\frac{T^2}{2^{2\lambda}})$, so we ignore this case. If there has been no input producing this output yet, choose a random (r_i, o_i) and assign the output of the random oracle on this string to be this commitment (with probability $\frac{1}{2^{4\lambda}}$, this string has not been queried and so this is indistinguishable to the adversary). In this way we have effectively defined a unique "share" r_i corresponding to each commitment.

Notice that for each honest player, we could choose a different opening and share (r_i, o_i) and reprogram the random oracle to still map this to c_i, making this indistinguishable to the adversary unless it has queried (r_i, o_i), which only occurs with probability $\frac{TN}{2^{2\lambda}}$. This means that the sum of the shares we define in this way is uniformly random, except with probability $O(\frac{T^2}{2^{2\lambda}})$.

Now we consider that during **CHECK_SHARED_VALUE**, for a corrupted player to not be blamed, each corrupted player i must post a satisfying

message at Line 7 which opens commitment i. The probability of finding an opening which is not the original query is at most $\frac{T}{2^{2\lambda}}$, so we can again assume this does not happen and the adversary must post the opening (r_i, o_i). This means the honest server players will return with the values $r_1, \ldots, r_N$ defined earlier in this proof, which are uniformly random. □

Proposition 17.6. *If an honest server player returns with x from* **CHECK_SHARED_VALUE**(X) *without posting a blame, where X is one shared value from a Beaver Triple, then x matches the shares posted during* **GET_SHARED_TRIPLE** *with probability at least $1 - \frac{T}{2^{2\lambda}}$ (where T is the total number of random oracle queries).*

Proof. Direct inspection shows that honest server players will post a valid opening of their share of a Beaver Triple, which will be recovered by other players in the **WAIT** at Line 7. If a corrupted player posts a satisfying message and is not blamed, it must be an opening matching the control program's commitment (as computed on Line 22 of **BRACKET**), since the honest server player saved that commitment. Matching an existing commitment has probability at most $\frac{T}{2^{2\lambda}}$. □

With these two propositions, we can prove our two main results of this section: either a corrupted player gets blamed, or **GEN** efficiently generates the parameters of the scheme and **AFF_INV_ACC** and **AFF_INV_SIGN** correctly compute the required values.

Proposition 17.7. *With probability at least $1 - O(\frac{T^2}{2^{2\lambda}})$, the loop to generate parameters at Line 8 in* **GEN** *completes within $O(1)$ repetitions, or blames a corrupted server player.*

Proof. The loop completes in one iteration unless one of $\tilde{Q}$ or $\tilde{Q}_m$ are the identity point. If they are, the checks at Line 26 ensure that the output of **CHECK_SHARED_VALUE** matches the components of $\tilde{Q}$ and $\tilde{Q}_m$. By Proposition 17.5, this output is uniformly random (or a corrupted server player is blamed). This means this condition of the parameters being identity points occurs with probability at most $2/q$, so only $O(1)$ repetitions are needed to find non-identity values. □

Proposition 17.8. *All honest server players that reach the end of execution of* **CHECK_GEN**(y) *will have the same set* blames. *If two honest server players return from* **CHECK_GEN** *with a set of secret shares, they will return with the same set.*

Proof. All honest server players will return the same shares from **CHECK_SHARED_VALUE**, if they return, by Proposition 15.6. Similarly, the shares that go into **gen_commit** are posted publicly, so all honest server players will obtain the same values. This means they all have the same values for the loop over servers to check shares at Line 8 in **CHECK_GEN**, so they will blame the same players, if any. □

Proposition 17.9. *If an honest server player enters the blame branch of* AFF_INV_ACC *or* AFF_INV_SIGN*, then with probability at least* $1-O(\frac{T^2}{2^{2\lambda}})$ *it will post a blame towards some player (where* T *is the total number of random oracle queries of the adversary).*

Proof. We prove by contradiction: assume that an honest player enters and completes the blame branch and also posts no blame, which should never happen. Our proof mainly relies on the last two propositions showing that checking shared values outputs the shared values we expect or blames a corrupted player.

We thus step through execution of the blame branch. First, during CHECK_GEN, no blame implies that $\mathsf{shares}_\alpha[j] \cdot \tilde{P} = \tilde{Q}_{com}[j]$ for all j, so $\sum_j \mathsf{shares}_\alpha[j]\tilde{P} = \sum_j \tilde{Q}_{com}[j]$. During GEN, honest server players computed $\tilde{Q} = \sum_j \tilde{Q}_{com}[j]$. Therefore, we can define $\alpha := \sum_j \mathsf{shares}_\alpha[j]$ and conclude that $\tilde{Q} = \alpha\tilde{P}$. After returning from CHECK_GEN, we conclude that shares_α add up to $y + \alpha$.

We then proceed through CHECK_INVERSE. Since CHECK_SHARED_VALUE posted no blame, Proposition 17.6 shows that it returns shares of a, b, and c which form a valid Beaver triple (since the control program posted them) with probability at least $1 - \frac{3T}{2^{2\lambda}}$. Checking r, CHECK_SHARED_VALUE returns some shares shares_r, which we do not yet guarantee match the r from INVERT, but as Proposition 17.5 shows, they add up to a uniformly random value r' except with probability $O(\frac{T^2}{2^{2\lambda}})$.

For CHECK_OPEN at Line 7 not to blame any player, the values posted in Line 4 of INVERT must be valid shares of $\epsilon = x - a$; similarly for $\delta = r' - b$. Then CHECK_OPEN at Line 12 implies that the value in INVERT must be $c + \epsilon \cdot r' + \delta \cdot x + \epsilon\delta$, which must equal $x \cdot r$ since a, b, c was a valid Beaver triple. At this point, $z^{-1}r' = x^{-1} \mod q$, unless $z \equiv 0 \mod q$. Since r' is uniformly random, this occurs only with probability $\frac{1}{q}$, so we ignore this case. Here $x = y + \alpha$.

This means shares_w returned from CHECK_INVERSE in Line 25 of CHECK_GEN will add up to $((y + \alpha)r')^{-1}r' = (y + \alpha)^{-1}$. Then shares_v will add up to $(y + \alpha)^{-1}V$. Since CHECK_OPEN compares this to the shares which opened to V', and blamed no player, this means the shares of V' match shares_v, so $V' = (y + \alpha)^{-1}V$.

In turn, this means that $e(V', y\tilde{P} + \tilde{Q}) = e((y+\alpha)^{-1}V, y\tilde{P} + \alpha\tilde{P}) = e(V, \tilde{P})$. Thus, the conditional statement starting this blame branch will not execute, contradicting the hypothesis. □

18 Accumulator Structure Proofs

18.1 Outline

Our next goal is to prove that the accumulator truly encodes the set of accumulated elements in some sense. Roughly speaking, we will simply show that when all server players act like honest server players (as Section 17 shows they must), the output matches the single-server case.

More specifically, we first show in Proposition 18.1 that the internal auxiliary data of an honest server player matches the observer's record of the accumulated set. By our previous results on the consistency of computations, we can show in Proposition 18.2 that honest players will compute the same value of the accumulator in each epoch (whether or not it is right), meaning that corrupt players must also post the same value or be blamed (Proposition 18.4). Then we argue based on the actual structure of the multi-party computation, i.e., that it performs a finite field inversion, that the inversions computed in **ADD** are correct. Since these are stored as witnesses for each player, these stored witnesses are correct (Lemma 18.6). Together this shows that **ADD** and **DEL** compute that they are supposed to, so Proposition 18.7 shows that the structure of the accumulator matches the single server case.

18.2 Proofs

Proposition 18.1. *For an honest server player at the end of execution of* **ADD** *or* **DEL**, *the set in* **aux** *is exactly the same as* $\mathcal{S}$[**acc_counter**].

Proof. We prove by induction. This is true when the accumulator starts, as both sets are empty.

Suppose this is true up to **acc_counter** $= n$, meaning that at the end of the execution of either **ADD** or **DEL** which modified **acc_counter** (the only functions to do so), $\mathcal{S}$[**acc_counter**] $=$ **aux**. Since **ADD** and **DEL** are executed in the order that they are posted, and the control function adds to $\mathcal{S}$ in the same order that **ADD** and **DEL** are posted, then $\mathcal{S}$[**acc_counter** $+ 1$] $=$ **aux** $\cup \{y\}$ if the next function call is **ADD**(y) (the same logic will hold for **DEL**). At Line 13 of this call to **ADD**, **aux** will get modified to **aux** $\cup \{y\}$, and then either the execution fails but blames a corrupted server player (in which case the honest server player never reaches the end of execution), or it does reach the end of execution and increments **acc_epoch**, proving the inductive case of the statement. □

Proposition 18.2. *All messages of type* ACC *from honest server players will have the same value of* m.accumulator *for the same value of* m.epoch. *Further, if an honest server player sends a message of type* ACC *with a value* m.epoch $= n$, *then the next message of type* ACC *from that player will have* m.epoch $= n + 1$.

Proof. The only time an honest server player posts a message of type ACC is during **POST_ACCUMULATOR**, and this is only called from **GEN**, **ADD**, and **DEL**.

All honest server players start with **acc_epoch** $= -1$, and they only update it to 0 during a call to **GEN** (they blame the adversary if a call to **ADD** or **DEL** comes earlier). During the first call to **GEN**, all honest server players will obtain the same value for the accumulator, since it is derived by calls to **OPEN**.

Since all core accumulator functions execute in the same order for all honest server players, they will execute the same **ADD** and **DEL** in the same order. During each, they increment **acc_epoch** exactly once before calling **POST_ACCUMULATOR**, when they post a message of type ACC. This proves the second part of the proposition.

For the first, we proceed by induction on **acc_epoch**. We already showed that the statement holds during **GEN**. If the next call to **POST_ACCUMULATOR** comes from a call to **ADD**, all honest server players will post the same value of the accumulator because it does not change in **ADD**. If it comes from a call to **DEL**, the only part that changes is the first value V, which is the return value of a call to **OPEN**, which all honest server players will obtain the same value for by Proposition 15.6. □

Proposition 18.3. *No honest server player will blame another honest server player during execution of* **POST_ACCUMULATOR**.

Proof. Since honest server players will call the same **WAIT** in the same round as each other (Theorem 16.6) and post the required values before doing so, no honest server will be blamed for failing to post an accumulator. The required condition to not be blamed by honest server player i during **POST_ACCUMULATOR**, at Line 5, is that the value of m.accumulator is not equal to the value that i obtains, but all honest server players will post the same value by Proposition 18.2. □

Proposition 18.4. *If a corrupted server player does not post the same value of accumulator as all honest server players post when they begin* **POST_ACCUMULATOR**, *the adversary will be blamed and the accumulator will abort.*

Proof. Once the honest server players begin **POST_ACCUMULATOR**, they will not blame each other by Proposition 18.3, and by Proposition 16.8 they will progress between each **WAIT** in a bounded time, so they will eventually wait for a value from a corrupted server player. Since they execute this loop in the same order, all honest server players will wait for a message from the same corrupted server player in the same round. If the corrupted server player posts nothing they will be blamed during the **WAIT**, and if they post a non-matching value, they will be blamed by all honest server players, which form a majority. □

Recall that $\mathcal{A}$, a variable of the observer program, is structured so that $\mathcal{A}[i][j]$ is the ith accumulator that has been posted, as posted by player j.

Corollary 18.5. *If the observer program's values for the accumulator in $\mathcal{A}$ at index n contains more than $|\mathcal{C}|$ values, then either they will be $|$**servers**$|$ repetitions of the same value, or within $2T_{wait}$ rounds the accumulator will abort and blame a corrupted player.*

Proof. The observer program only modifies $\mathcal{A}$ at Line 9 of **OBSERVER_UPDATE_IDEAL**. $\mathcal{A}$ is a two-dimensional array, indexed first by the accumulator epoch, and then by the player ID of server players. It only modifies each epoch once for each server player. This means that a single player (corrupt or honest) can only modify one value in $\mathcal{A}[n]$ once, for each n. Thus, for a specific epoch n, the adversary only controls at most $|\mathcal{C}|$ values.

The observer program only adds a new array to $\mathcal{A}$ when a message of type ACC is posted. While corrupt server players could post such a message at any time, this could produce at most $|\mathcal{C}|$ values. To find more values posted, an honest server player must post a value, which they only do as part of **POST_ACCUMULATOR**. However, if they start this, then by Proposition 18.4, the adversary's accumulator value must match the honest players or they will be blamed. By Proposition 16.8 the corrupted players must post a value within $2T_{wait}$ rounds, and they will be blamed if they do not match the same value as the honest players. □

We use **wits** to assist in computations for **DEL**, and we show the necessary properties for this.

Lemma 18.6. *For all honest players at the end of any round, either* $\mathbf{wits}[y] = \frac{1}{y+\alpha}V$*, where* $\mathbf{acc} = (V, \tilde{Q}, \tilde{Q}_m)$ *and* $\tilde{Q} = \alpha\tilde{P}$ *or an honest player posts a* BLAME *message.*

Proof. Only **GEN**, **ADD**, and **DEL** modify **wits**. The statement holds vacuously during **GEN**. We prove for the others by induction on the calls to add or delete.

During **ADD**, when honest server players call **AFF_INV_ACC**(y), if they do not follow the blame branch, then that means the output V' satisfies $e(V', y\tilde{P} + \tilde{Q}) = e(V, \tilde{P})$ which means $V' = \frac{1}{y+\alpha}V$ by non-degeneracy of pairings. Thus, when V' is added to **wits**, the statement holds for that value y. Since the accumulator value doesn't change during **ADD**, the statement still holds for all other values y'.

We then consider calls to **DEL**(y). At Line 11 it sets $V' \leftarrow \frac{1}{y+\alpha}V$, and this will become the new accumulator value. Since y is removed from **wits**, we can ignore it. Then, for all other $y' \in \mathbf{wits}$, Line 14 in **DEL** sets $C' = \frac{1}{y-y'}(C - V')$, where we let $C := \mathbf{wits}[y']$ (i.e., before it's modified). By induction, $C = \frac{1}{y'+\alpha}V$. This tells us that

$$C' = \frac{1}{y-y'}\left(\frac{1}{y'+\alpha}V - \frac{1}{y+\alpha}V\right) = \frac{1}{y'+\alpha}V'$$

Thus, the property holds for all values in **wits**. □

For the next Proposition, we define $\alpha \in \mathbb{F}_q$ such that $\tilde{Q} = \alpha\tilde{P}$.

Proposition 18.7. *Let* $y_1, y_2, \ldots$ *be the sequence of arguments* y *to* **ADD** *and* **DEL***, ordered as the calls to* **ADD** *and* **DEL** *were ordered on the message board. Then, in all rounds, for* $i > 0$*, if* $|\mathcal{A}[i]| > |\mathcal{C}|$ *and no server players have been blamed, then if the* i*th call (starting at* 1*) to* **ADD** *or* **DEL** *was a call to:*

- **ADD**, *then* $\mathcal{A}[i][0] = \mathcal{A}[i-1][0]$
- **DEL**, *then letting* $(V, \tilde{Q}, \tilde{Q}_m) = \mathcal{A}[i-1][0]$, *we have that either:*
  - $\mathcal{A}[i][0] = (\frac{1}{y_i+\alpha}V, \tilde{Q}, \tilde{Q}_m)$ *if* $y_i \in \mathcal{S}[i-1]$
  - $\mathcal{A}[i][0] = \mathcal{A}[i-1][0]$ *if* $y_i \notin \mathcal{S}[i-1]$

Proof. By Corollary 18.5, the value of $\mathcal{A}[i]$ is consistent and matches the value posted by honest server players if the condition on its size is met and no player is blamed. Thus we can consider the execution of honest server players; they execute **ADD** and **DEL** in the same order as these functions are posted on the message board, and these are the only functions that cause them to post messages of type ACC. If they call **ADD**, then they do not change the accumulator value at all.

If they call **DEL**(y), then since **aux** $= \mathcal{S}[i-1]$ by Proposition 18.1 then the new accumulator V' is either the same as the old one (if $y \in$ **aux**) or the witness for player y in **wits**, which by Lemma 18.6 will equal $\frac{1}{y+\alpha}V$, and the statement holds. □

19 Witness Proofs

19.1 Outline

A witness for this accumulator is defined as a tuple $(i, x, C, R_m) \in \mathbb{N} \times \mathbb{F}_q \times G_1 \times G_1$, consisting of an epoch number i, a long-term secret x, a short-term signature C (which is updated as the accumulator changes) and a long-term signature R_m. We say such a witness is valid if there is a consensus accumulator posted for accumulator round i, and the witness passes the pairing checks for that accumulator.

Our goal in this section is to prove that honest user players obtain valid witnesses when they are initialized. Our protocol has the servers compute a witness for a user when the user is added to the accumulator, and once they do, they post a consensus accumulator for that epoch. Thus, as in Proposition 19.1, once a consensus accumualtor is posted for the addition of y, the witness messages for y are posted. A user can then call **FIND_WITNESS** and recover this witness (Proposition 19.2), and this requires no waiting if the consensus accumulator is already posted.

A user also needs the long-term signature R_m. This is computed the first time that a user sends an Add message with a proof argument, so we show that honest users will send valid proofs and thus trigger the signature computation in Proposition 19.3. This shows that once a user posts such an Add message, it can eventually find both components of its witness. Since **WIT** posts such a message, Proposition 19.4 shows that it provides a valid initial witness to a user.

19.2 Proofs

The first proposition states that once the server players post the messages in Add, then they will post the data necessary for a user to reconstruct their witness.

Proposition 19.1. *Let n be such that there is a consensus accumulator posted for epoch n and $y \in \mathcal{S}[n] \setminus \mathcal{S}[n-1]$. Then the public message board contains a unique message from each server of the form:*

- (“witness”, y, i, n) *for some i*
- (open_commit, $i+3, c$) *for the same i and some opening c, from each server player*
- (open_reveal, $i+3, o$) *for the same i and some opening o, from each server player.*

Proof. If an honest server player calls **ADD**, then it only posts a new accumulator (say, $(V, \tilde{Q}, \tilde{Q}_m)$) at Line 28. Thus, if a consensus accumulator exists for epoch n, then the honest server players finished execution of the rest of **ADD**. By Proposition 18.1, the accumulator must have been posted as a resut of **ADD**(y), and during execution, $y \notin$ **aux** during execution of Add(y).

This means all honest server player will enter the IF block at Line 12. In this block, at Line 13 the server players will post the messages of the form (“witness”, y, **open_number**, **acc_epoch**). At Line 19, the honest server players will return with a value of V' which satisfies $e(V', y\tilde{P} + \tilde{Q}) = e(V, \tilde{P})$ by the same logic as Lemma 18.6. This means that the value V' was opened, so the commitments and openings which construct V', as produced during **OPEN**, are present on the message board before the honest server player posts the new accumulator.

Counting the calls to **OPEN** during **AFF_INV_ACC** shows that the values of i are correct.

Because the open numbers and accumulator epochs increment with each posting, each server only posts one message of each of these types. □

Proposition 19.2. *If an honest user player calls* **FIND_WITNESS** *such that a consensus accumulator is posted for the argument of* max_epoch*, then the user will return in the same round. If* min_epoch $\leq n \leq$ max_epoch *is the maximum epoch such that $y \in \mathcal{S}[n] \setminus \mathcal{S}[n-1]$, then the honest user returns with a value C and the epoch number n, such that for the nth accumulator $(V, \tilde{Q}, \tilde{Q}_m)$,*

$$e(C, y\tilde{P} + \tilde{Q}) = e(V, \tilde{P}).$$

If no such n exists, the user will return $\perp$.

Proof. By Proposition 19.1, such an n exists if and only if the message required at Line 2 of **FIND_WITNESS** contains a unique satisfying message. Thus, the user will immediately return from the calls to **WAIT** at Lines 10 and Line 10.

The values it returns will correspond to shares of C that satisfying the pairing equation by Proposition 19.1.

Then, if no such n exists, the user returns $\perp$. □

Proposition 19.3. *If an honest user is the first player to send an Add message for some argument y, then when the honest server players execute that* **ADD**, *they will call* **AFF_INV_SIGN** *from within* **SIGN**.

Proof. By inspection of **ADD**, honest server players will execute **SIGN** precisely on the first call to **ADD** for each ID y. Honest users only post **ADD** messages as part of **WIT**. Comparing the proof they construct to the check in **SIGN** gives the result. □

Proposition 19.4. *Suppose a player with user id y starts* **WIT**, *and suppose they post the first message of* Add *with argument y. Let m be the first round after a consensus accumulator is posted with $y \in \mathcal{S}$. Then by round m at the latest, that player will have returned with a valid witness, or the accumulator will exit by blaming a corrupted player.*

Proof. During **WIT**, the user will post an Add message with data (y, proof). This is the only time a user will post an Add message (no other function requests this, and thanks to the check at Line 1, users only execute this once). Hence, if they post the first Add message with y, then when the honest server programs execute this **ADD**, $y \notin$ **Y**. Then the servers will call **SIGN**. By Proposition 19.3, the honest server players will call **AFF_INV_SIGN**. If they do not blame a corrupted player, then by Proposition 17.9 they did not enter the blame branch, so the value R_m that they post will match what the user requires.

Similarly, during this execution of **ADD**, $y \notin$ **aux** for any honest server player, so they will enter **AFF_INV_ACC** and (if they do not blame) will post V' such that $e(V', y\tilde{P} + \tilde{Q}) = e(V, \tilde{P})$. where V is the accumulator in the round m that they post in the message $(\text{“witness”}, y, n, m)$.

Once this is posted, the honest servers will then post the accumulator value. That means that if m is the first round that a consensus accumulator is posted (meaning all server players have posted an accumulator value) then all players completed **AFF_INV_ACC**, so the value V' was posted to the message board by then.

By inspection, the messages that **FIND_WITNESS** requires to return any witness are precisely those posted during **ADD**(y). Since a $(\text{“witness”}, y, n, m)$ message was already posted, **FIND_WITNESS** will wait until the required data is posted, which must be done before the consensus accumulator. □

20 Update Correctness Proofs

20.1 Outline

In this section we show that updates run correctly and blame adversarial players as needed. The main computations of an update are the batch update polynomials of [VB20], though done obliviously with Shamir secret shares [Sha79], though the full update protocol is more complicated to handle pathological cases, such as updating over epochs where a user is repeatedly removed and re-added to the accumulator.

We start with Proposition 20.1, showing that the batch update polynomials with the secret sharing work correctly when honest server players respond. To handle corrupt players, recall that the batch update involves server players evaluating a polynomial based entirely on public data (previous accumulators and the deleted IDs) on the shares the user provides. Thus, if the returned value is incorrect, a user can reveal their shares and any player can replicate exactly the response each server should have provided to determine which server misbehaved and should be blamed. Honest server players will not be blamed, by Proposition 20.2. Since we assume an honest majority, a simple argument in Proposition 20.4 based on polynomial interpolation shows that corrupt players must match the honest behaviour or be blamed, and so the the subroutine which evaluates the update polynomial (**GET_UPDATE**) will succeed. Since it is only two rounds of communication, and a user only needs the honest players to respond (they can reconstruct the result from only a portion of the shares, since they use Shamir shares), **GET_UPDATE** always takes exactly 2 rounds (Proposition 20.3).

Next we consider the wrapper function **RETURN_UPDATE**, which handles the edge cases. If a user is deleted, one of the update polynomials is 0 and this function will check for a new witness posted to the public message board. If it finds one, it must recurse to update this new witness to the latest epoch; however, it only recurses once (Proposition 20.6) because it always chooses the latest new witness. Since only **GET_UPDATE** needs to wait for other players, we see in Proposition 20.8 that **RETURN_UPDATE** will take at most 4 rounds.

Another edge cases is if a user is requested to update multiple times. Multiple requests in the same round are ignored (only the latest epoch is used), but since each update takes several rounds, a previous update may be in progress when a new one is requested. Since the update polynomials can be evaluated without a valid witness, a user can request the update polynomials be evaluated for a future update, wait for a a previous update to finish, then apply the update polynomials to the newly-updated witness locally. Thus, Proposition 20.10 shows that **RETURN_UPDATE** takes at most 4 rounds when called as part of **UPDATE**, and Corollary 20.11 extends this to when it is called in **PROVE**. This gives us all the update progress results we need.

For correctness, in Proposition 20.12 we argue that since the update polynomials work, and a user will find a new witness on the message board if it exists,

that if a user updates to an epoch for which they are in the accumulated set, they will return with a valid witness. Since user witnesses are always the output of updates (except their first witness, which is valid by Proposition 19.4), Corollary 20.13 can easily show that user witnesses for epochs where the user is accumulated will always be valid.

Conversely, we must also show that if a user is *not* in the accumulated set, their witness should be invalid. Proposition 20.14 shows this by exhaustively checking subcases, and similarly in Proposition 20.15 we show that if a user has a witness for an accumulator they are not part of, their witness is invalid.

Together, the main result of this section is Proposition 20.16, showing that verification works correctly. The only remaining step for Proposition 20.16 is arguing that the elliptic curve arithmetic of the zero-knowledge proofs is correct.

20.2 Proofs

Our first Proposition is essentially the proof that the batch updates of [VB20] work correctly.

Proposition 20.1. *If an honest user with an ID of y_u begins* **GET_UPDATE**(old_epoch, new_epoch, $*$) *starting with a valid witness* (old_epoch, x, C, R_m) *such that y_u was not deleted from the accumulator between epoch* old_epoch *and* new_epoch*, and the user obtains all of its values of Ws and Ds from honest server players, then it will exit with $\mathcal{D}$ and $\mathcal{W}$ such that no element ot $\mathcal{D}$ is* 0 *and such that* **PROCESS_UPDATE**$(\mathcal{W}, \mathcal{D}, C)$ *is a valid witness for* new_epoch*.*

Proof. We start by considering what the honest servers will send, via **UPD_HELP**. The elements $y_1, \ldots, y_d$ are numbered so that y_1 was deleted first; thus, $y_1, \ldots, y_k$ are the first k elements deleted. Similarly, the accumulator values are labelled $V_1, \ldots, V_d$. Ultimately, **UPD_HELP** returns tuples of (D, W) which are the necessary components for a batch update of k elements, so applying this repeatedly will give the full update.

How we will show this is to show that the first two elements D and W of $\mathcal{D}$ and $\mathcal{W}$ (respectively) will make $D^{-1}(C - W)$ a valid witness for the accumulator V_k. Repeating this argument shows that after the next iteration, the user has a valid witness for V_{2k}, until finally the user has a valid witness for V_d.

The polynomial $d(x)$ computed in Line 5 of **UPD_HELP** has $y_1, \ldots, y_k$ elements as its roots. Similar, $w_s(x)$ (Line 6) has the first $s-1$ of these elements as its roots.

The shares that the user sends to each server are such that $y\_\mathsf{shares}[j]$ are shares of y_u^j. This means for all ℓ, we have for the first value returned, (using $[[\cdot]]$ to denote a share of some value):

$$[[D]] = \sum_{j=0}^{k} d[j][[y_u^j]] = [[d(y_u)]] \mod q$$

This means when the user retrieves all their responses from honest server players, they can reconstruct $d(y_u)$ as the first element of $\mathcal{D}$, since the secret-sharing scheme is affine. Here $d(y_u) = 0$ if and only if y_u is a root, meaning $y_u \in \{y_1, \ldots, y_k\}$ contradicting our assumption that y_u was not deleted.

Similarly, we have for all ℓ, the shares computed in **UPD_HELP** are

$$\begin{aligned}[[W[\ell]]] &= \sum_{s=0}^{k-1} \left(\sum_{j=0}^{s} v_s[j][[y_u^j]] \right) V_{k-s} \\ &= \sum_{s=0}^{k-1} V_{k-s}[[v_s(y_u)]]\end{aligned}$$

Then the user reconstructs the first element of $\mathcal{W}$ as $\sum_{s=0}^{k-1} V_{k-s} v_s(y_u)$. This means

$$\begin{aligned}W &= \sum_{s=0}^{k-1} V_{k-s} v_s(y_u) \\ &= \sum_{s=0}^{k-1} \underbrace{V_{k-s} \prod_{t=1}^{s} (y_t - y_u)}_{:= U_{k-s}}\end{aligned}$$

We notice that since $V_{s+1} = (y_{s+1} + \alpha)^{-1} V_s$, we have that $U_s = \frac{y_{s+1}+\alpha}{y_s - y_u} U_{s+1}$. Further, $\frac{1}{y_u+\alpha} U_1 = \frac{1}{y_1+\alpha} C$. This means

$$\begin{aligned}C - W =& C - U_1 - U_2 - \cdots - U_k \\ =& \frac{y_1 - y_u}{y_u + \alpha} U_1 - U_2 - \cdots - U_k \\ =& \frac{(y_1 - y_u)(y_2 - y_u)}{y_u + \alpha} U_2 - \cdots - U_k \\ &\vdots \\ =& \frac{d(y_u)}{y_u + \alpha} U_d\end{aligned}$$

Thus, $\frac{1}{d(y_u)}(C - W) = \frac{1}{y_u+\alpha} U_k$, and since $U_k = V_k$ (the accumulator value after k updates), the update succeeds. □

Honest users will only post **BLAME** if the update goes wrong, and this prompts the honest server players to check the required messages. We show that this blame works correctly.

Proposition 20.2. *An honest server player will only be blamed from the* **BLAME** *function with probability $O(\frac{T^2}{2^{2\lambda}})$, where T is the total number of queries to the random oracle.*

Proof. Since we have an honest majority, this can be shown by proving that no honest server player will blame another honest server player.

Suppose an honest server begins **BLAME**, with arguments of sender and id such that id is the ID of an honest server player. If there is no set of messages posted that form blames, the player will return immediately.

Suppose then that blames has a non-$\perp$ entry. This means there was some message from sender with k elements of $\mathbb{F}_q$, which we will denote as $y = y_0, \ldots, y_{k-1}$. It also means there is a response from id with a label of wit, values of W and D, and a string h matching the output of the random oracle on $(y, \mathsf{upd\_end}, \mathsf{upd\_start})$.

The only point at which an honest server player will post such a message is during $\mathbf{UPD_S}$ at Line 16. This means if player id (the server player) is honest, they computed h as $\mathbf{RANDOM\_ORACLE}(y, \mathsf{upd\_end}, \mathsf{upd\_start})$. As we only consider messages posted by an honest player, the value h will always be an output of the random oracle. There is some chance that another message matches this value (say, the adversary finds an ID y that creates a collision) but the probability of this is only $O(\frac{T^2}{2^{2\lambda}})$. Thus, the values of W and D that id posted are exactly the output of **UPD_HELP** with the arguments V, Y, y, d, k, m, and t as computed in **BLAME** (since these are computed in exactly the same way as $\mathbf{UPD_S}$ computes them). This means the final check of W_{check} and D_{check} against W and D will pass, and id will not be blamed. □

We then show a basic progress result, since it will be convenient to argue timing at the same time as correctness.

Proposition 20.3. *An honest user executing* **GET_UPDATE**(m, n) *will exit from the loop at Line 25 with at least t non-$\perp$ values each of Ws and Ds.*

Proof. At Line 3 and Line 11 the user posts a function message to the public message board calling for Upd_s and posts shares of y to all server players. Suppose this happens in round n. Since Upd_s is a user function, it runs asynchronously. This means each honest server player will begin Upd_s in round $n+1$. Since the user has already posted values matching y_shares, then in round $n+1$, the honest server players will compute Ws and Ds and post them privately to the user.

Since there are at least t honest server players, this means at least t messages satisfying the requirements for Ws and Ds will be posted in round $n+1$. That means that in round $n+2$, the user will find all of these messages. □

Now we show that corrupted server players must compute the update polynomial correctly or be blamed.

Proposition 20.4. *If an honest user returns from* **GET_UPDATE** *with $d = \perp$ or $W = \perp$, then a corrupted server player will be blamed in 2 rounds.*

Proof. Users only call **GET_UPDATE** from Line 6 of **RETURN_UPDATE**, and they immediately call $\mathbf{BLAME_U}$ if d or W is $\perp$. Considering $\mathbf{BLAME_U}$, it obtains the same values of V, Y, y_shares, d, k, and m which honest server

players will pass to UPD_HELP. Since UPD_HELP is deterministic, its outputs W_{check} and D_{check} will precisely match the outputs that an honest server player would produce. Thus, calling BLAME_U with id of an honest server player will return without calling any blame.

Since this is the only condition not to start the blame condition of BLAME_U, then if the user does not execute this branch, then all values in Ws and Ds must precisely fit the same polynomial, so neither d nor W will be $\perp$, contradicting the condition which started BLAME_U. Since this is a contradiction, we know that at least one corrupted player i sent values of W and D which do not match what an honest server player would have posted, and the user will enter the blame branch of BLAME_U for player i.

Once the user does this, they reveal all messages between i and the user for the period from when the user sent shares of y_u to i and to when i returned W, D. In our formalism the control program posts these, though in practice this can be achieved by signing all messages and having the user post the signed messages themselves. Since the function message of BLAME will be posted in the same round as these values, the server players which begin the call to BLAME in the next round will find these messages when they run BLAME and they will produce a non-empty array $\mathcal{Y}$. Once they have found this, they will search for messages from i to the user player with W, D, h. As the messages the user requests will be revealed in the same round (again, formally by the control program; in practice by the user with the servers' signature), if they find the messages with shares of y_u, they will find the incorrect response from the corrupted server player. Since we know the values W and D that the corrupted server player i returned do not match the output of UPD_HELP, all honest server players will post a BLAME message towards player i in this round and the accumulator will abort. □

We next argue that updates work correctly when users are not deleted. This is the normal case, though we will need to do a lot of work later to show that updates work correctly in other cases.

Proposition 20.5. *If* witness *is a valid witness for epoch n and a user begins* RETURN_UPDATE(witness, n, m) *and their user ID y_u was not deleted between epoch n and m, then either they will return with a valid updated witness within* 2 *rounds or a corrupted server player will be blamed and the accumulator will abort.*

Proof. Once a user begins RETURN_UPDATE, they immediately begin GET_UPDATE. As Proposition 20.3 shows, within exactly 2 rounds they will return. After this there are three cases: $\mathcal{D}$ or $\mathcal{W}$ have one or more $\perp$ values, $\mathcal{D}$ has a zero value, or $\mathcal{D}$ has no zero values.

By Proposition 20.4, if the first case occurs, a corrupted server player is blamed.

If this case does not occur, then for each reconstructed value, OPEN_SHARE reconstructed a polynomial $\tilde{p}$ from t shares that matches all other returned shares. Let p be the "true" polynomial, i.e., the one that would be formed if

all players were honest. Since p and $\tilde{p}$ evaluate to the same value on all honest server players IDs, and there are at least t honest server players, then $p - \tilde{p}$ has t roots. Since it has degree at most $t-1$, it must be the zero polynomial. Thus, all responses from all players match the output of honest players. Inspecting the logic of UPD_HELP, this means the batch update polynomial was correctly evaluated: the user has $\mathcal{D}$ and $\mathcal{W}$ as defined in Proposition 20.1.

The values of $\mathcal{D}$ contain a 0 if and only if y_u was deleted, which by assumption did not occur. In the final case, where $\mathcal{D}$ is not zero for any term, this means the user will have a valid updated witness by Proposition 20.1. □

Proposition 20.5 relies on the ability of the control program to reliably output the contents of the private message board, which in practice will rely on digital signatures on all messages.

Proposition 20.6. *Assuming no server players are blamed,* RETURN_UPDATE *will recurse at most once.*

Proof. The only time RETURN_UPDATE is called, the calling function ensures that new_epoch $\leq |$accs$|$. A recursive call to RETURN_UPDATE occurs only if GET_UPDATE returns $\mathcal{D}$ with a zero value.

Suppose that some value is 0. This induces a call to FIND_WITNESS at Line 19, which returns the latest available witness posted to the accumulator with an epoch at most new_epoch, and then RETURN_UPDATE recurses only if the epoch n of this witness is less than new_epoch.

During the recursive call, it could only recurse again if some value of $\mathcal{D}$ again, but this implies that the user's ID y_u was deleted between n and new_epoch. We can conclude that the user was not re-added between n and new_epoch, or FIND_WITNESS would have returned the newer witness produced by this add. Thus, if $\mathcal{D}$ also contained a zero value in the recursive call, so that RETURN_UPDATE calls FIND_WITNESS again, it will return $\perp$, and RETURN_UPDATE will return immediately at Line 22.

□

We can now prove a maximum number of rounds for an update to complete. The timing will depend on the user, whether they need to recurse (based on whether their ID was deleted and re-added) and whether they are waiting for a previous update. This doesn't matter for progress or correctness as long as we have a bound, but the distinction matters for anonymity, which we address in more detail in Section 26.

Proposition 20.7. *Assuming no server players are blamed, if an honest user player calls* RETURN_UPDATE(witness, old_epoch, new_epoch) *with* witness.epoch = old_epoch *and* new_epoch $\leq |$accs$|$*, then they return in either 2 or 4 rounds.*

Proof. First, they return in exactly 2 rounds from GET_UPDATE. If no value if $\mathcal{D}$ is 0, then by the assumption on old_epoch they will skip the loop at Line 36 and immediately return. If some value of $\mathcal{D}$ is 0, the user calls

FIND_WITNESS. From Proposition 19.2 they will return in the same round, since the maximum round argument will be at most the highest round of a consensus accumulator, so they will not need to wait for the honest server players to post their messages because they are already posted. If the user recursively calls RETURN_UPDATE, then it takes another 2 rounds for GET_UPDATE and then they will again immediately return from FIND_WITNESS, or return immediately if they take other branches of execution. □

Proposition 20.8. *If* RETURN_UPDATE$(*, *, \mathsf{new_epoch})$ *returns a witness, the epoch of the new witness equals* new_epoch.

Proof. Clear by inspection of all return statements in RETURN_UPDATE. □

We now must handle the case of multiple simultaneous updates. First, multiple update requests in the same round are ignored: the user chooses the latest one only. However, if they are requested to update again before the first update is finished, then this could de-anonymize them if they waited for the first update to finish. We can avoid this because the batch update polynomials do not need the old witness; we can save the results of the update and apply them to a new witness once a previous update finishes. To track this, we use next_witness to refer to the maximum epoch of any requested update. We next show that this works correctly:

Proposition 20.9. *For the value of* next_witness *of an honest server player at any point in execution, either they have a witness* $(\text{next\_witness}, C, R_m)$ *or there is a function* UPDATE(next_witness) *in their private function queue.*

Proof. Suppose there is no function UPDATE in the user's private function queue. Since UPDATE is the only function that updates next_witness, either no UPDATE has been called (in which case next_witness $= 0$ and the proposition holds) or previous calls have finished. A previous call to UPDATE will only complete after setting next_witness to the round of the new player witness returned by RETURN_UPDATE, proving the other case of the theorem statement. □

We next cover the case where a previous update is still executing when a new update begins. The main point is that the second update can immediately use the results of the previous update, so there is almost no delay to the second update. The only way a delay can occur is if the second update takes only 2 rounds for GET_UPDATE, but the first update started 1 round before and takes 4 rounds. Then the second update completes as soon as the first update finishes, but this is 3 rounds after the second update starts. To formally summarize:

Proposition 20.10. *If* $m >$ next_witness*, then* UPDATE(m) *returns in either 2, 3, or 4 rounds, and when it returns, the epoch of* witness *equals* m *(or a corrupted server player is blamed). If* next_witness $\geq m$*, then* UPDATE(m) *returns immediately without changing* witness*.*

Proof. The logic of **UPDATE** makes the second half of the statement clear. For the first half, we prove by induction on n, the number of calls to **UPDATE**.

For any n, if the epoch of **witness** equals **next_witness** when **UPDATE** is called, then they will also be equal when it passes these arguments to **RETURN_UPDATE**, and by Proposition 20.7 and Proposition 20.8, the result holds. This proves the base case, since **next_witness** always matches the player's current witness if no updates have been called, as well as proving the inductive case in certain instances (i.e., when only one update is in progress at a time).

It remains to prove the statement when **next_witness** does not match the epoch of the player's current witness during the nth update. In this case, by Proposition 20.9, there is a call to **UPDATE**(**next_witness**) in the private function queue. By assumption, **next_witness** $< m$. Since calls to **UPDATE** in one round are sorted in reverse order of the epoch argument, that means the previous call could not have started in the same round as the nth call, which has epoch argument m.

Because the previous call did not immediately return, the inductive hypothesis implies that it finishes in 2 or 4 rounds of when it was called, i.e., if the previous call was in round t_{n-1}, it returns in round $t_{n-1}+2$, $t_{n-1}+3$, or $t_{n-1}+4$. Once it returns, it updates the value of epoch for **witness** to match the value of old_epoch given to **RETURN_UPDATE** during the nth call to **UPDATE**, and it updates **wit_waits**[old_epoch] to equal this new witness.

For the nth call, which is called in round t_n, **RETURN_UPDATE** finishes the first call to **GET_UPDATE** in 2 rounds. We consider two cases based on the value $\mathcal{D}$ returned:

- If all of $\mathcal{D}$ is non-zero, then (if it does not blame anyone) it will reach Line 36 of **RETURN_UPDATE** in round $t_n + 2$, where it waits for the new value of **wit_waits**[old_epoch]. We just showed that this will be updated by round $t_{n-1}+4$ at the latest, which we know is less than t_n+4. Once it has this value, it returns with a witness. This gives the timing result for this case.

- If some value of $\mathcal{D}$ is 0, then **RETURN_UPDATE** will look for a new witness at Line 19. Whether it finds a valid new witness or not, it ignores the previous update (as it should, since the user was deleted). By the same logic as Proposition 20.7, it takes at most 2 more rounds to finish recursing and return with a new witness.

Finally we show that the epoch of the new witness is updated: in both cases **RETURN_UPDATE** returns with a witness, and by Proposition 20.8, when **RETURN_UPDATE** returns the epoch of the witness it provides matches the argument of new_epoch, which is m. □

Corollary 20.11. **RETURN_UPDATE** *returns in either 2, 3, or 4 rounds.*

Proof. The case where it's called from **UPDATE** is already covered, so we consider **PROVE**. As it also calls **RETURN_UPDATE** with the value of

next_witness for old_epoch, the logic of Proposition 20.10 holds here as well. □

Having shown basic progress results for updates, we go on to prove correctness of the update. First we show that if a user is in the accumulated set, their update must return a valid witness.

Proposition 20.12. *Let n be an epoch with a consensus accumulator at some point in execution of the accumulator. If an honest user player has an ID y_u which is in $\mathcal{S}[n]$, then the user will return from* **RETURN_UPDATE**(witness, old_epoch, n) *with a valid witness, or a corrupted server player will be blamed and the accumulator will abort.*

Proof. We prove the first statement by induction on n. The statement is vacuously true for $n = 0$, as $\mathcal{S}[0] = \emptyset$.

For larger n, there are several cases based on the validity and epoch of the existing witness and whether the user was deleted before epoch n.

If witness is valid:

- **If the user was not deleted between epoch old_epoch and n:**

  If witness.epoch = old_epoch, and the user's ID is not deleted before epoch n, then they will end with a valid witness within 2 rounds by Proposition 20.5.

  If witness.epoch ≠ old_epoch and the user's ID is not deleted before epoch n, then the values of $\mathcal{D}$ returned from **GET_UPDATE** will be non-zero or else a corrupted server player gets blamed (see the logic of Proposition 20.1). Execution will skip to the loop waiting for **wit_waits**. Since a user player only calls **RETURN_UPDATE** with old_epoch set to the value of **next_witness** from the start of **UPDATE** or **PROVE**, then by Proposition 20.9, old_epoch equals the value of new_epoch from some previous call to **RETURN_UPDATE**.

  By the inductive hypothesis, this previous call will return a valid witness (since we assume the user was not deleted and they are in the accumulated set for epoch n). Inspecting **UPDATE**, it will update the array **wit_waits**. Since the previous call runs first, by the ordering on private user functions, then the later call will run, find the updated witness in **wit_waits**, and update it correctly.

- **If the user was deleted between epoch old_epoch and n:** In this case the value of $\mathcal{D}$ returned by **GET_UPDATE** will be contain a zero value (regardless of the value of witness.epoch, and they will call **FIND_WITNESS**. If the user's ID is in $\mathcal{S}[n]$, then this will return with a valid witness from the latest addition of y_u to the accumulator. We know that y_u was not subsequently deleted, since $y_u \in \mathcal{S}[n]$: **FIND_WITNESS** selects the latest available witness, so if y_u were deleted again before epoch n it would need

to be added again and this would create a later witness. The user may recurse RETURN_UPDATE, but by Proposition 20.5 they will then return within 2 rounds with a valid witness. If they do not recurse (meaning they were re-added in precisely epoch n), they return immediately with a valid witness.

If witness is invalid: In this case RETURN_UPDATE was not called recursively (since it will only recursively call itself after finding a valid witness with FIND_WITNESS), but was called directly from UPDATE or PROVE, both of which use witness as the argument. This means at Line 16 of RETURN_UPDATE, the user will follow the branch to recurse. Since their witness is invalid, then by contrapositive of the inductive case, this means $y_u \notin \mathcal{S}[\text{witness.epoch}]$. Thus, when they call FIND_WITNESS, we know they will find a valid witness for some epoch m, because $y_u \in \mathcal{S}[n]$, so they must have been added. Similar arguments apply to show that they were not deleted between m and n, so Proposition 20.5 shows the recursive call will return a valid up-to-date witness for epoch n.

□

We use this to show that users which *should* have valid witness do have a valid witness.

Corollary 20.13. *Suppose an honest user player with ID y has a witness (epoch, C, R_m) and $y \in \mathcal{S}[\text{epoch}]$. Then the user's witness is valid.*

Proof. If $\text{epoch} = 0$, then the statement holds vacuously because $\mathcal{S}$ is empty. The two ways to modify the value of a user's witness is by calling WIT or UPDATE. Proposition 19.4 shows WIT will return a valid witness for some round. In UPDATE, it only modifies the witness after calling RETURN_UPDATE. Proposition 20.10 shows that RETURN_UPDATE returns a witness for the epoch given as argument. The contrapositive of Proposition 20.12 means that if the value returned is not a valid witness, then $y \notin \mathcal{S}[\text{epoch}]$, which is what we need to prove. □

Taking all these proofs together, we've shown that if a user starts an update, they return with a witness within 4 rounds, and it is valid if they are in the accumulated set. We have so far given no guarantees about what happens if they are *not* in the accumulated set, other than stating that the user has some witness. For correctness, we need to ensure that users that are not in the accumulated set, they will not pass a verification protocol. Thus, the next proposition shows that users have an invalid witness for any epoch in which they are not in the accumulated set.

Proposition 20.14. *If an honest user calls*

$$\text{RETURN\_UPDATE}(\text{witness}, \text{old\_epoch}, \text{new\_epoch})$$

and $y \notin \mathcal{S}[\text{new\_epoch}]$, then RETURN_UPDATE will return an invalid witness.

Proof. We will proof by induction on new_epoch. At new_epoch $= 0$, y is not in the accumulator, but their witness is $(0, \mathcal{O}, \mathcal{O})$. This is invalid because $e(\mathcal{O}, y\tilde{P} + \tilde{Q}) = 1$, and $e(V, \tilde{P}) \neq 1$ for any accumulator because of non-degeneracy and because $e(P, \tilde{P}) \neq 1$.

If the adversary returns unexpected shares, then by the same logic as Proposition 20.5, the reconstructed polynomial is inconsistent and the user detects this and blames them.

Otherwise, for the inductive case, we proceed through two cases, based on whether the user was deleted or not between the epochs of the update. There is no overall structure to this proof, just an exhaustive accounting of possible cases and sub-cases.

The user was not deleted between old_epoch **and** new_epoch**:** We distinguish two sub-cases:

- If the argument witness is invalid, then the user enters the branch at Line 16. If witness is valid, then we know the and calls **FIND_WITNESS**, but they will find no witness since they cannot have been added to the accumulator in this interval if they were not deleted. Thus, the user returns at Line 22 with an invalid witness.

- If the argument witness is valid, then the user must have been deleted between the epoch of this witness and new_epoch. Since we assumed the user was not deleted between old_epoch and new_epoch, then the epoch of witness does not match old_epoch. The user will return $\mathcal{D}$ with no zero values and enter the branch at Line 31. They will wait for **wit_waits**[old_epoch] to update, and it will update with a witness for old_epoch, which must be invalid by induction.

  If $\mathcal{D} = [D_1, \ldots D_m]$ and $\mathcal{W} = [W_1, \ldots, W_m]$ are computed as honest server players, we know, similar to the logic for Proposition 20.3, that they can produce a valid witness for y, given a valid witness $C_0 = \frac{1}{y+\alpha}V$ as a starting point, as follows:

  $$\frac{1}{D_m}\left(\frac{1}{D_{m-1}}\left(\ldots\frac{1}{D_1}(C_0 - W_1)\cdots - W_{m-1}\right) - W_m\right) = \frac{1}{y+\alpha}V'$$

  At this point no player *has* the value C_0. We can multiply this out to get

  $$\frac{1}{D}(C_0 - W) = \frac{1}{y+\alpha}V'$$

  for $D = \prod_{i=1}^{m} D_i$ and a more complicated formula for W. We can solve for $W = \frac{1}{y+\alpha}V - \frac{d}{y+\alpha}V'$. Since the formula $\frac{1}{D}(* - W)$ is equivalent to what the user actually calculates in **PROCESS_UPDATE**, this implies that the new witness the user calculates at Line 45 of **RETURN_UPDATE** is

  $$C' = \frac{1}{d}(C - W) = \frac{1}{d}(C - \frac{1}{y+\alpha}V) + \frac{1}{y+\alpha}V'$$

This is valid if and only if it equals $\frac{1}{y+\alpha}V'$, but this occurs if and only if $C = \frac{1}{y+\alpha}V$, which implies that C is a valid witness of the accumulator value V. By inductive hypothesis, this is a contradiction, so C' must be an invalid witness.

The user was deleted between old_epoch **and** new_epoch**:** If $\mathcal{D}$ is computed honestly, then in this case it has a zero value. This prompts the user to enter the recursive branch at Line 16 and look for a new witness on the public message board. If no such witness exists, the user returns $\mathcal{O}$, which is invalid.

If a new witness exists, it means the user's ID was re-added between old_round and new_round; however, the proposition hypothesis implies they must have been re-deleted. When they call **RETURN_UPDATE** recursively, they will return $\mathcal{D}$ with a 0 value. They will then follow the recursive branch at Line 16 again and call **FIND_WITNESS** again. This time it will not find a new witness, and thus it will return $\mathcal{O}$, an invalid witness. □

Proposition 20.15. *If an honest user player with ID y has a witness for round n such that $y \notin \mathcal{S}[n]$, then the witness is not valid.*

Proof. When the user is initialized, they set their witness to $\mathcal{O}$, which is invalid.

Once the user is added to the accumulator, the witness returned from **WIT** corresponds to the first round they were added, so y is in the accumulator at that point and the proposition holds vacuously. Any later witness is the output of **RETURN_UPDATE**, so by Proposition 20.14, the user will not have a valid witness for round n if they update to such a round n. □

Finally, this all shows that verification works correctly.

Proposition 20.16. *If an honest player j receives a message of type* data *that begins with* "proof" *from an honest user with id y_u with* epoch *such that there is a consensus accumulator for* epoch*, then in the next round player j will call* **OBSERVER_CHECK_PROOF**$(*, j, b,$ epoch$)$ *such that $b = 1$ if and only if $y_u \in \mathcal{S}$*[epoch].

Proof. By Proposition 20.12 and Corollary 20.13, if $y_u \in \mathcal{S}$[epoch] then the witness will be valid. The user will compute all of the pairing and elliptic curve arithmetic and post to the private message board of the verifier. Inspecting the elliptic curve for a valid witness shows that it will always pass. The prover incorporates the verifier's challenge into its proof, and the verifier loops over all messages, so it will find the correct message and exit the loop.

Conversely, if $y_u \notin \mathcal{S}$[epoch], then the user will not have a valid witness, by Proposition 20.15. Then they will send $\perp$ as a proof, which always fails verification. □

21 Correctness

21.1 Outline

In this section we show the first security property, correctness. At this point all that is left to show is that users are able to verify their identity before the observer program times out.

The previous bounds on update times imply that if a consensus accumulator already exists, then the verification protocol completes within only 5 rounds (Proposition 21.1). However, if no consensus accumulator exists, the verifier will wait until it is posted. We need to show that this wait is not too long.

We have most of the results on progress from Section 16, but we need fixed bounds. For this we first show that honest players consistently track Beaver triple indices (Proposition 21.2) and thus Proposition 21.3 that they need only one round to retrieve Beaver triples (since we abstracted away triple generation, so the control program posts them), and Proposition 21.4 that they need a bounded number of rounds to retrieve shared random values.

This is all we need to show that all the core accumulator functions have bounded times, in Proposition 21.5. This means, Corollary 21.6, the number of rounds to finish all queued changes – i.e., all calls to Add or Del that have not yet been executed – is at most the bound computed by the observer function in **OBSERVER_START_PROOF**. Proposition 21.7 shows that every verification challenge will be resolved before the accumulator times out, by combining the last result with the basic structure of the verification protocol. This means – Proposition 21.8 – that the accumulator will not time out, at least with reasonable lower bounds on the observer program's constants.

From our previous result Proposition 20.16 that proofs result in the correct answer, these results bounding the time for progress are enough to show Theorem 21.9, that ALLOSAUR is correct.

21.2 Proofs

First we show that if a user is requested to prove their identity when a consensus accumulator exists, they do so within 5 rounds.

Proposition 21.1. *If an honest user player with pseudonym* verifier *receives a* Ver*-type function message in round t with arguments i,* epcoh *such that:*

- *this message is the only message received in round t and is the first message to the pseudonym* verifier*,*
- *player i is an honest user player with ID y_u,*
- verifier $\notin \mathcal{C}$*,*
- *there is a consensus accumulator for* epoch*,*

then verifier *will call* **OBSERVER_CHECK_PROOF**$(*, *, n)$ *with* $n \geq$ epoch *within* 5 *rounds.*

Proof. By assumption the verifier will pass the check at Line 2 and continue execution, because there is already a consensus accumulator. They will then send a message to the prover, who will begin execution of **PROVE** in the next round. Once the prover starts, if epoch is greater than **next_witness**, they will execute **RETURN_UPDATE** and by Corollary 20.11 they will return with within 4 rounds with a witness for the accumulator with index epoch, which prompts them to post a Ver_s message to the private message board of the verifier. If epoch $<$ **next_witness**, they will post a Ver_s message with a witness for the accumulator with index **next_witness**.

One round later, the verifier will start executing **VER** because it was posted in its private message queue. We showed that there is a value of **accs** for this round number, so the verifier will execute all of the checks. Then the verifier will call **OBSERVER_CHECK_PROOF** in the same round, using the epoch number in the Ver_v message, which is at least as large as epoch. □

However, a user may be requested to verify their identity for an accumulator for which a consensus accumulator does not yet exist. We must show that the consensus accumulator will be posted within some bounded time.

For this, we first ensure that honest server players always have enough secret shares, based on when they ask for more secret shares. They request new shares based on an internal count of random shares, so we show that this internal count is never larger than the actual number of shares posted.

Proposition 21.2. *At all points, for any honest server player,*

$$\textbf{triple\_share\_counter} \leq \textbf{triple\_share\_count}.$$

Further, if **triple_share_count** $= n$ *in some round, then by at most the next round, the public message board and the private message board of each server player will contain messages with shares and commitments with ID values up to* $n - 1$.

Proof. For the first statement, by induction, it holds at the start of the accumulator (both variables are initialized to 0). In all subsequent rounds, **triple_share_counter** only increments at the end of **GET_SHARED_TRIPLE**, but it only reaches the end if it finds a posted share with ID equal to **triple_share_counter**, and such a value is only posted if **triple_share_count** is at least as high as that value.

For the second statement, the control program posts public and private messages with shares with ID **triple_share_count** before incrementing that value. Thus, for any value of **triple_share_count**, public and private messages with all integer IDs less than that value have been at least queued to be posted. Since a queued message is posted within one round, they will all appear on the public and private message boards within at most one round. □

Proposition 21.3. *Honest server players return from* **GET_SHARED_TRIPLE**() *within at most 1 round.*

Proof. The control program function for **INPUT_TRIPLE** always post messages publicly and privately with the same IDs, so in any round, if the public data is available for a share with some ID, the private data is available as well. Thus, if a player returns from the **WAIT**, it will return in the same round from **PRIVATE_WAIT**.

If there is already a satisfying message for the **WAIT**, the player will return in the same round that it called **GET_SHARED_TRIPLE**. If there is not such a message in the player's **data_set**, then the player will enter the first **if** branch before calling **WAIT**. This branch tells the control program to post another random share; this will increment **triple_share_count**, so by Proposition 21.2, this ensures that **triple_share_counter** $\leq$ **triple_share_count** $+ 1$.

Proposition 21.2 also shows that by the next round, a share with ID equal to **triple_share_counter** will be posted on the public and private message boards. Thus, the player will complete both **WAIT** and **PRIVATE_WAIT** by the next round at the latest. □

Proposition 21.4. *Honest server players return from* **GET_SHARED_RANDOM**() *within* $|\mathcal{C}|\cdot$**WAIT_THRESHOLD**$+1$ *rounds.*

Proof. The structure of **GET_SHARED_RANDOM** is the same as any other **WAIT** loop, so by Proposition 16.8, it will return in $|\mathcal{C}|\cdot$**WAIT_THRESHOLD**$+$ 1 rounds. □

We now have the timings of all subroutines of the core accumulator functions. To finish:

Proposition 21.5. *If a corrupted player does not get blamed, then the maximum number of rounds to execute the following functions is given by:*

- **OPEN***:* $2(|\mathcal{C}|$**WAIT_THRESHOLD** $+ 1)$
- **INVERT***:* $6(|\mathcal{C}|$**WAIT_THRESHOLD** $+ 1)$
- **DEL***, if it does not execute the blame branch:* $10|\mathcal{C}|$**WAIT_THRESHOLD**$+$ 11
- **ADD***, if it does not execute the blame branch:* $19|\mathcal{C}|$**WAIT_THRESHOLD**$+$ 20

Proof. **OPEN** has 2 **WAIT** loops, immediately giving the number of rounds with Proposition 16.8; **INVERT** calls **OPEN** 3 times.

AFF_INV_ACC and **AFF_INV_SIGN** calls **INVERT**, **OPEN**, **GET_SHARED_RANDOM** and **GET_SHARED_TRIPLE** once each, giving $9|\mathcal{C}|$**WAIT_THRESHOLD** $+ 10$ as the maximum number of rounds each. **POST_ACCUMULATOR** has one more **WAIT** loop. Since **DEL** calls **POST_ACCUMULATOR** and **AFF_INV_ACC**, those are the only two functions containing calls to **WAIT** and this gives the result. Similarly for **ADD**, but it may also call **AFF_INV_SIGN** during **SIGN**. □

For convenience, we thus define $T_{wait} := (\mathcal{C}|\textbf{WAIT\_THRESHOLD} + 1)$, and then $T_{open} := 2T_{wait}$, $T_{inv} = 6T_{wait}$, $T_{del} := 10T_{wait} + 1$, and $T_{add} := 19T_{wait} + 1$.

Corollary 21.6. *Let* T *and* changes *be defined as in*

OBSERVER_START_PROOF

during some round n*. Then if*

$$\textbf{time\_per\_change} \geq T_{add},$$

all posted changes to the accumulator via **ADD** *and* **DEL** *will be completed by round* $n + \mathsf{changes} \cdot \textbf{time\_per\_change}$*.*

Proof. Every time **ADD** or **DEL** is posted to the message board, the size of $\mathcal{A}$ increases. Every time one of these functions finishes, all server players post an accumulator value which the control program adds to $\mathcal{A}$. If the corrupted players do not also post an accumulator, they will be blamed and the accumulator will abort, so we can assume they post some value. This means there are $|\textbf{servers}|$ messages. If the corrupted server players try to post extra accumulator values, they will not post enough values since there is at least one honest server player. This all implies that the value of T computed at Line 16 will be exactly the index of the last accumulator that the honest server players posted. Thus, changes precisely corresponds to the number of uncompleted changes to the accumulator.

Since each change requires a call to **ADD** or **DEL**, and these calls will occur immediately after one another, we use Proposition 21.5 to bound the total time for all of the calls. □

From now on, we assume that $\textbf{ver\_time\_limit} \geq 6$

Proposition 21.7. *If the unique and first message to a pseudonym* j *is a* Ver *message with arguments* i *and* epoch *such that* i *and* j *are pseudonyms of honest user players and* $\mathsf{epoch} \leq |\mathcal{A}|$*, then before round* time_limit *(as calculated in* **OBSERVER_START_PROOF** *at Line 19), player* j *will call*

$$\textbf{OBSERVER\_CHECK\_PROOF}(i, *, *).$$

Proof. The value time_limit is calculated as the difference between the length of $\mathcal{A}$, which represents the total number of **ADD** or **DEL** messages, and the current highest index of a consensus accumulator (denoted as T). By Corollary 21.6, all these posted changes will be completed within $\mathsf{changes} \cdot \textbf{time\_per\_change}$ rounds. Since the round number is at most as large as $\mathcal{A}$, one of these changes will post an accumulator with a counter at least equal to the argument to Ver_u.

Until that round, the verifier will loop in **PROVE** at Line 5. Once that accumulator is posted, they will exit the loop and then by Proposition 21.1 the verification will finish within at most another 6 rounds. This gives a total of $6 + \mathsf{changes} \cdot \textbf{time\_per\_change}$, less than time_limit computed in Line 19. □

Proposition 21.8. *If*

$$\textbf{ver\_time\_limit} > 7 + 2\textbf{time\_per\_change}$$

and

$$\textbf{time\_per\_change} \geq 17(\mathcal{C}|\textbf{WAIT\_THRESHOLD} + 1),$$

then the accumulator will not time out (i.e., run **ACCUMULATOR_FAIL** *because* **round_num** $>$ **round_limit***) before blaming a corrupted player.*

Proof. First, **OBSERVER_START_PROOF** will not add a challenge (and hence no time limit added to the accumulator) if either the prover i or the verifier j are corrupted. Thus we can assume they are both honest.

Proposition 21.7 shows that for each challenge $(i, j, n, \mathsf{time\_limit})$, player j will call **OBSERVER_CHECK_PROOF**(i, b, n) for player i, valid for an accumulator of index at least n (by Proposition 21.1), before time_limit. This means that challenge will be deleted before the time limit. As the value **round_limit** as always the minimum time_limit over all non-deleted challenges, then **round_limit** will always increase before that round is reached. □

Theorem 21.9. *The accumulator is correct.*

Proof. The adversary wins the correctness game if the game outputs FAIL. There are three ways for this to occur: an honest player is blamed, a verification between honest players gives the wrong answer, or the accumulator times out.

By assumption, there are not enough adversaries to exceed the blame threshold to blame an honest player unless some honest server players also blame another honest player. The only times honest players will post blames are:

1. during **GEN** when generating parameters
2. during **WAIT**, if it times out
3. during **ADD**, if:
   - the sender posted the function request before **GEN**
   - if a server does not post a label with a "witness" label
   - during **AFF_INV_ACC** or **AFF_INV_SIGN** if the inversion is not computed correctly

   the sender posted the function before **GEN**
4. during **POST_ACCUMULATOR**, if a player posts something not matching the accumulator
5. during **BLAME**

Honest players will not blame each other in any of these instances because, respectively:

1. Proposition 17.8 states that honest players do not blame each other during **GEN**.
2. Corollary 16.7 states that honest players are not blamed during waits.
3. Proposition 17.4 states that **ADD** and **DEL** do not blame honest players.
4. Proposition 18.3 states that honest server players are not blamed when posting accumulators.
5. Proposition 20.2 shows that honest server players do not blame each other, except with negligible probability, during **BLAME**.

For verifications, Proposition 20.16 shows that verifications between honest users will succeed if and only if the user is in the accumulated set. As this is exactly what the observer program checks, it will not call **ACCUMULATOR_FAIL** during these calls.

By Proposition 21.8, the accumulator will not fail by timing out. □

22 Simulation

22.1 Outline

For our proofs of security, we will need to simulate the control program and all honest users. We will use several techniques many times, so the following lemmas will be helpful.

Throughout, a *simulator* will refer to any program that is interacting with the adversarial program and simulates the behaviour of the control program and honest users. We let $k = |\mathcal{C}|$, the number of corrupted server players.

First, the verification proofs are honest-verifier zero knowlege, thanks to Nguyen [Ngu05], so in Lemma 22.1 we show that any simulator can then reprogram the random oracle to simulate verification proofs at any time.

The next main task of simulation is in cases where we use an accumulator adversary to solve a hard group problem, where the secrets of the accumulator are replaced by unknown aspects of the group problem. This becomes problematic because the accumulator requires multi-party computation with those shared secrets, which the simulator would not have. To simulate this, we first make a hybrid argument in Lemma 22.2 that because the commitments used in **OPEN** are binding, we can always assume the adversary is the first player to open its commitments. Because a simulator can reprogram the random oracle, they can open their own commitments to anything they want, this gives us a powerful result (Lemma 22.3) that a simulator can indistinguishably open any shared secret to any value it chooses, so long as the result is otherwise indistinguishable to the adversary. For example, in Proposition 22.5, we apply this to Beaver triples and shared randomness, and show that a simulator can choose whichever valid Beaver triples or random value it wants. Notice that if it opens the components of a Beaver triple to values that are not a valid Beaver triple, an

adversary may (and in most cases, will) detect this change. This final result relies on faking the commitments to these values, which requires the accumulator to instantly blame the adversary in any blame branch. Lemma 22.4 shows that such a hybrid is just as secure, since we previously showed (e.g., Theorem 21.9) that only corrupt players are blamed anyway.

22.2 Proofs

First, we show that we can simulate verification proofs.

Lemma 22.1 ([Ngu05])**.** *The verification protocol is honest verifier zero-knowledge. Specifically, a simulator can simulate v proof messages without using a valid witness, with probability at least $1-\frac{vT}{2^{2\lambda}}$, where T is the number of random oracle queries.*

Proof. To simulate proofs, the simulator will execute all functions as expected (e.g., by acting as control program, random oracle, honest players, etc.), except for **PROVE**. As the simulator will know all pseudonyms and the accumulator structure, when a proof request is sent to a pseudonym p, the simulator knows whether user y owning p should have a valid witness or not. It then executes the updates honestly, as necessary. If y should not have a valid witness, the simulator sends a message with a proof of $\perp$.

If y should have a valid witness, the simulator generates uniformly random values for $s_0, \ldots, s_7$ and for c, then uniformly random U_1, U_2, and R. It then computes simulate:

$$\begin{aligned}
T_1 =& s_1 X_1 + s_2 X_2 + s_3 Y - cR \\
T_2 =& s_4 X_1 + s_5 X_2 + s_6 Y - s_7 R \\
\Pi_1 =& e(K, \tilde{K})^{s_0} e(U_1 \tilde{K})^{-s_7} e(Y, \tilde{Q}_m)^{k_1} e(K, \tilde{K})^{c} e(U_1, \tilde{Q}_m)^{-c} \\
\Pi_2 =& e(U_2, \tilde{P})^{-s_7} e(Y, \tilde{P})^{s_5} e(Y, \tilde{Q})^{s_2} e(V, \tilde{P})^{c} e(U_2, \tilde{Q})^{-c}
\end{aligned}$$

which produces the same distribution as real *valid* proofs. It then modifies the random oracle to output c on input $(\mathsf{challenge}, V, U_1, U_2, R, T_1, T_2, \Pi_1, \Pi_2)$ (where challenge was the argument of the proof challenge message), which is undetectable with probability at most $\frac{T}{2^{2\lambda}}$.

By Proposition 20.16, this matches what honest users should send, so it is an indistinguishable simulation, and the union bound over all proof messages gives the result. □

We then argue that any adversary against the original accumulator definition implies an adversary against a slightly modified definition. In this definition we want to force the adversary to open its shares first for any secret share.

For this new definition, we add a call to the following function after Line 6 of **GAME_NEXT_ROUND**:

```
CHECK_REVEAL ():
1:  // Loops over all public messages
2:  i ← 0
3:  loop
4:    // Get the next public message
5:    m ← MESSAGE_BOARD_PULL(i)
6:    if m = ⊥ then
7:      BREAK
8:    end if
9:    i ← i + 1
10:   // Checks for any commitment message from a corrupt server
11:   if m.type = DATA and m.label = "open_commit and m.sender ∈ C
      then
12:     // Iterate through all messages again, looking for the matching
        opening
13:     j ← 0
14:     found_reveal ← FALSE
15:     loop
16:       m′ ← MESSAGE_BOARD_PULL(j)
17:       if m′ = ⊥ then
18:         BREAK
19:       end if
20:       if m′.type = DATA and m′.label = "open_reveal" and
              m′.sender = m.sender and
              m′.open_number = m.open_number then
21:         Let c be the commitment from m and o the opening from m′
22:         // If the open number matches, but the opening doesn't, this
            fails
23:         if OPEN_COMMIT(o, c) == ⊥ then
24:           ACCUMULATOR_ABORT(m.sender)
25:         else
26:           // When a valid opening exists
27:           found_reveal ← TRUE
28:         end if
29:       end if
30:       j ← j + 1
31:     end loop
32:     // If it exits the loop without setting found_reveal, it found nothing
        and should blame the adversary
33:     if not found_reveal then
34:       ACCUMULATOR_ABORT(m.sender)
35:     end if
36:   end if
37: end loop
```

Program 75: Checks if the adversary has posted an opening to all of the commitments it has posted, and aborts (blaming the adversary) if it cannot find one.

Denote this game as an adversary-first accumulator. We prove next that adversaries against the original accumulator are easily made into adversary's against the new accumulator, which follows readily from the binding property of the commitment scheme. Since our commitment scheme is based on a random oracle, we give the advantage loss directly.

Lemma 22.2. *An adversary $\mathcal{A}$ against the original accumulator that succeeds with probability p implies an adversary $\mathcal{A}'$ against an adversary-first accumulator that succeeds with probability $p' \geq p - O(\frac{CT^2}{2^{2\lambda}})$, where T is the total number of random oracle queries of $\mathcal{A}$ and C is the total number of times it opens a shared value.*

Proof. We describe $\mathcal{A}'$. With the following exceptions, it simply forwards messages between the adversary-first accumulator and $\mathcal{A}$, though it stores all inputs and outputs to the random oracle that the adversary provides. When $\mathcal{A}$ posts a message with label open_commit, $\mathcal{A}'$ checks for an input (x, r) to the random oracle with output that matches the commitment. If there is exactly one input, then $\mathcal{A}'$ sets $o = (x, r)$ and posts a message of type open_reveal to the control program. Otherwise, $\mathcal{A}'$ aborts.

If $\mathcal{A}$ posts open_commit with a commitment it produced from the random oracle, then $\mathcal{A}'$ will get a valid opening to send to the control progam. If $\mathcal{A}$ posts a matching open_reveal, then $\mathcal{A}'$ will also faithfully forward this message to the control program. In this case, $\mathcal{A}'$ is a valid adversary-first accumulator adversary, and it perfectly simulates the accumulator to $\mathcal{A}$, so it has the same advantage. It remains to show that this case occurs, i.e., that $\mathcal{A}$ does not cheat on its commitments.

We first note that the probability that $\mathcal{A}$ makes *two* queries with the same random oracle output (which would cause $\mathcal{A}'$ to abort) is $O(\frac{T^2}{2^{2\lambda}})$. If $\mathcal{A}$ does not use the output of the random oracle to produce its commitment, then $\mathcal{A}$ has only a $\frac{T}{2^{2\lambda}}$ probability of finding an opening that will match its commitment. If the opening does not match, $\mathcal{A}$ would be blamed and the accumulator will abort, so $\mathcal{A}$ would not succeed. There is also some chance that $\mathcal{A}$ made only 1 query that output the commitment, but will later post a different open_reveal which also opens to the same commitment. Again, the probability of this is at most $\frac{T}{2^{2\lambda}}$.

Thus, the probability of any of these cases is at most $O(\frac{T^2}{2^{2\lambda}})$ for each opening. □

This is a vital hybrid because it allows us a simulator to spoof the opening. Implicitly this relies on the hiding property of the commitment scheme, where

again we show the security directly based on our random oracle commitment construction.

Lemma 22.3. *Let $\mathcal{A}$ be an adversary-first accumulator adversary where the game makes at most T calls to the random oracle. Let $\mathcal{X}$ be the set of all values that could be produced by a call to* **OPEN** *that would be indistinguishable to the adversary from interaction with the true protocol. Then with probability at least $1 - \frac{T}{2^{2\lambda}}$, a simulator can output messages so that the* **OPEN** *opens to any $x \in \mathcal{X}$.*

Proof. To accomplish this, the simulator will record all calls to the random oracle made by the adversary. During **OPEN** it posts random strings $\{c_{i_1}, \ldots, c_{i_k}\}$ as its commitments for the open_commit messages. The corrupted players will then post their own commitments, and also post openings to the commitments to the control program.

Since these are all valid openings, after they are all posted the simulator can choose shares for its own openings that will combine with the adversary's shares to produce any value in $\mathcal{X}$. To ensure these match the simulator's commitments, it then selects random values r_i and sets the random oracle so that **RANDOM_ORACLE**$(x_{i_j}, r_{i_j}) = c_{i_j}$ for $j = 1, \ldots, k$. If (x_{i_j}, r_{i_j}) was given as input to the random oracle at any previous point in the execution, we assume the simulation fails, but this will only happen with probability $\frac{T}{2^{2\lambda}}$, since r is chosen randomly. Otherwise, this is undetectable because the outputs c_{i_j} were chosen randomly.

If none of the above failures happes, then the commitments and the randomness are randomly distributed, and hence this is a perfect simulation. □

We then construct another hybrid, with the following change: As soon as an honest player posts a message of type BLAME (whether start, end, or with a user), the adversary loses. We call this a blame-free accumulator. We need this hybrid because once a blame begins, honest players will start to reveal their secret data. In many of the simulations we will need for security proofs, the simulator will not actually have this secret data and so the simulation would become detectable. However, since we have previously shown that once a blame starts, the adversary is always blamed, we would like to simply ignore anything that happens in the game about a blame starts and just immediately blame the adversary. Thus, we prove the following lemma:

Lemma 22.4. *If $\mathcal{A}$ is an adversary against an adversary-first accumulator with advantage ϵ making T random oracle queries, then there exists an adversary $\mathcal{A}'$ with advantage $\epsilon' = \epsilon - \frac{4T}{2^{2\lambda}}$ against a blame-free adversary-first accumulator.*

Proof. The propositions in previous sections show that when honest server players blame a player, they all blame the same player. By Theorem 16.6, they will all post the blames in the same round, so the adversary loses in the same round in either case if a blame is posted with a corrupted player's ID.

The only time BLAME messages are posted with start or end are during **BLAME**, **AFF_INV_ACC**, or **AFF_INV_SIGN**.

By Propositions Proposition 17.9 and Proposition 17.4, if an honest player enters the blame branch during the latter two functions, they will blame a corrupted player. They will end up blaming the same corrupt player, so an adversary is guaranteed to eventually lose if this happens. Since an adversary in the original game is not allowed to win while a (BLAME, start) message is posted, they cannot win between when this message is posted and when the majority blames them, so they have the same advantage in a blame-free accumulator.

The only other time that an honest player might post a blame is during **BLAME**, called during a witness update. By Proposition 20.2 this will blame an adversarial player if any honest server player posts any blame; therefore, it does not affect the adversary's chances of winning if we abort as soon as one honest player posts a blame message. □

Even though the simulator can spoof **OPEN**, most values are detectable in the broader context of the accumulator. As an example, a simulator could not open the values of a Beaver triple to (a', b', c') where $a'b' \neq c'$ without being detected. Next, Proposition 22.5 shows that shared randomness and Beaver triples can be opened to any random, valid value. That is, in the original protocol, the Beaver triples and randomness are created when they are committed to the public message board, and the actual values are out of the control of any player. Here we show that the simulator can control the value, as long as it fits the expected distribution.

Proposition 22.5. *With probability at least* $1 - \frac{NRT}{2^{2\lambda}}$ *(where* N *is the number of server players,* R *is the number of shared random values, including those in Beaver triples, and* T *is the number of random oracle calls the adversary makes), a blame-free adversary-first accumulator simulator can select random values for shared random values and Beaver triples in the same round that they are used, as long as they are randomly selected and, as required, valid Beaver triples.*

Proof. In a blame-free accumulator, the commitments to shared randomness and shared Beaver triples are never opened. The simulator will replace the commitments to random values.

Suppose the simulator switches its shares of a random value x to a different random value x'. If both x and x' are uniformly random (i.e., for a random secret share) then any computations involving the values of x and x' are statistically indistinguishable. For Beaver triples, the same argument applies, but a', b', and c' must be selected so that $a'b' = c'$. By lemma 22.3, this is so far indistinguishable.

To see that the commitments are also indistinguishable, for each share x'_i, the simulator can select random values r'_i and set **RANDOM_ORACLE**(x'_i, r'_i) to match the random commitment to the share x'_i it produced earlier. With probability at most $\frac{T}{2^{2\lambda}}$, (x'_i, r'_i) was not queried to the random oracle and so this is indistinguishable from an honest execution that initially committed to to x'_i. The probability that this fails for any of the $N - k$ commitments to any of the R shared random values is, by the union bound, at most $\frac{(N-k)RT}{2^{2\lambda}}$. □

23 Extraction

To show commitmentment soundness (Definition 10.2), we need an extractor, and we need to define the functions Acc (which produces an accumulator from values posted by all server players) and $\mathsf{Ver}(A, y, w)$, which outputs whether a witness w is valid for a given ID y and accumulator A. The function Acc simply takes an array of n accumulator values, returns $\perp$ if they are not all the same, and returns the first value if it matches all the other values.

The function $\mathsf{Ver}(A, y, w)$ computes the following, noting that $A = (V, \tilde{Q}, \tilde{Q}_m)$ and $w = (x, n, C, R_m)$:

$$e(C, y\tilde{P} + \tilde{Q}) =?e(V, \tilde{P})$$
$$e(R_m, y\tilde{K} + \tilde{Q}_m) =?e(xK + K_0, \tilde{Q}_m)$$

If both equations are true, it outputs 1; otherwise 0.

The only result in this section is Lemma 23.1, which takes most of its structure from the soundness proof of the Nguyen accumulator and its blind signatures (called an IAID protocol in [Ngu05]. The main difference (besides notation) is that we include a challenge string, sent by the verifier, which prevents replay attacks and is missing from [Ngu05].

Lemma 23.1. *Suppose that during the accumulator game an adversary $\mathcal{A}$ which makes at most T queries to the random oracle, with probability p an honest player calls* **OBSERVER_CHECK_PROOF**$(\pi, 1, n)$ *where π is a pseudonym belonging to an adversary. Then either there is an extractor $\mathcal{E}$ that produces a user ID y and a valid witness (n, x, C, R_m) for y with probability at least $p(\frac{p}{T} - \frac{1}{q}) - \epsilon$, or an algorithm to solve the discrete logarithm problem with probability $\frac{\epsilon}{3}$.*

Proof. Inspending **VER**, we see that there must have been a message of type data beginning with "proof" posted one round before the verifying player calls **OBSERVER_CHECK_PROOF** (let m be that round). That message must have $\mathsf{data.proof} \in G_1^3 \times (\mathbb{Z}_q)^8$, i.e.,

$$(U_1, U_2, R, c, s_1, \ldots, s_7) \leftarrow \mathsf{data.proof}$$

Since this message came from the adversary, the extractor can read it in the adversary's state, so the extractor then computes $\Pi_{1,2}$ and $T_{1,2}$ as in **VER** and checks all queries to the random oracle for an input of

$$(\mathsf{challenge}, V, U_1, U_2, R, T_1, T_2, \Pi_1, \Pi_2),$$

where the accumulator matching the epoch of the VER message is $(V, \tilde{Q}, \tilde{Q}_m)$, and challenge matches a message from the verifier to the adversary received one round before, i.e., round $m - 2$.

If such an input does not exist, then the value c will be completely independent of the "proof", and the probability that these checks will pass is $1/q^2$, so

the extractor can select a random $x, y \in \mathbb{Z}_q$, $C, R_m \in G_1$ and this will be a valid witness with probability $1/q^2 \geq \frac{1}{Tq^4}$.

The probability of the adversary querying precisely that input before round $m-1$ (when it received the challenge string) is $1/2^{2\lambda}$, so we ignore that case. Instead we assume the adversary queried the proof to the random oracle in round $m-1$. The extractor rewinds the adversary to that round and supplies a different random oracle output c' to the adversary.

Generally, the extractor will not be able to simulate the accumulator and honest players because it does not know their secret data. However, the extractor is able to replay all of round $m-1$ since all the honest players will only receive messages that the adversary posted in round $m-2$. This means even if the adversary modifies its execution in round $m-1$ (including communication to the accumulator) based on the random oracle output, it will not expect the accumulator to respond until round m. Thus, the extractor can indistinguishably simulate round $m-1$.

An honest verifier will only accept a proof sent 1 round after it sends its challenge. Thus, for the adversary to expect to win, it must post its proof in round $m-1$,, and so the extractor will receive a new proof before it loses its ability to simulate the game.

By the forking Lemma [BN06], the probability that the adversary sends another verification message with a proof also starting with $(U_1, U_2, R, c', \dots)$, which also verifies, is at least $p(\frac{p}{T} - \frac{1}{q})$.

Since this second proof verifies, the values of T_1', T_2', Π_1', and Π_2' computed in **VER** for the second proof must produce output such that the random oracle on input $(\pi, V, U_1, U_2, R, T_1', T_2', \Pi_1', \Pi_2')$ gives the value c' which the adversary sent as part of the proof. With all but negligible probability, the random oracle will not produce c' for any other input except the original input $(V, U_1, U_2, R, T_1, T_2, \Pi_1, \Pi_2)$, so that means $T_1' = T_1$, etc.

Then, given this second proof, we also have $s_1', \dots, s_7'$. We follow the proof of Lemma 2 from Nguyen [Ngu05]. We set

$$\begin{aligned} x &\leftarrow \frac{s_0 - s_0'}{c - c'} \\ y &\leftarrow \frac{s_7 - s_7'}{c - c'} \\ r_i &\leftarrow \frac{s_i - s_i'}{c - c'}, i \in \{1, 2, 3\} \\ R_m &\leftarrow U_1 - r_1 Z \\ C &\leftarrow U_2 - r_2 Z \end{aligned}$$

The extractor then checks that $r_i = \frac{s_{i+3} - s_{i+3}'}{y(c-c')}$ for $i \in \{1, 2, 3\}$ and aborts if this fails. We cover two cases:

All r_i values pass the check. Then the extractor sets (x, n, C, R_m) as a witness for y. We now prove that this is a valid witness.

We see that

$$\begin{aligned} e(C, y\tilde{P} + \tilde{Q})^{c-c'} =& e(U_2, y\tilde{P} + \tilde{Q})^{c-c'} e(Z, y\tilde{P} + \tilde{Q})^{-r_2(c-c')} \\ =& e(U_2, \tilde{P})^{y(c-c')} e(U_2, \tilde{Q})^{c-c'} e(Z, \tilde{P})^{-r_2 y(c-c')} e(Z, \tilde{Q})^{-r_2(c-c')} \end{aligned}$$

By how we calculated y, we know $y(c-c') = s_7 - s_7'$ and similarly $r_2(c-c') = s_2 - s_2'$ and $r_2 y(c-c') = s_5 - s_5'$. This gives

$$= e(U_2, \tilde{P})^{s_7 - s_7'} e(U_2, \tilde{Q})^{c-c'} e(Z, \tilde{P})^{-(s_5 - s_5')} e(Z, \tilde{Q})^{-(s_2 - s_2)'}$$

As argued above, since both proofs validate, Π_2 and Π_2', as calculated in **VER**, are equal, which implies:

$$\begin{aligned} e(U_2, \tilde{P})^{-s_7} e(Z, \tilde{P})^{s_5} e(Z, \tilde{Q})^{s_2} e(V, \tilde{P})^{c} e(U_2, \tilde{Q})^{-c} \\ = e(U_2, \tilde{P})^{-s_7'} e(Z, \tilde{P})^{s_5'} e(Z, \tilde{Q})^{s_2'} e(V, \tilde{P})^{c'} e(U_2, \tilde{Q})^{-c'} \end{aligned}$$

or equivalently,

$$\begin{aligned} e(U_2, \tilde{P})^{s_7 - s_7'} e(Z, \tilde{Q})^{-(s_2 - s_2)'} e(Z, \tilde{P})^{(s_5 - s_5')} e(U_2, \tilde{Q})^{c-c'} \\ = e(V, \tilde{P})^{-(c-c')} \end{aligned}$$

The left-hand-side is equal to the right-hand-side of Section 23, showings that

$$e(C, y\tilde{P} + \tilde{Q})^{c-c'} = e(V, \tilde{P})^{c-c'}$$

and since $c \neq c'$, $e(C, y\tilde{P} + \tilde{Q}) = e(V, \tilde{P})$.

Next we show that the second relation holds. We must show that $e(R_m, y\tilde{K} + \tilde{Q}_m) = e(K + xK_0, \tilde{K})$. We raise the left-hand-side to the power of $c - c'$ and use $R_m = U_1 - r_1 Z$ to obtain:

$$\begin{aligned} e(R_m, y\tilde{K} + \tilde{Q}_m)^{c-c'} =& e(U_1 - r_1 Z, \tilde{K})^{y} e(U_1 - r_1 Z, \tilde{Q}_m) \\ =& e(U_1, \tilde{K})^{y(c-c')} e(Z, \tilde{K})^{-r_1 y(c-c')} e(U_1, \tilde{Q}_m)^{c-c'} e(Z, \tilde{Q}_m)^{-r_1(c-c')} \end{aligned}$$

Once again substituting $y(c-c') = s_7 - s_7'$, $r_1 y(c-c') = s_4 - s_4'$ and $r_1(c-c') = s_1 - s_1'$:

$$e(R_m, y\tilde{K} + \tilde{Q}_m)^{c-c'} = e(U_1, \tilde{K})^{s_7 - s_7'} e(Z, \tilde{K})^{-(s_4 - s_4')} e(U_1, \tilde{Q}_m)^{c-c'} e(Z, \tilde{Q}_m)^{-(s_1 - s_1)'}$$

But since we know that $\Pi_1 = \Pi_1'$, we obtain

$$\begin{aligned} e(U_1, \tilde{K})^{-(s_7 - s_7')} e(Z, \tilde{K})^{s_4 - s_4'} e(Z, \tilde{Q}_m)^{s_1 - s_1'} e(U_1, \tilde{Q}_m)^{-(c-c')} \\ = e(K_0, \tilde{K})^{-(c-c')} e(K, \tilde{K})^{-(s_0 - s_0')} \end{aligned}$$

which when substituted in the above gives

$$e(R_m, y\tilde{K} + \tilde{Q}_m)^{c-c'} = e(K_0, \tilde{K})^{c-c'} e(K, \tilde{K})^{s_0 - s_0'}$$

Raising both sides to $1/(c-c')$, and noting that $x = \frac{s_0 - s_0'}{c-c'}$, we obtain

$$e(R_m, y\tilde{K} + \tilde{Q}_m) = e(K_0, \tilde{K})e(K, \tilde{K})^x = e(K_0 + xK, \tilde{K}).$$

Thus, this is a valid witness.

At least one r_i value fails the check. The extractor computed $r_i = \frac{s_i - s_i'}{c-c'}$ for $i \in \{1,2,3\}$. Define $r_j = \frac{s_j - s_j'}{c-c)}$ for $j \in \{4,5,6\}$. Here we assume that there is some $i \in \{1,2,3\}$ such that $r_i \neq yr_{i+3} \mod q$ (in the previous section, if that equality held for all i, extraction worked).

Given two successful witness proofs, we have that $T_1 = T_1'$ and $T_2 = T_2'$. This gives

$$s_1X + s_2Y + s_3Z - cR = s_1'X + s_2'Y + s_3'Z - c'R$$

Rearranging, dividing by $c - c'$, and using the values of r_1, r_2, r_3 gives

$$R = r_1X + r_2Y + r_3Z$$

Similar logic with T_2 and T_2' gives

$$\frac{s_7 - s_7'}{c - c'}R = yR = r_4X + r_5Y + r_6Z$$

Suppose that $X = m_xP$, $Y = m_yP$ and $Z = m_zP$. Together these equations give

$$((r_4 - yr_1)m_x + (r_5 - yr_2)m_y + (r_6 - yr_3)m_z)\, P = \mathcal{O}$$

Since P is a generator, this gives

$$(r_4 - yr_1)m_x + (r_5 - yr_2)m_y + (r_6 - yr_3)m_z \equiv 0 \mod q$$

To use this to solve the discrete log problem in G_1, we take a discrete log challenge (P, R) and randomly select one of X, Y, or Z to be R (without loss of generality, assume we chose $R = x$), and then we select random m_y and m_z and compute $Y = m_yP$ and $Z = m_zP$. We then simulate the control program and honest players of the accumulator, and run the extraction procedure. This is statistically indistinguishable from an honest accumulator, since X, Y, and Z are otherwise selected randomly in the public parameters.

At this point we know all variables in Section 23 except m_x, which is the discrete log of R with respect to P, and we can thus find it, unless $r_4 - yr_1 = 0$. If the adversary produces values such that $r_i - yr_{i-3} \neq 0$ for *some* i, we have a $\frac{1}{3}$ chance of guessing that value of i and solving the discrete log.

Summary. Let ϵ be the probability that, after the extractor obtains a valid witness, rewinds the adversary and obtains a second one, that there exists some i such that $r_i \neq yr_{i+3} \mod q$.

If we use the adversary to extract a witness, the probability of success is then $p(\frac{p}{T} - \frac{1}{q}) - \epsilon$.

If we use the adversary to solve the discrete log problem in G_1, since the interaction with the adversary is indistinguishable from the extraction case, the probability of $r_i \neq yr_{i+3} \mod q$ for the i we used as the discrete log challenge is $\frac{\epsilon}{3}$. □

We remark here that a general extractor is incapable of replaying the accumulator because it does not know the accumulator's secret values, adding difficulties to a simpler rewinding-based proof. This is no accident: a naively constructed verification protocol is vulnerable to replay attacks where an adversary simply forwards messages from honest users. Our solution to include a session-specific challenge and require a fast response from the adversary is by no means the only possible solution. Other types of session-specific input to the proof (e.g., the adversary's current pseudonym) would likely work, but would require a more difficult extraction proof.

24 Soundness

24.1 Outline

We now show soundness. At its core, our construction is the same as [KB21], who prove it is non-adaptively secure. Our multi-party definition inherently requires an adaptive security definition, so we must define a new hardness assumption, the n-Inversion Symmetric Diffie-Helman problem (n-ISDH), as follows:

Definition 24.1. Let G_1, G_2, and G_T be groups with a type-3 pairing from $G_1 \times G_2$ to G_T. Given G, $\lambda G, \gamma G \in G_1$, $\tilde{G}, \lambda\tilde{G} \in G_2$, and query access to a function $f : (y, Q) \mapsto \frac{1}{\lambda+y}Q$ for $Q \in G_1$, compute

$$\left(\prod_{i=1}^{k}(\lambda + y_i)\gamma G, y_1, \ldots, y_k\right)$$

such that at least one value of y appears in the list $y_1, \ldots, y_k$ at least one more time than we queried it to f.

We prove security in the general group model in Section 28.

This group problem is clearly similar to the accumulator, and we will attempt to simulate an accumulator whose secret value α (or s_m) is the hidden λ of the group problem. We can see that query access to f will allow us to simulate the affine inversions necessary for **ADD** and **SIGN**. However, to use this power to simulate an accumulator, we will need to simulate the multi-party computation that would normally produce the outputs of these functions. These computations are supposed to use the secret values directly, which we will not have if we are simulating the accumulator using the group problem oracle. Hence, using

the results of Section 22, Lemma 24.2 shows that we can use these outputs to simulate all the affine inversions in the core accumulator functions. This result means we can use the group problem oracle to simulate a full accumulator.

Recall that the soundness definition, Definition 10.2, requires that for all extractors, they should not be able to extract a witness for any ID except those belonging to the adversary and which are in the accumulated set corresponding to the epoch of the witness. This ensures adversaries cannot steal witnesses from honest users nor fake witnesses for IDs that should be revoked. We prove the two requirements separately: Proposition 24.3 uses the n-ISDH problem oracle to simulate additions and deletions to the accumulator, and shows that the extracted witness must be in the accumulated set if the n-ISDH problem is hard. Then Lemma 24.5 uses the n-ISDH problem oracle to simulate the long-term signatures (computing the additions and deletions honestly), showing that the extracted witnesses must belong to the adversary if n-ISDH and the discrete log problem are hard.

Finally, Theorem 24.6 summarizes these results to show that ALLOSAUR is blind commitment sound.

Throughout this section, unless specified otherwise, we consider only adversary-first blame-free accumulators.

24.2 Proofs

Lemma 24.2. *Suppose during execution of an accumulator against an adversary-first blame-free adversary $\mathcal{A}$, the function* **AFF_INV_ACC**(y) *begins. If a simulator knows a value V' such that $e(V', y\tilde{P} + \tilde{Q}) = e(V, \tilde{P})$, then the simulator can simulate execution of the function so that it outputs V' with probability $1 - \frac{4NT}{2^{2\lambda}}$. Similarly for* **AFF_INV_SIGN**(y, R).

Proof. Pairing non-degeneracy implies that $V' = \frac{1}{y+\alpha}V$, where α is defined such that $\tilde{Q} = \alpha\tilde{P}$.

During **AFF_INV_ACC**(y), the first call will be to **INVERT**, and as soon as the adversary posts its commitments to ϵ and δ, the simulator can open these commitments because it is an adversary-first game, and obtain the adversary's shares of ϵ and δ. Since the simulator knows the values of $([a], [b], [c])$ given to the adversary, they can compute the values the adversary should be using for $[x]_i$ and $[r]_i$ for each corrupted player (e.g., $[x]_i = [\epsilon]_i + [a]_i$). Here the values of $[x]_i$ must be the adversary's shares of $y + \alpha$, or else the computation would not proceed correctly and the blame branch would start and the adversary would lose.

The simulator picks a random z, and then can compute the adversary's point $[V']_i = [w]_i V = z^{-1}[r]_i V$. They can then select random $[w]_i$ for all simulated players except for some distinguished player i_0, who creates their share of V' by taking the value V' that we assumed the simulator knew and subtracting all the shares of other simulated players and the adversary. This means there is an "effective" share $[w]_{i_0}$, which is unknown to the simulator. Since, by construction, $\sum_i [w]_i V = V'$, it remains true that $\sum_i [w]_i = \frac{1}{y+\alpha}$. We

can choose the distinguished player at random from among the honest server players, since we will show how to simulate output from both the distinguished player and other honest server players.

In turn, this defines effective shares $[r]_i = z^{-1}[w]_i$ for each honest server player, where now $[r]_{i_0}$ is unknown to the simulator. The simulator thus chooses random shares $[\delta]_i$ and $[\epsilon]_i$; these define shares $[b]_i$ and $[a]_i$, where again $[b]_{i_0} = [r]_{i_0} - [\epsilon]_{i_0}$ and $[a]_{i_0} = [y + \alpha]_{i_0} - [\delta]_{i_0}$ are unknown to the simulator.

The simulator can then compute ϵ, δ, and thus all shares $[z]_i$ for all non-distinguished simulated players and adversarial players. For the share of the distinguished player, they will set $[z]_{i_0}$ to be the random value z, minus the shares of all other players.

At this point they resume execution and post the required shares for each honest server player. Since the honest server players' shares of $[a]$, $[b]$, $[c]$, and $[r]$ have changed, the simulator reprograms the random oracle so that the commitments to these values are valid; this is indistinguishable with probability at least $1 - \frac{4NT}{2^{2\lambda}}$ (there are $4(N-K)$ commitments to honest shares and T random oracle queries). If the adversary's shares of $[z]$ do not match what the simulator calculated, then it would not properly compute the inversion and would be blamed, so we can ignore this case because it is a blame-free accumulator.

We will show that this is identical to an honest transcript with a valid Beaver triple. To do this, we show that there are values of $[c]_i$ that would produce the revealed shares of $[z]_i$, which form a valid Beaver triple with the effective shares of a and b.

We first note that $\sum_i [a]_i = \sum_i [y + \alpha]_i - \epsilon = y + \alpha - \epsilon$. Similarly, $\sum_i [b]_i = r - \delta = \frac{z}{y+\alpha} - \delta$. This means

$$ab = z - \delta(y + \alpha) - \frac{z\epsilon}{y + \alpha} + \epsilon\delta$$

Then we see that:

$$\begin{aligned}\sum_i [c]_i =& \sum_i [z]_i - \epsilon \sum_i [r]_i - \delta \sum_i [y + \alpha]_i + \epsilon\delta \\ =& z - z\frac{\epsilon}{y + \alpha} - \delta(y + \alpha) + \epsilon\delta\end{aligned}$$

We thus see that this matches the transcript of a valid Beaver triple. In the normal protocol, there are q^3 possible values for r, a, b, and c, and for each variable there are q^{k-1} possible collections of shares, which then precisely determines a unique transcript. This gives q^{3k} possible transcripts consistent with an honest interaction.

The simulated interaction produces entirely random shares of ϵ and δ, for which there are thus q^k choices for each. It chooses z randomly (q choices), and then chooses q^{k-1} shares of $[w]_i$ randomly. Thus, there are also q^{3k} possible values which correspond to the same transcript, all chosen uniformly at random. If the commitments to these random values are never opened, then this is indistinguishable with high probability by Proposition 22.5, since the simulator has produced a transcript precisely matching a random r, a, b, c.

The final opened value will satisfy the relation for the accumulator. If the adversary expected it to enter the blame branch, the adversary expects to be blamed and thus will lose; thus, we have simulated this correctly for an adversary that expects to win. □

Recall that blind commitment soundness Definition 10.2 has two components: first, any adversary implies an extractor; second, for any extractor, the extracted witnesses must belong to the adversary. We showed the first in Lemma 23.1, so now we show that the extracted witnesses must belong to the adversary.

For this we first show that any extracted witness must correspond to an ID which is in the accumulated set for the epoch of the extracted witness.

Proposition 24.3. *For all adversaries against the accumulator's commitment soundness that make at most T queries to the random oracle, A additions to the accumulator, then if an extractor produces a user ID y and witness (n, x, C, R_m) such that $y \notin \mathcal{S}[m]$ for any $m \geq n$ with probabliity p, then there is an algorithm for the A-ISDH problem that succeeds with probability at least p using at most A queries to the A-ISDH problem oracle.*

Proof. Our goal will be to simulate the control program and honest players of the accumulator, as follows:

- Given the n-ISDH challenge G, λG, γG, $\tilde{G}$, $\lambda\tilde{G}$, the simulator sets $P = G$, $\tilde{P} = \tilde{G}$ as public parameters for the group.
- To simulate **GEN**, the simulator opens V to γG and $\tilde{Q}$ to $\lambda\tilde{G}$. By Proposition 22.5 this is simulatable.
- To simulate **ADD**(y), the simulator queries the group function f with the current accumulator value V and the input y to obtain V'. Using Lemma 24.2, they can simulate the remainder of **AFF_INV_ACC**.
- All other functions (and other subroutines of the previous functions) can be simulated honestly, since they do not use the secret value α (which is now the secret λ of the n-ISDH problem).

This simulates the accumulator perfectly, so we wait until the adversary outputs (j, i) to win the blind commitment soundness game. We then pass the contents of the message board and the adversary to the extractor, and we assume with probability p it outputs an ID y and a witness (n, x, C, R_m) such that $y \notin \mathcal{S}[m]$ for any $m \geq n$. The arguments that this simulation is indistinguishable to the adversary imply it is also indistinguishable to the extractor, which is also PPT.

The set of queries to the group problem oracle is precisely the multiset of elements added to the accumulator. If an element was added, deleted, then added again, it is queried twice.

Let $\mathcal{D}_n$ and $\mathcal{A}_n$ be the multisets of elements that have been deleted and added (respectively) to the accumulator with index n. If y appears k times in

$\mathcal{A}_n$, then y appears either $k-1$ or k times in $\mathcal{D}_n$. It appears k times if and only if $y \notin \mathcal{S}[n]$, since in that case y was deleted for every time it was added.

If the nth accumulator is $(V, \tilde{Q}, \tilde{Q}_m)$, we have

$$V = \left(\prod_{y \in \mathcal{D}_n} \frac{1}{y + \alpha} \right) \beta P.$$

The extractor output a valid witness (n, x, C, R_m); let $(V, \tilde{Q}, \tilde{Q}_m) = \mathsf{Acc}(\mathcal{A}[n])$, then

$$e(C, y\tilde{P} + \tilde{Q}) = e(V, \tilde{P})$$

which by non-degeneracy of pairings tells us that $C = \frac{1}{y+\alpha} V$.

We let $y_1, \ldots, y_k$ be the multi-set $\mathcal{D}_n \cup \{y\}$. Then our answer to the group problem is $(C, y_1, \ldots, y_k)$. First we notice that we have

$$C = \frac{1}{y + \alpha} V = \frac{1}{y + \alpha} \prod_{y' \in \mathcal{D}_n} \frac{1}{y' + \alpha} \beta P$$

as required. We must show that we have queried y one less time than it appears in the list. By construction, we know that y is not in $\mathcal{S}[m]$ for any $m \geq n$. This means there were no calls to $\mathbf{ADD}(y)$ since round n, and thus the number of queries to the group oracle with y is the total number of times that y appears in $\mathcal{A}_n$, which must be exactly as many times as it appears in $\mathcal{D}_n$ becauase $y \notin \mathcal{S}[n]$. Since it appears one more time than this in Section 24.2, we have solved the group problem. □

Importantly, we require that y was not re-added to the accumulator. The accumulator does not change when elements are added, so a witness valid at epoch n will remain valid until some element is deleted.

We next show that if the extractor produces a valid witness for an ID that does not belong to the adversary, it also solves the hard group problem. The proof is in some sense symmetric to Proposition 24.3, using the extracted witness to solve n-ISDH for the long-term signature part of the witness.

By the way we manage how users are added, there is some risk that an adversary will tell the observer program to create a user with ID y, but then the adversary will post $\mathsf{Add}(y)$ before the honest user can do so. Since the honest server players will only produce one signature for each ID y, only the adversary will have that signature. Thus, we must regard such users as corrupt; we prove that the observer program correctly identifies this case.

Proposition 24.4. *If the adversary sends the first* Add *message with argument y, then* **user_IDs**$[y]$ *will be a player ID belonging to the adversary.*

Proof. The first Add message prompts the observer program to associate the sender with **user_IDs**$[y]$ (during the message board commitment phase, which processes messages in the same order in which they are posted, at Line 19 of **OBSERVER_UPDATE_IDEAL**), and no further messages will do this. Thus, this permanently sets the "owner" of ID y to be the adversary. □

Lemma 24.5. *Under the same assumptions as Lemma 23.1, if (with probability p) an extractor produces an ID y and a valid witness (n, x, C, R_m) such that* **user_IDs**$[y] \notin \mathcal{C}$ *from an adversary that makes at most T queries to the random oracle, then there is an algorithm that solves the n-ISDH problem with probability at least $p - \frac{T}{2^\lambda} - \epsilon$, or there is an algorithm that solves the discrete logarithm problem with probability ϵ.*

Proof. Suppose that in the extraction game the extractor outputs a witness (n, x, C, R_m) for an ID y. Let ϵ be the probability that $xK + K_0 = \mathcal{O}$. We show how to solve the discrete log problem, or to solve the n-ISDH problem.

Discrete log solver. Let $G, H \in G_1$ be a discrete log challenge. We construct a simulator that behaves exactly as a true accumulator and honest parties, except we set $K = G$ and $K_0 = H$ in the public parameters. As these parameters are chosen randomly, this is indistinguishable to the adversary (and thus also to the extractor); therefore, there is a probability ϵ that at the end of the extraction game, the witness (n, x, C, R_m) satisfies $xK + K_0 = \mathcal{O}$. We then output $-x \mod q$ as the discrete log of H with respect to G, which is correct.

n-ISDH solver. Given an n-ISDH challenge G, λG, γG, $\tilde{G}$, $\lambda\tilde{G}$, the simulator sets $P = G$, $K = \gamma G$, and selects a random k_0 to set $K_0 = k_0 K$. All other parameters are set randomly.

During **GEN**, all but one player, which we will call the distinguished player i_0, is simulated as an honest player. The distinguished player will not compute s explitily, and instead will use Proposition 22.5 to open $\tilde{Q}_m$ to the value $\lambda\tilde{G}$ given by the group challenge. The simulator computes V and $\tilde{Q}$ honestly.

If an honest user i is requested to call **WIT**, the simulator selects x_i randomly and sets $R_{ID} = x_i(yP + \lambda P)$, where $\lambda P = \lambda G$ comes from the group problem. Since R_{ID} is distributed uniformly at random in the true protocol, this is (so far) indistinguishable. To simulate the proof in the Add message that **WIT** posts, the simulator follows the usual strategy of simulating Schnorr proofs: it chooses two random values $h, s \in \mathbb{F}_q$ and compute $R = sK + hR_{ID}$. It then sends (h, s, R) as the proof in the Add message, and sets the random oracle to output h on input (R_{ID}, R), which is undetectable with probability $\frac{T}{2^{2\lambda}}$. This produces an identical distribution as real proofs computing during **WIT**, since the secret ID value is selected randomly.

Given this construction, we then set $R_m = x_i G = x_i P$. This ensures that it is a valid signature, i.e.,

$$e(R_m, y\tilde{K} + \tilde{Q}_m) = e(P, \tilde{K})^{x_i(y+\alpha)} = e(R_{ID}, \tilde{K}).$$

Thus we can use this R_m as the output of **AFF_INV_SIGN**(y, R_{ID}), called during **SIGN**(y, proof). By Lemma 24.2, we can then simulate this. By Proposition 24.4, we do not need to call **SIGN** if y has already been added

If an honest user is requested to verify, we use Lemma 22.1 to simulating the proof message.

To simulate **SIGN**(y, proof) for other y (which did not come from **WIT**), we will need to simulate **AFF_INV_SIGN**$(y, R_{ID} + K_0)$. We will query the group problem for $f(y, R_{ID} + K_0)$, using Lemma 24.2 to set the opened value to equal the result of the query. The honest server players will only call **SIGN** at most once for each y, and they will call it for the first message with a proof argument. By Proposition 24.4, this means that if the adversary posted this y, then **user_IDs**$[y] \in \mathcal{C}$, i.e, y is considered corrupt.

All other aspects of the accumulator can be simulated honestly.

Having successfully simulated the accumulator, after the adversary outputs a valid result, we pass the adversary and the message board transcript to the extractor, which produces y and (n, x, C, R_m) with probability p. As before, our indistinguishable simulation remains indistinguishable to the extractor. The exracted witness is valid, meaning that $e(R_m, y\tilde{K} + \tilde{Q}_m) = e(xK + K_0, \tilde{K})$. In turn, this implies that $R_m = \frac{1}{y+\alpha}(xK + K_0)$.

As this is indistinguishable to the adversary and extractor except with probability $\frac{T}{2^{2\lambda}}$, we know that $xK + K_0 \neq \mathcal{O}$ with probability $p - \frac{T}{2^{2\lambda}} - \epsilon$. If $xK + K_0 = \mathcal{O}$, we fail at the n-ISDH problem. Otherwise, since $K = \beta P$ and $K_0 = k_0 \beta P$, we have that $x \neq -k_0 \mod q$ so we can compute

$$(x + k_0)^{-1} R_m = \frac{1}{y + \alpha} \beta P$$

We can define the "owner" of an ID y via **user_IDs**, which is only set once for each y. If **user_IDs**$[y] \notin \mathcal{C}$, this is equivalent to either y having no owner or y belonging to an honest user. This means y was either not called to **AFF_INV_SIGN** at all (for no owner), or we simulated **AFF_INV_SIGN** without calling the group problem oracle (for honest users). In either case, we did not query the group problem for y, so we can return $((x + k_0)^{-1} R_m, y)$ as a solution to the n-ISDH problem.

The probability that the extractor produces a witness for y such that **user_IDs**$[y] \notin \mathcal{C}$ was assumed to be p. The probability that this is true and that $xK + K_0 \neq \mathcal{O}$ is at least $p - \epsilon$. The negligible losses from the simulations give the final probability. □

Theorem 24.6. *Under hardness of the n-ISDH problem and the discrete logarithm problem in group G_1, ALLOSAUR has blind commitment soudness (Definition 10.2.*

Proof. First consider an adversary-first, blame-free accumulator. Suppose an adversary wins the blind commitment soundness game with probability p, then Lemma 23.1 implies that there is someϵ such that there is an extractor that produces a valid witness (n, x, C, R_m) for an ID y with probability at least $p(\frac{p}{T} - \frac{1}{q}) - \epsilon$, where T is the number of random oracle queries, and an algorithm to solve the discrete log problem in G_1 with probability ϵ. Assuming the discrete log problemis hard, then ϵ is negligible, so if p is non-negligible, the extractor has non-negligible chance of succeeding.

From there, Lemma 24.5 shows that if an extractor has a probability p of producing a valid witness (k, x, C, R_m) for an ID y such that y is not corrupt,

then there is some ϵ such that there is a discrete log solver that succeeds with probability ϵ and an n-ISDH solver that succeeds with probability $p - \epsilon - \frac{T}{2^{2\lambda}}$. If n-ISDH and discrete log are both hard, these probabilities must be negligible, so p must also be negligible.

Next, suppose there exists an extractor with probability p of producing a valid witness (k, x, C, R_m) for an ID y such that $y \in \mathcal{S}[m]$ for some $m \geq k$ (i.e., there should be a valid witness for y for that epoch). By Proposition 24.3, this implies an algorithm to solve the n-ISDH problem with probability $p - \text{negl}(\lambda)$, so p must also be negligible.

Thus, an adversary-first, blame-free accumulator is blind commitment sound. By Lemma 22.2 and Lemma 22.4, this implies ALLOSAUR is also blind commitment sound with only negligible loss. □

25 Message Indistinguishability

For indistinguishability, we proceed in two steps: in this first section, we show that the actual content of private user messages is indistinguishable and thus we can define hybrids where this data is entirely removed, leaving only metadata. We then show that the metadata gives precisely the anonymity loss that we specified in the protocol.

Our first hybrid is a no-verify accumulator, Definition 25.1, where proof messages contain no data. We show that that this hybrid is no more distinguishable than the original in Proposition 25.2. The second hybrid is a no-update accumulator, Definition 25.3, where users do not send any shares for their updates and instead compute them locally using the public data of the accumulator. Proposition 25.4 shows that this is also no more distinguishable. Combining these two hybrids leaves no data except metadata.

Definition 25.1. A no-verify accumulator game proceeds identically to the original accumulator game, except that **OBSERVER_START_PROOF** and **OBSERVER_CHECK_PROOF** are the empty function.

The no-verify version of ALLOSAUR is the same as the original, with the following exceptions:

1. **VER** returns immediately after Line 15
2. During **PROVE**, the value of proof is computed simply as 1 if the user has a valid witness, and 0 otherwise.

Obviously this no-verify accumulator is no longer sound, but it provides more user indistinguishability: there is no user-specific data in a verification.

Proposition 25.2. *For every indistinguishability adversary against the original accumulator, there is an indistinguishability adversary against the no-verify accumulator with an additive loss in advantage of $\frac{v(T+P)}{2^{2\lambda}}$, where T is the number of random oracle queries the original adversary makes, v is the number of verifications performed, and P is the number of honest user player pseudonyms.*

Proof. We construct a no-verify adversary $\mathcal{A}'$, which honestly forwards all messages between the original adversary $\mathcal{A}$ and an honest execution of the no-update accumulator. However, $\mathcal{A}'$ must simulate any expected verify messages.

When a user sends a proof message with a proof of 1, $\mathcal{A}'$ simulates the verification proof using Lemma 22.1 by simulating certain random oracle queries from $\mathcal{A}$, which is undetectable with probability at most $\frac{vT}{2^{2\lambda}}$, where v is the number of positive verifications.

If the intercepted Ver_v message had a proof of 0, $\mathcal{A}'$ simply sends $\perp$ as the proof.

By Proposition 20.16, this follows the same distribution as the proofs sent by valid users, so this is undetectable to the original adversary.

VER sends no messages after Line 6, so the messages will be indistinguishable, but the result of the verification might change behaviour. However, the only way in which a verification result can change execution of the accumulator is if it is incorrect and triggers **ACCUMULATOR_FAIL**. By Theorem 21.9, $\mathcal{A}$ cannot cause this with more than non-negligible probability. □

We next define a no-update accumulator.

Definition 25.3. A no-update accumulator is identical to the original accumulator.

The no-update version of ALLOSAUR differs from the original in the following ways:

1. During **GET_UPDATE**, users do not execute Line 11 (they do not post the shares of their secret).

2. User players will use the data posted in the public message board to simulate the servers response in **GET_UPDATE**, and will wait one round to do so.

Proposition 25.4. *For every indistinguishability adversary against a no-verify accumulator, there is an indistinguishability adversary against the no-update no-verify accumulator with no loss in advantage.*

Proof. Let $\mathcal{A}$ be a no-verify adversary. To construct a no-update no-verify adversary $\mathcal{A}'$, we have $\mathcal{A}'$ forward all messages as expected between the real accumulator and $\mathcal{A}$, though $\mathcal{A}'$ must simulate the update messages. The expected messages are formed by a t-out-of-n secret sharing scheme, so since $\mathcal{A}$ controls fewer than t servers, the shares they receive are uniformly random. Thus, $\mathcal{A}'$ can simply send uniformly random elements to the adversary. Since it sees the number of epochs to update, it can compute the values of m, k, and d as in **GET_UPDATE** so we know precisely how many uniformly random elements to send.

This is only distinguishable from the expected messages if $\mathcal{A}$ sends the incorrect data back to the user. In this case $\mathcal{A}$ will be detected and blamed by Proposition 20.2 (if the user expects a valid witness) or by Proposition 20.14 (if the user expects an invalid witness). If $\mathcal{A}'$ fails to properly simulate in either case, there is no loss in advantage, since the original adversary $\mathcal{A}$ would fail. □

26 Metadata Indistinguishability

26.1 Outline

The last section showed that the content of the messages is completely anonymous, and all that remains is metadata. By metadata, we mean all extra information: which messages are sent to which other users, which public messages are sent, and any non-obfuscated data.

Recall that users use a single pseudonym for each user function they call (**VER**, **WIT**, **UPDATE**, or **PROVE**) and ignore unrelated messages to the same pseudonym. An anonymity set is defined for each pseudonym, containing a set of user IDs. Informally it should represent all the users who, if asked to perform one of these user functions, would respond in a particular way.

To capture this we first define notion of a *message response pattern*, which is intuitively exactly what it says: the pattern of messages that a user will send in response to any messages they receive. These are indexed by pseudonyms, so we want to show that all users in the anonymity set of some pseudonym would produce the same message response pattern as that pseudonym. Each of the user functions begins after being sent an initial function message, so our first task is to prove that the message response patterns in response to each of the four functions above will be the same for users in the same anonymity set.

Proposition 26.2 shows this for **VER**, which is straightforward as verification neither uses nor modifies any internal user data except the public accumulator value.

Other messages do depend on internal user data, and specifically on the epoch of the user's latest witness, since if a user updates, they will update from that epoch and must send the value of the epoch to the servers to get their update. First we show that the observer program correctly tracks a user's internal epoch in Proposition 26.3. Then we show that user response patterns to proof requests are the same, under different conditions. First, Proposition 26.4, if there is no consensus accumulator for the epoch of a proof request a user will not respond at all. All users receiving such a request vacuously send the same messages. The next case is that a user performing an update for their proof may need to recurse during **RETURN_UPDATE**, so we characterize precisely when this happens in Proposition 26.5: when the user is deleted but also re-added to the accumulator between the initial and final epochs of the update.

The observer program assumes a recursive user is completely de-anonymized, so we only need to show that users which do not recurse, and which had the same witness epoch to begin with, send the same update messages after being requested to prove Proposition 26.6. We then show that the observer program correctly identifies these different cases and assigns users in each case to distinct anonymity sets, and thus if two users are in the same anonymity set, they respond in the same way to any proof requests Proposition 26.7.

Next we need the same result for update requests. Again we show in Proposition 26.8 that if users are not deleted and re-added, and hence do not recurse,

they send the same update messages. An added difficulty for updates is that they change the user's internal state. One can see the reason for our design decision to use ephemeral updates during the verification protocol. Our observer program thus restricts the anonymity set for an update to only include users which received the exact same update request in the same round, and excludes users who received multiple update requests to different pseudonyms. Proposition 26.9 describes fully what users remain in the same anonymity set of a pseudonym which received an update request. This characterization gives us Proposition 26.10, stating that if a pseudonym receives an update request, not only do users in its anonymity set respond in the exact same way to an update request, they also update to the same epoch in that round.

For witness requests, there is little to prove because a pseudonym that receives a witness request will post a message $\mathsf{Add}(y, \mathsf{proof})$ containing the ID y of the user owning that pseudonym. This fully de-anonymizes the pseudonym. It also changes the user's state, so we exclude any users who have not finished obtaining their first witness from all other anonymity sets.

This covers all type of function requests, so we can prove that any two users in the same anonymity set have pseudonyms that induce the same message response pattern. Inuitively, this means an adversary sending to one of these pseudonyms can't tell which user the pseudonym belongs to, because all the users respond in exactly the same way.

To finish the proofs, we reduce to a hybrid in Proposition 26.13 to rid ourselves of the edge case where an adversary guesses a pseudonym before the pseudonym program selects it (pseudonyms are 2λ bits long, so such a guess is nearly impossible).

Since pseudonyms are assigned randomly, this means that for every execution where two users are assigned a pair of psuedonyms, there is an equally probable execution where the pseudonyms were swapped. By the construction of our anonymity sets, we know that users in the same anonymity set respond in the same way to messages and they update their internal state in the same way. Thus, pseudonyms can be swapped among users in the same anonymity set, and the resulting transcipt of interactions with the adversary is *exactly* the same. Proposition 26.14 formalizes and proves this concept.

Recall that the indistinguishability game, Definition 10.3, requires an adversary to output two pseudonyms p_1 and p_2. The adversary wins if p_1 and p_2 belong to the same user. Suppose the adversary did win. Since the adversary's output depends only on the messages it receives and its internal state, then in another execution where p_1 was assigned to a different user but the transcript was the same – which exists in every case by the logic above – then in this execution the adversary will *still* output p_1 and p_2, but now they lose.

Counting all the possibilities shows that the adversary's probability of winning depends only on the sizes of the anonymity sets and their intersection, exactly the probability in Definition 10.3 that the adversary must exceed. Hence, unrolling our hybrid reductions, Theorem 26.15 shows that ALLOSAUR is indistinguishable.

26.2 Proofs

Throughout this section, unless otherwise specified, we assume we are working with a no-update, no-verify accumulator.

One of our main tools is to show that pseudonyms in the same anonymity set will respond in the same way to future messages. We use the following definition:

Definition 26.1. A *message response pattern* is a function taking as input a public message board, a pseudonym p, and a private message board for a pseudonym p, which outputs a probability distribution χ on the space of possible sequences of messages, according to the following rule: if an execution of the accumulator produces the message boards given as input (including the implicit round number), then the probability that p posts a sequence of messages $(m_1, \dots)$ in that round is given by χ, where the probability is taken over all executions of the accumulator and that produce the message boards given as input.

What we capture here is that the pseudonym belonging to a user may change what messages it posts based on that other messages are posted; however, if two pseudonyms respond in exactly the same way (equivalent to having the same message response pattern), they will be indistinguishable.

Our proof strategy will be to show that users in the same anonymity set will have the same message response pattern. The actual statement will be slightly more technical. To get there, we show how message response patterns will be the same for each function if users are in the same anonymity set. We divide this by the type of message a user sends.

Verifier messages. The first proposition does not reference anonymity sets, since verifiers are completely anonymous in a verification protocol.

Proposition 26.2. *The message response pattern of all pseudonyms p belonging to honest user players, such that the first message sent to a pseudonym p is a function-type message with function* Ver *and arguments of a pseudonym j and epoch n in round t, are the same.*

Proof. We will explicitly construct the message response pattern of such a user. Any honest user receiving a Ver message as in the proposition statement as the first message to pseudonym p will enter the loop at Line 2 until the first round a consensus accumulator for the epoch n is posted (part of the public message board). Once such a message is posted, they will then send a Prove message to pseudonym j; 3 rounds after that message is posted, they send a random proof commitment. These are distributed uniformly at random. These are the only messages they send.

Honest user players will not respond to any other type of message, since they ignore any subsequent requests to that pseudonym. Since they only depend on the messages sent to p and no user-specific data, the message response pattern is the same for all such users. □

Prover messages. We next want to make the same arguments for psuedonyms receiving a Prove message. However, such users will start an update, and their message response pattern will depend on the epoch of their current most up-to-date witness. Thus, we must prove that the observer correctly tracks this.

Proposition 26.3. *In round t, during execution of* RESTRICT_ANONYMITY *until Line 85, the value of* user_epochs$[y]$ *equals the value of* next_witness *at the beginning of* UPDATE *in round $t+1$. After Line 85, the value of* user_epochs$[y]$ *equals the value of* next_witness *during* PROVE *in round $t+1$.*

Proof. If there exists an honest user with ID y, it means there was no Add message posted before the first Wit message with argument y. The first Wit message prompts the user to set next_witness $= 0$ and also RESTRICT_ANONYMITY() sets user_epochs$[y] = 0$ as well, at Line 35, in the round before this message was posted.

From there, the user will only update next_witness when the first consensus accumulator is posted that contains y, and it will update in exactly that round. This is because the servers execute SIGN before posting the new accumulator, so the user will return from waiting for the "signature" portion of their witness in Line 20 before they return with the other part of the witness. This is also the first round when RESTRICT_ANONYMITY() will update user_epochs$[y]$ to this new value, in Line 7.

Once this happens, a user will update next_witness in each round to the maximum epoch argument passed in a call to UPDATE that corresponds to a consensus accumulator. However, user functions are run by the loop at Line 29 of RUN_PLAYER_HONEST, which iterates through a queue that sorts the functions. The ordering on functions ensures that it calls UPDATE first; thus, the user will not modify next_witness until it returns from any update calls in that round. While this matches how RESTRICT_ANONYMITY will eventually update user_epochs$[y]$, RESTRICT_ANONYMITY will not change user_epochs$[y]$ until Line 85.

Since PROVE runs after the UPDATE changes next_witness by the sorting of queued functions, it will find a value of next_witness that was been increased according to any update requests in that round, which matches RESTRICT_ANONYMITY after Line 85. □

The message response pattern for a proof first depends on whether there is a consensus accumulator, and then if the user was deleted and re-added, and finally based on the user's starting epoch. The next three proofs cover these cases, culminating in Proposition 26.7.

Proposition 26.4. *Let p and p' be two pseudonyms belonging to users y and y' such that*

- *the first message to each pseudonym is a function message of type* Prove *from the same pseudonym j with the same epoch n, sent in round t*

- $|\mathcal{A}| < |\textbf{servers}|$ *in round* $t+1$

Then p *and* p' *have the same message response pattern.*

Proof. If $|\mathcal{A}| < |\textbf{servers}|$, then a user receiving a Prove message will ignore it and not respond to any other messages to that pseudonym. Thus, trivially, the message response pattern is the same: such a user will not send any messages from that pseudonym. □

Next we consider two users asked to prove, where they will perform the same update over epochs where neither was deleted and re-added. We clarify the precise conditions for this to occur:

Proposition 26.5. *A user with ID* y *calling* **RETURN_UPDATE**$(*, n_1, n_2, *)$ *will recurse if and only if there is some* k *with* $n_1 \leq k < n_2 - 1$ *such that* $\mathcal{S}[k+1] \setminus \mathcal{S}[k]$ *and* k *is the maximum integer satisfying these properties.*

Proof. We prove the "only if" first. Inspecting **RETURN_UPDATE**, it will only recurse if **GET_UPDATE** returns $d = 0$ or it started with an invalid witness, and the latest valid witness posted on the public message board has an epoch less than n_2. When updating from epoch k to $k+1$, honest server players will only call **AFF_INV_ACC**, which posts the new witness, during **ADD**$(y, *)$ if $y \notin$ **aux**, equivalent to $y \notin \mathcal{S}[k]$. If they do, once they finish, $y \in \mathcal{S}[k+1]$.

For the converse, if $y \in \mathcal{S}[k+1] \setminus \mathcal{S}[k]$ for some $n_1 \leq k < n_2 - 1$ then $y \notin \mathcal{S}[k]$. Then either $y \notin \mathcal{S}[j]$ for all $n_1 \leq j \leq k$, in which case the user's starting witness was invalid and **RETURN_UPDATE** will call **FIND_WITNESS**, or y was in some previous epoch but was deleted, in which case **GET_UPDATE** returns the value 0 and similarly **RETURN_UPDATE** will call **FIND_WITNESS**.

By assumption, the honest server players added y in epoch $k+1$, so they posted a witness for y for epoch k. **FIND_WITNESS** will find the witness posted for the maximum such k; by assumption, $k < n_2 - 1$ so the obtained witness has epoch less than n_2, so **RETURN_UPDATE** must recurse. □

We will say that the epochs (n_1, n_2) matching the conditions of Proposition 26.5 are a *recursive update* for user y. Epochs which are a recursive update for one y may not be for other values y'.

Proposition 26.6. *Let* p *and* p' *be two pseudonyms belonging to users* y *and* y' *such that*

- *the first message to each pseudonym is a function message of type* Prove *from the same pseudonym* j *with the same epoch* n*, sent in round* t
- *at the beginning of execution of* **RESTRICT_ANONYMITY** *in round* t*,* **user_epochs**$[y] =$ **user_epochs**$[y']$
- *either both* y *and* y' *are in* $\mathcal{S}[n]$ *or neither is*

- *The epochs* $((\textbf{user\_epochs}[y], n)$ *are not a recursive update for either* y *or* y'*.*

Then p *and* p' *have the same message response pattern.*

Proof. If $|\mathcal{A}[n]| < |\textbf{servers}|$ in round $t+1$, the message response pattern is the same by Proposition 26.4. Thus, assume $|\mathcal{A}[n]| = |\textbf{servers}|$ in round $t+1$.

In this case, both users will begin execution of **PROVE**, skip the early return at Line 5. **PROVE** prompts users to post two kinds of messages: update requests, and proofs.

Update messages. By assumption on **user_epochs** and Proposition 26.3, **next_witness** is identical for the two users. Thus, if this is greater then or equal to n, neither user will post any update messages. If it is less than n, both users will post Upd_s messages in round $t+1$, with epoch arguments of **next_witness** and n.

For possible remaining update messages, we assumed that the epochs are not a recursive update for either player. Thus, neither recurses during **RETURN_UPDATE** and they will only send one set of update messages. As this is a no-update accumulator, these messages are the same.

Proof messages. Exactly four rounds after receiving the Prove message (since **RETURN_UPDATE** takes at most 4 rounds by Proposition 20.10, and **PROVE** waits 4 rounds after sending it), both users will be ready to respond to a proof challenge. If they receive a proof challenge, they will respond one round later, with either a 1 or 0 depending on whether they are in $\mathcal{S}[n]$ or not (by Proposition 20.16). Since both users are either in or out of the accumulator, they send the same message here. □

A final case is when a user is deleted and re-added (i.e., the given epochs are a recursive update). This is actually completely de-anonymizing. If a user is given recursive update epochs, then their recursive update will proceed from the epoch they were added to the final epoch. Since only one user is added in each epoch, the epoch uniquely identifies the user. Thus, there is no need to state anything about the message response pattern; we do not need to show it is the same for any two users.

We can now summarize these propositions:

Proposition 26.7. *Suppose* p *is a pseudonym such that the first message to it is a function message of type* Prove*, and the only message sent to* p *in that round. Then for all users* $y \in \textbf{anon\_sets}[p]$*, if an unused pseudonym* p' *of* y *received the same message in the same round, then* p' *would have the same response pattern.*

Proof. Suppose the Prove message sent to p was sent in round t from verifier j with epoch argument n.

If there is no consensus accumulator in round t for epoch n, the message response pattern is empty for all users. Hence, the result holds for this case.

If there is a consensus accumulator for epoch n, we show that all unused psuedonyms of all $y \in \textbf{anon\_sets}[p]$ satisfy (almost all) the conditions of Proposition 26.6. First, **RESTRICT_ANONYMITY** removes all pseudonyms that have already received a message or receive more than one message in this round, so we know this is not true for any of the pseudonyms of these y. It sets $\textbf{anon\_sets}[p]$ to a subset of $\mathcal{Y}_{upd}$, calculated in Line 95 to contain only y with the $\textbf{user\_epochs}[y]$ constant, satisfying the second condition. The set $\mathcal{Y}_{del}$, constructed in Line 96, is precisely the users given recursive update epochs, and this is removed from $\textbf{anon\_sets}[p]$, satisfying the last condition. If the owner of p is in $\mathcal{Y}_{del}$, their anonymity set is restricted to a singleton set, from which the result follows trivially.

This means the unused pseudonyms of all $y \in \textbf{anon\_sets}[p]$ almost satisfy the conditions of Proposition 26.6, except perhaps they are not all either in $\mathcal{S}[n]$ or not. However, inspecting **PROVE** shows that the only difference in response pattern occurs in the proof messages sent in response to challenge messages, but as soon as a proof message gets sent (and before the adversary can see it),**RESTRICT_ANONYMITY** at Line 19 restricts the anonymity set to only users either in the accumulator or not, so after this point, for any y still in the anonymity set, its pseudonym has the same message response pattern of p. □

Update messages. Now we show the same result for updates. This is similar to the proof case, since proofs send the same pattern of update messages; however, proofs do not change any aspect of the user's internal state, while updates do. This requires extra care.

The next proposition is an analogue of Proposition 26.6.

Proposition 26.8. *Let p and p' be two pseudonyms belonging to users y and y' such that*

- *the first message to each pseudonym is of a function message of type* Upd_u *with the same epoch n, sent in round t*
- *at the beginning of execution of* **RESTRICT_ANONYMITY** *in round t,* $\textbf{user\_epochs}[y] = \textbf{user\_epochs}[y']$
- *the epochs* $(\textbf{user\_epochs}[y], n)$ *are not recursive update epochs for either y or y'*

Then p and p' have the same message response pattern.

Proof. In this case both users will begin execution of **UPDATE**. If there is no consensus accumulator, then p and p' both have empty, and thus equal, message response patterns.

By assumption on **user_epochs** and Proposition 26.3, **next_witness** is identical for the two users. Thus, if this is greater then or equal to n, neither user will post any update messages. If it is less than n, both users will post Upd_s messages in round $t+1$, with epoch arguments of **next_witness** and n.

By the third assumption, neither user recurses, so each will only send one set of update messages. As this is a no-update accumulator, these update messages are the same. Neither pseudonym will respond to any other messages. □

To summarize the behaviour of **RESTRICT_ANONYMITY**, the following proposition characterizes anonymity sets after update messages:

Proposition 26.9. *If the first and only message sent to a pseudonym p in round t is an* Upd *function message, if the anonymity set for p contains more than one ID, then it is restricted to users y such that:*

- *y possesses a pseudonym p' such that p' received only one message in round t, which was the first message to p'*
- *the only* Upd *function message sent to any pseudonym of y was sent to p'*
- **user_epochs**$[y]$ *is constant for all y in the anonymity set*
- *the epoch of the* Upd *function message (denote it by n) is constant for all y in the anonymity set*
- (**user_epochs**$[y], n$) *are not recursive update epochs for y*

Proof. For the first point, pseudonyms receiving multiple messages in round t or which had previously received messages are removed from the set of messages.

Second, if multiple update messages are sent, then the message (and user) are moved to $\mathcal{MY}_{dup}$, where the anonymity sets are restricted to singleton sets.

Third, if the epochs are , then at Line 66, the anonymity set of the pseudonym of the message sent to y is restricted to a singleton set. Otherwise, the anonymity set is restricted to all other $y \in \mathcal{MY}_i$.

Third, the partitioning of messages at Line 56 ensures all $m \in \mathcal{MY}_i$ have the same epoch argument.

Fourth, the partition at Line 57 ensures that all starting user epochs are the same for $y \in \mathcal{MY}_i$, which contains the anonymity set.

Finally, the set $\mathcal{Y}_{del,i}$ constructed in Line 59 contains all $y \in \mathcal{MY}_i$ for which the epochs are recursive update epochs. Users in this set get a singleton anonymity set, and for any remaining users the set $\mathcal{Y}_{del,i}$ is removed from the anonymity set of their pseudonym, giving the last result. □

What we show now is that not only do updating pseudonyms have the same response pattern, they also update to the same epoch in the same round. This ensures that the changes to internal state are the same across users in the same anonymity set.

Proposition 26.10. *Suppose p is a pseudonym such that the first message to it is a function message of type* Upd_u*, and this is the only message sent to p in that round. Then either:*

1. *p has an empty response pattern*

2. *all users $y \in$ anon_sets$[p]$ own a pseudonym p' which received an identical update function message in the same round and for which p' has the same response pattern.*

Proof. Suppose the Upd_u message sent to p was sent in round t with epoch argument n.

If there is no consensus accumulator in round t for epoch n, the message response pattern for p is empty.

If anon_sets$[p]$ contains only one ID, then the result holds trivially.

From there we use Proposition 26.9, whose conclusions imply the premises of Proposition 26.8, showing that they will have the same response pattern.

If y does not have an empty response pattern, then it will start an update in round t. If $y' \in$ anon_sets$[p]$, then by Proposition 26.9, y' also received an update message with the same epoch as y received. Neither of them received a second message to those pseudonyms, neither received another update message in that round, and both had the same starting value of user_epochs (and hence of next_witness). Thus, they will both start UPDATE to the same epoch. □

Witness issuance messages. The final type of message is a Wit message, but these are trivially de-anonymizing since a user executing WIT will post an Add message that includes their ID y as an argument. These do modify state, which is why we remove all users that do not yet have a valid witness from the anonymity sets of other messages.

Summary. Thus, we have all the components necessary to show indistinguishability of message response patterns.

Proposition 26.11. *For any pseudonym p, let t be the first round when any messages are send to p and let m be the set of all messages sent in round t. For any user $y \in$ anon_sets$[p]$, if any unused pseudonym p' of y received the same messages m in round t, then p' would have the same response pattern.*

Proof. First, notice that anon_sets$[p]$ is always restricted to honest users.

If m contains more than one message, this creates an empty response pattern for any user.

If m contains only one message, then if it is not of type Prove, Ver, Upd_u, or Wit, any user receiving this message will ignore it, again creating an empty response pattern.

For the case of Prove, Ver, and Upd_u, Propositions Proposition 26.2, Proposition 26.6, and Proposition 26.8, respectively, prove the result.

For Wit, the loop in Line 30 restricts the anonymity set for the receipient to a unique y, so the result holds trivially. □

A slightly annoying case we need to deal with is the probability that an adversary will guess a pseudonym. Thus we define a new hybrid:

Definition 26.12. A *no-guess* accumulator is an accumulator in which the control program checks the destination pseudonym of every message that is posted. If the destination pseudonym belongs to a user player, but it was not given as output by **GAME_DEANONYMIZE** before the round in which it was posted, the control program aborts on the ID of the sender of the message and the accumulator game ends.

Proposition 26.13. *Any adversary against a no-verify no-update accumulator implies an adversary against a no-guess, no-verify, no-update accumulator, with at most a* $\frac{PM}{2^{2\lambda}}$ *loss in advantage, where* P *is the total number of pseudonyms used and* M *is the total number of messages sent.*

Proof. We will use the no-verify no-update adversary directly against the no-guess no-verify no-update accumulator.

Inspecting the logic of how honest user players obtain pseudonyms, all of their pseudonyms are outputs of **GAME_DEANONYMIZE**. Thus, the only way to trigger the new abort condition is if a message is sent to a pseudonym and later that same pseudonym is assigned to a player. Since pseudonyms are chosen randomly, the probability of a new pseudonym matching an existing message is $\frac{M}{2^{2\lambda}}$. Taking the union bound over all pseudonyms produced gives the result. □

The next proposition is almost enough for our security definition. We show that when the pseudonym program assigns two pseudonyms, if the same users are in the anonymity set of each one, then the pseudonyms could be swapped without the adversary noticing. That is, since pseudonyms are assigned randomly, there is an equally probably execution where the pseudonyms *were* assigned to different users, and we show that as long as the users are in the same anonymity set, the adversary's view of each execution is exactly the same.

Proposition 26.14. *At any point in execution of a no-guess, no-update, no-verify accumulator, except during* **RESTRICT_ANONYMITY**, *if* y *owns pseudonym* p *and* $y' \in$ **anon_sets**$[p]$, *then there is an equally probable execution of the accumulator such that* p *was assigned to* y' *(taken over the randomness of the control programs and honest users) that produces the same transcript of interactions with the adversary.*

Proof. Consider a user y with pseudonym p and some $y' \in$ **anon_sets**$[p]$. Consider the call to **GAME_DEANONYMIZE** when p was assigned, which occurred before p received any messages since this is a no-guess accumulator. The argument to **GAME_DEANONYMIZE** included both y and y', so suppose y' was given pseudonym p'.

We now consider an execution where the randomness of the accumulator and honest parties is different such that user y is given p' and user y' is given p. This is identical from the adversary's view when **GAME_DEANONYMIZE** returns, since the pseudonyms are sorted before being returned. Then user y' will receive the first message sent to pseudonym p, and by Proposition 26.11,

pseudonym p will respond in the same way in this execution as in the last execution.

If the message was Wit, then $y = y'$ because the anonymity set has only one element; if the message was Prove or Ver, the user does not modify any global variables, so execution for the two players will proceed indistinguishably.

If the adversary sends Upd_u to pseudonym p, then by Proposition 26.9, y' was also sent an update to the same epoch since they are in the same anonymity set. By assumption **anon_sets**$[p]$ has at least 2 elements, so no other update message was sent to y or y' in the same round, so both will respond the same way. Thus, any future interaction is also indistinguishable: both users will respond to update and proof requests in the same way, since they both updated to the same epoch. □

Now recall our formal definition of indistinguishability, Definition 10.3. An adversary outputs two pseudonyms p_1 and p_2, and they win if these two pseudonyms belong to the same user.

Theorem 26.15. *ALLOSAUR is indistinguishable.*

Proof. First we argue that the no-guess, no-update, no-verify hybrid is indistinguishable. Suppose an indistinguishability adversary outputs two pseudonyms p_1 and p_2; let $A_i =$ **anon_sets**$[p_i]$. We fix the adversary's randomness, and vary the accumulator's randomness but condition on all executions where the transcript of interactions was identical with this one. Since the input to the adversary is identical, then the adversary outputs p_1 and p_2 in all of them.

By Proposition 26.14, for each $y \in A_i$, there is an equally probable execution with the same transcript where p_i belonged to y. We will label sets of executions by (y_1, y_2), the IDs of which users had pseudonym p_1 and p_2. As these are equally probable, the probability of each pair (y_1, y_2) is $\frac{1}{|A_1| \cdot |A_2|}$.

However, the adversary only wins in executions where $y_1 = y_2$. The number of such executions is only $|A_1 \cap A_2|$. Thus, the probability of the adversary winning is $\frac{|A_1 \cap A_2|}{|A_1| \cdot |A_2|}$, exactly what is needed. Since this does not depend on the transcript, only the size of the anonymity sets of the adversary's answer, we can sum over all executions with the same size anonymity sets and prove perfect indistinguishability.

Finally, we use Proposition 25.2, Proposition 25.4, and Proposition 26.13 to add only a negligible increase in the adversary's advantage, thus showing that ALLOSAUR is indistinguishable. □

27 Anonymity

Our definition of indistinguishability is somewhat insufficient, since we leave it up to the protocol to specify the anonymity function. A simple way to produce an "indistinguishable" accumulator is to make the anonymity set of each message equal a singleton set containing the message's sender. Such an accumulator is completely non-anonymous, however!

Thus, we will now show some conditions in which our protocol offers actual anonymity. We cannot guarantee much more than this, since there are expected patterns of behaviour where anonymity is degraded (for example, if all but one user ID is deleted from the accumulator).

We say that a user is "established" in round t if their ID has been added to the accumulator at some previous point, and a function message of type Upd_u with epoch argument n was posted to that user at least 5 rounds previous, there exists a consensus accumulator for epoch n, and n is greater than the last epoch in which they were added or deleted from the accumulator.

We restrict the messages the adversary sends in the following ways, for some parameter j:

- The adversary does not send more than one function message to any pseudonym
- If an adversary sends at least one established user ID y in its call to **GAME_DEANONYMIZE**, they must send the IDs of least $j-1$ other established users in that call.
- There must be at least j users added to the accumulator before the first Upd_u message is sent. After that, there must be at least j established users if any Upd_u or Ver_u message is sent.
- If one Upd_u message is sent in round t with argument n to pseudonym p, then in round t exactly one pseudonym for every user must also receive an Upd_u message with argument n in round t.
- If one Ver message is sent in round t with epoch n to verifier i with pseudonym p, then (a) p must be the output of a call to **GAME_DEANONYMIZE** with at least j established users, such that either all users are in $\mathcal{S}[n]$ or none are; (b) i must receive one Ver_u message in round t for each pseudonym p' returned by **GAME_DEANONYMIZE**, all with epoch n.
- If the adversary sends a Prove message in round t with epoch n to pseudonym p, then (a) (a) p must be the output of a call to **GAME_DEANONYMIZE** with at least j established users, such that either all users are in $\mathcal{S}[n]$ or none are; (b) each pseudonym returned by this call to **GAME_DEANONYMIZE** must receive an identical Prove message from the same adversarial pseudonym in round t.
- If a user is deleted from the accumulator in epoch n, then at least $j-1$ other users must be deleted before the epoch of the next Upd_u message.

We refer to this as an "established-anonymous" adversary.

To ensure that these requirements are not too restrictive, we prove that such adversaries exist and that they can interact in such a way that they create established users.

Proposition 27.1. *An established-anonymous adversary exists, such that at some point in execution, there is at least one established user.*

Proof. The adversary first creates j users with calls to **WIT**. From that point on, whenever the adversary wants to send an update request, it sends the IDs of all users to **GAME_DEANONYMIZE**. It then sends an identical Upd_u message to each pseudonym it receives in return in that round. This satisfis the requirements on update messages, as long as it does not un-establish the first j users. This ensures there is at least one established user.

The adversary can track which users are established because it knows when users were added, and all users receive updates at the same time.

When the adversary wants to send Ver or Prove messages, it selects an epoch n selects a subset of at least j established users, for which either all of them are in $\mathcal{S}[n]$ or none of them are, and sends their IDs to **GAME_DEANONYMIZE**. It receives a set of pseudonyms P in return. For Ver messages, it then picks any other subset of users and sends their IDs to **GAME_DEANONYMIZE**. It picks one or more IDs from what is returned, and for each one, it picks an epoch number n and sends one Ver message for each $p \in P$, all with epoch n. For Prove messages, it sends one Prove message to each $p \in P$, all with epoch number n.

The adversary can add users at will. If it deletes users, it will always ensure that it can remove at least j users while still leaving j established users in the accumulated set. It will not make any update requests until the the users are deleted. □

Proposition 27.2. *Against an established-anonymous adversary, at the beginning of each round, all established users have the same value of* **next_witness**.

Proof. The value of **next_witness** is set to the maximum epoch argument of Upd_u every time an established user receives such a message. Thus, once a player is established, they remain in sync, because all users receive the same update messages in the same round. A player is only established after their first update message once they are added to the accumulator, and this will update **next_witness** to match the update epoch of all the other established users. □

Proposition 27.3. *Against an established-anonymous adversary, any update request to an established user does not have recursive update epochs.*

Proof. By definition, an established user has previously had an update request to some epoch n which is greater than the last epoch when the user was added to or deleted from the accumulated set. By Proposition 20.10 this means they have a witness for epoch n, so any update to n' has update epochs (n, n'), which are not recursive update epochs. □

Proposition 27.4. *Against an established-anonymous adversary, all messages from established users have anonymity sets of size at least j.*

Proof. First we consider update messages from established users. To begin, they come from pseudonyms p that were returned by **GAME_DEANONYMIZE** called on the set of all users, so the anonymity set of p at that point has size at least j. An established-anonymous adversary sends only one function message to p, so we consider the different cases.

For Wit messages, if the user has already received a Wit message, they will ignore any future Wit messages and post nothing. If they have not received a Wit message, they are not an established user.

For an Upd message to an established user's pseudonym p, we step through all restrictions of update message anonymity sets in **RESTRICT_ANONYMITY**. Established-anonymous adversaries do not send to the same pseudonym twice, and send only one update message to each user in a round (if they send any), meaning $\mathcal{MY}_{upd}$ contains a message to each honest user. Further, since the adversary sends the same epoch argument, $\mathcal{MY} = \mathcal{MY}_{upd}$ during the only iteration of the loop at Line 56.

No established user gets recursive update epochs by Proposition 27.3, established users must have a valid witness of epoch greater than 0, and by Proposition 27.2 they start with the same epoch. Thus in Line 71, all established users are in **anon_sets**$[p]$. We required that the adversary only send Upd messages when there are at least j established users, giving the result for p.

If an established user sends a Prove message, the anonymity set of this pseudonym contains all the IDs given to **GAME_DEANONYMIZE**, as Prove messages *sent* by honest users do not restrict their anonymity set at all.

Users only send proof messages in response to Prove messages, which was sent by either a corrupt or honest player. The requirements on the adversary ensure that if an established user's pseudonym p receives such a message, p was returned by **GAME_DEANONYMIZE** with an argument of at least j established players, so the anonymity set starts with at least j established players. Again checking how the anonymity set is restricted, it is first restricted to all users that have had some valid witness at Line 91, but all established users have had valid witnesses at some point. Then Line 95 restricts to users that start with the same witness epoch, but Proposition 27.2 shows all established users have the same epoch. Finally, by Proposition 27.3, established users do not recurse, so the anonymity set is set in Line 101. Thus, it contains all established users given by **GAME_DEANONYMIZE**, of which there are at least j.

When these same established users send a proof message, their anonymity set is restricted to all users that are either in the accumulator or not for that epoch. By assumption, this is true for all these established users, so the anonymity set does not change in size. □

We emphasize here that in our definition of anonymity sets, anonymity can "heal". This means that if a user loses anonymity in one round – meaning an adversary can associate specific messages to that user – they can regain anonymity in later rounds, such that future messages can no longer be associated to them. Even though being added to or deleted from the accumulator can

drastically reduce user anonymity, the anonymity is quickly regained as the accumulator progresses.

In our established-anonymous accumulator, one can see that if an established user is deleted, they are no longer established, but after they are re-added and an update is called, they become established again.

28 Inversion Symmetric Diffie-Helman Problem

Here we prove the security of our new group assumption, the n-ISDH problem, in the generic group model. Recall the definition:

Definition 28.1. Let G_1, G_2, and G_T be groups with a type-3 pairing from $G_1 \times G_2$ to G_T. Given G, $\lambda G, \gamma G \in G_1$, $\tilde{G}, \lambda\tilde{G} \in G_2$, and query access to a function $f : (y, Q) \mapsto \frac{1}{\lambda+y}Q$ for $Q \in G_1$, compute

$$\left(\prod_{i=1}^{k}(\lambda + y_i)\gamma G, y_1, \ldots, y_k\right)$$

such that at least one value of y appears in the list $y_1, \ldots, y_k$ at least one more time than we queried it to f.

We assume opaque encodings $\alpha_1, \alpha_2, \alpha_T : \mathbb{Z}_q \to \{0,1\}^*$, which encode G_1, G_2, and G_T, respectively, such that $\alpha_1(a)$ encodes aP, $\alpha_2(a)$ encodes $a\tilde{P}$, and $\alpha_T(a)$ encodes $e(P, \tilde{P})^a$.

An addition oracle $M(\alpha(a), \alpha(b))$ returns $\alpha(a + b)$; a scalar multiplication oracle $S(a, \alpha(b))$ returns $\alpha(ab)$; a pairing oracle $P(\alpha_1(a), \alpha_2(b))$ returns $\alpha_T(ab)$.

Theorem 28.2. *In the generic group model for $|G_1| = |G_2| = |G_T| = q$, an adversary making Q queries to the group oracle and n queries to the affine inversion oracle f succeeds in solving the n-ISDH problem with probability at most*

$$\frac{n+2}{q-1}\frac{(Q+n+6)^2}{2} + O\left(\frac{Q^2 n}{q} + \frac{n^3}{q}\right)$$

Proof. A simulator $\mathcal{S}$ will interact with $\mathcal{A}$. It maintains three lists $L_1 = \{(F_{1,s}, \alpha_s)\}$ for $s = 0$ to $S_1 - 1$; $L_2 = \{(F_{2,s}, \beta_s)\}$ for $s = 0$ to $S_2 - 1$, and $L_T = \{(F_{T,s}, \gamma_s)\}$ for $s = 0$ to $S_T - 1$. $F_{i,s}$ is a rational function of polynomials in $\mathbb{Z}_q[x, z]$. Then the strings $\alpha_s, \beta_s, \gamma_s$ will be the strings that $\mathcal{S}$ outputs, so that $\alpha_s = \alpha(F_{1,s})$ (similarly for β, γ).

To start, $\mathcal{S}$ chooses random strings for α_0, α_1, and sets $F_{1,0} = 1$, $F_{1,1} = x$, and $F_{1,2} = z$. It also chooses random strings β_0 and β_1 and γ_0 and sets $F_{2,0} = F_{T,0} = 1$ and $F_{2,1} = x$. All the strings get sent to $\mathcal{A}$.

For addition in group G_i for $i \in \{1, 2\}$ or multiplication for $i = T$, $\mathcal{A}$ sends α_s and α_t. Then $\mathcal{S}$ finds the indices j, k where α_s and α_t are in its list (it returns $\perp$ if no such indices exist), and $\mathcal{S}$ computes $F = F_{i,j} + F_{i,k}$. If F is in L_i (as a polynomial), then $\mathcal{S}$ returns the associated string; otherwise, it chooses a random string α and adds a new pair (F, α) to L_i before returning α

to $\mathcal{A}$ and incrementing S_i by 1. Subtraction is the same process, except with $F = F_{i,j} - F_{i,k}$.

For scalar multiplication (exponentiation in G_T) in G_i, $\mathcal{A}$ inputs α_s and $a \in \mathbb{Z}_q$. $\mathcal{S}$ finds the index j for $\alpha_s \in L_i$, returning $\perp$ if it's not there, and computes $F = F_{i,j}a$. It checks for $F \in L_i$, and returns the associated α if it is. otherwise, it adds (F, α) for a random string α, increments S_i, and returns α to $\mathcal{A}$.

For pairings, $\mathcal{A}$ inputs α and β. Then $\mathcal{S}$ finds i such that $\alpha_i = \alpha$ and j such that $\beta_j = \beta$ (returning $\perp$ if there are no such indices). It computes $F = F_{1,i}F_{2,j}$ and checks for $F \in L_T$. If it finds F, it returns the associated string. Otherwise, it generates a random γ, adds (F, γ) to L_T, increments S_T and outputs γ.

For the affine inverse oracle, $\mathcal{A}$ inputs α and y. $\mathcal{S}$ finds the index i so that $\alpha_i = \alpha$, then sets $F = \frac{F_{1,i}}{x+y}$ and checks if $F \in L_1$; if so, it returns the associated α. Otherwise, it selects a random α and adds (F, α) to L_1, increments S_1 and outputs α.

We argue that if Y is the **multi**set of all y_i that the adversary has queried (i.e., if the adversary queries twice with a specific y, it appears twice in Y), then every polynomial $F_{1,s}$ will have the form $\frac{p_1(x)+p_2(x)z}{\prod_{y_i \in I}(y_i+x)}$ for some $I \subseteq Y$, and the degrees of $p_1(x)$ is at most $|I|+1$ and the degree of $p_2(x)$ is at most $|I|$. This holds by induction: It is certainly true for the initial elements sent to $\mathcal{A}$. It holds after a scalar multiplication. For an addition, we can see that

$$\begin{aligned}
&\frac{p_1(x) + p_2(x)z}{\prod\limits_{y_i \in I_1}(y_i + x)} + \frac{r_1(x) + r_2(x)z}{\prod\limits_{y_i \in I_2}(y_i + x)} \\
=&\frac{(p_1(x) + p_2(x)z)\prod\limits_{y_i \in I_2 \setminus I_1}(y_i + x) + (r_1(x) + r_2(x)z)\prod\limits_{y_i \in I_1 \setminus I_2}(y_i + x)}{\prod\limits_{y_i \in I_1 \cup I_2}(y_i + x)}
\end{aligned}$$

Here the union of two multisets is defined by taking each entry the maximum number of times it appears in either multiset. The degree of the left term in the numerator, with no z terms is at most $|I_1| + 1 + |I_2 \setminus I_1| = |I_1 \cup I_2| + 1$, which is the same bound for the degree on the right term. With the z, the degree is at most $|I_1| + |I_2 \setminus I_1| = |I_1 \cup I_2|$

The polynomials $F_{2,s}$ will have degree at most 1, since $\mathcal{A}$ can only access additions and scalar multiplications in G_2.

The polynomials $F_{T,s}$ will have the form $\frac{p_1(x)+p_2(x)z}{q_s(x)}$, where $q_s(x) = \prod_{y_i \in I}(y_i + x)$ and $\deg(p_{1,2}(x)) \leq |I| + \{1, 2\}$. This holds by induction with the same proofs as for $F_{1,s}$, but with a base case that includes a product $F_{1,s}F_{2,t}$, which will increase the degree of the numerator by at most 1.

When $\mathcal{A}$ is finished, it outputs $(\alpha_y, y_1, \ldots, y_k)$. The value α_y must correspond to a functions F_y in L_1 (if not, $\mathcal{A}$ guessed at random and we can add a random exponentiation of some F as (F, α_y) in L_1 to L_1). If $\mathcal{A}$ is successful, we

have $F_y P = z$, where there is some multiset A of values y such that

$$P(x) = \prod_{y \in A} (x + y)$$

We can rearrange to $(p_{1,y}(x) + z p_{2,y}(x))P(x) - q_y(x)z = 0$. We argue that this polynomial is not identically 0. For it to be identically zero, we would need $p_{2,y}(x)P(x) - q_y(x) = 0$. However, by the conditions of solving hte problem, there is some $y \in A$ and some $k \geq 1$ such that $(y + x)^k$ divides $P(x)$ and $(y + x)^{k-1}$ divides $q_y(x)$, but $(y + x)^k$ does not divide $q_y(x)$. Thus, factoring out $(y+x)^{k-1}$, evaluating the polynomial at $-y$ gives $-q_y(-y) \neq 0$. Hence, the polynomial cannot be identically zero.

Hence, there are two ways for the adversary to win: For specific values of x^*, z^*, either the required polynomial $F_y P$ evaluates to z for x^*, or the adversary detects the simulation by noticing two functions $F_{i,t} \neq F_{i,s}$ which are equal when evaluated at x^*, z^* for $i \in \{1, 2, T\}$. We assume that $\mathcal{A}$ wins with probability 1 in either case.

If we evaluate on z^*, then all functions become rational functions of x. If the adversary made a total of n affine inverse queries, then $F_{i,s}(x) - F_{i,t}(x) = 0$ can be rearranged to a polynomial of degree at most $n+1$ if $i = 1$, or 1 if $i = 2$, or $n + 2$ if $i = T$. Thus, for a random challenge x^*, this holds with probability at most $\frac{n+1}{q-1}$ or $\frac{n+2}{q-1}$. Similarly, $F_y(x^*, z^*)p_I(x^*) = z^*$ with probability $\frac{2n+2}{q-1}$, which can be deduced by bounding the degree of $P(x)$ as $n+1$ and $p_{1,y}$ as $n+1$ as well. The probability of success is then the sum over all pairs from each case:

(Notice that $Q + n = (S_1 - 2) + (S_2 - 2) + (S_T - 1)$, since it makes Q group queries, n affine inversion queries, and S_1, S_2 start with 2 elements and S_T starts with 1 element. This means $S_1 + S_2 + S_T = Q + n + 5$, and binomial terms are convex:

$$\begin{aligned}\epsilon \leq & \binom{S_1}{2} \frac{n+1}{q-1} + \binom{S_2}{2} \frac{1}{q-1} + \binom{S_T}{2} \frac{n+2}{q-1} + \frac{2n+2}{q-1} \\ = & O\left(\frac{Q^2 n}{q} + \frac{n^3}{q}\right)\end{aligned}$$

□