Exploring How to Authenticate Application Messages in MLS: More Efficient, Post-Quantum, and Anonymous Blocklistable

Keitaro Hashimoto
National Institute of Advanced Industrial Science and Technology (AIST)

Shuichi Katsumata
AIST & PQShield

Guillermo Pascual-Perez
Institute of Science and Technology Austria (ISTA)

Abstract

The Message Layer Security (MLS) protocol has recently been standardized by the IETF. MLS is a scalable secure group messaging protocol expected to run more efficiently compared to the Signal protocol at scale, while offering a similar level of strong security. Even though MLS has undergone extensive examination by researchers, the majority of the works have focused on *confidentiality*.

In this work, we focus on the *authenticity* of the *application messages* exchanged in MLS. Currently, MLS authenticates every application message with an EdDSA signature and while manageable, the overhead is greatly amplified in the post-quantum setting as the NIST-recommended Dilithium signature results in a 40x increase in size. We view this as an invitation to explore new authentication modes that can be used instead. We start by taking a systematic view on how application messages are authenticated in MLS and categorize authenticity into four different security notions. We then propose several authentication modes, offering a range of different efficiency and security profiles. For instance, in one of our modes, $\texttt{COSMOS}^{++}$, we replace signatures with one-time tokens and a MAC tag, offering roughly a 75x savings in the post-quantum communication overhead. While this comes at the cost of weakening security compared to the authentication mode used by MLS, the lower communication overhead seems to make it a worthwhile trade-off with security.

1 Introduction

1.1 Background

A *secure group messaging* (SGM) protocol allows a group of users to asynchronously communicate in an end-to-end encrypted fashion. The *Messaging Layer Security* (MLS) protocol [12, 20], a recently standardized SGM protocol by the IETF, is a proposal developed in a joint effort by academics and industry for a *scalable* SGM protocol supporting groups with tens of thousands of users. Similarly to the Signal protocol [36, 37, 46], considered the gold standard for two-user SGMs, it offers a strong level of forward secrecy and post-compromise security, limiting the scope of device compromise. The draft versions of MLS are already running in production in Cisco's Webex [65] and RingCentral [74], and other companies, including AWS, Cloudflare, and Google, are planing deployment.[1] Furthermore, with the recent adoption of the Digital Markets Act by the European Union, a standard like MLS is hoped to be a potential solution for the interoperability problem in secure messaging [59].

The security of MLS (and its variants) has undergone extensive examination by researchers during the standardization process, e.g., [4–8, 22, 29, 52, 53, 57, 78], and the protocol has been continuously updated leading up to 20 drafts in total[2] until the issuance of the RFC. The majority of works on MLS have focused on the *confidentiality* of the exchanged messages (or the shared group secret key). In contrast, relatively less attention has been directed towards the *authenticity* of messages, which is often viewed as a means to establish confidentiality.

In MLS, there are two types of messages being authenticated [12, Sec. 2]: *application* and *handshake* messages. While the former carry the actual payloads such as chat texts, the latter carry group operations affecting the group state (e.g., authenticating that user u added a new user v to the group). In this work, we revisit how MLS authenticates application messages motivated by the following two issues.

Issue 1: Heavy Reliance on Signatures. In MLS, every user u has a signature key pair $(\mathsf{vk}_u, \mathsf{sk}_u)$ and signs the application message am for authentication. It further independently encrypts the message am and signature sig_u using a symmetric key encryption scheme whose key is derived from the group secret key to conceal the application message and its identity from the delivery server. The resulting (tuple of) ciphertext ct_u is then sent to the group. We call this mode of authentication *Enc-Sign mode*.[3] The recent work by Hashimoto et

---

[1] https://www.ietf.org/blog/mls-protocol-published/

[2] https://datatracker.ietf.org/doc/rfc9420/

[3] In contrast, handshake messages can be sent in *Sign mode*, where the user simply sends the pair $(\mathsf{m}, \mathsf{sig}_u)$.

al. [53] proposed adding an additional signature $\overline{\mathsf{sig}}_G$ on top of ct_u, using a signing key derived from the group secret key. This mode, called the *Sign-Enc-Sign mode*, is a simple but powerful enhancement of the Enc-Sign mode, allowing to anonymously block outsiders from injecting malicious messages to the group, similarly to Signal's two-user Sealed Senders [60].

While adding signatures provides stronger authenticity guarantees, it comes with an increase in the communication and computational costs. This is currently manageable as MLS uses an EdDSA signature with an overhead of 64 B. However, this overhead is greatly amplified in the post-quantum setting. For instance, the NIST-recommended Dilithium signature is 2.4 KB, a 40x increase to EdDSA signatures. Given that a typical application message contains less than 100 B [47], the overhead has a noticeable effect. We thus view this as an invitation to explore alternatives designs. We note that while handshake messages incur the same overhead when turning to post-quantum security, the effect is marginal as the size of the handshake message is larger, and the rate at which group operations are performed is less frequent compared to sending application messages.

Issue 2: Lack of Formal Model for Authentication. Compared to the comprehensive study of the confidentiality guarantees of MLS, authentication has drawn less attention. This lack of focus on authenticity may lead to unforeseen attacks on MLS that do not contradict confidentiality but still harm the protocol. As an illustrative example, the MLS is prone to abuse from malicious insiders (e.g., [8]). Notice that both Enc-Sign and Sign-Enc-Sign modes conceal the sender from the server. This allows a malicious insider to craft a malformed message and send it to the group. If the signature sig_u included in the ciphertext is malformed, even the group users cannot trace back the sender, meaning that a malicious sender can stealthily repeat the attack. While the users can reject these malformed messages, this can only happen *after* downloading them from the server and processing them. This opens the door for a malicious insider to mount a DoS attack on the group. A similar issue was pointed out by Tyagi et al. [75] for Signal's two-user Sealed Senders [60], who experimentally verified that such an attack can easily drain a recipient's battery in a short period of time.

A formal security model that comprehensively captures these properties allows us to better understand the strengths and limitations of a given authentication mode.

1.2 Our Contributions

In this work, we explore new approaches to authenticate application messages in MLS. Our contribution is explained below in more detail and an overview is provided in Tab. 1.

Formal Model for Authentication. In Sec. 2, we study how application messages are authenticated in MLS and systematically analyze the types of adversaries and threat models needed to be considered. More technically, the core of MLS can be regarded as a combination of two protocols: a *continuous group key agreement* (CGKA) and a *forward-secure payload delivery* (FSPD) protocol [5].[4] The former (resp. latter) handles handshake (resp. application) messages. In this paper, we formalize the authentication guarantees of the application messages handled by the FSPD protocol and introduce four different security notions: unforgeability, anonymity, anonymous blocklisting, and tracing soundness. To the best of our knowledge, this is the first work to put a focus on the authenticity of the application message; previous works on MLS studied the different types of CGKA protocol and focused on the confidentiality of the application message [4–8, 22, 29, 52, 53, 57, 78].

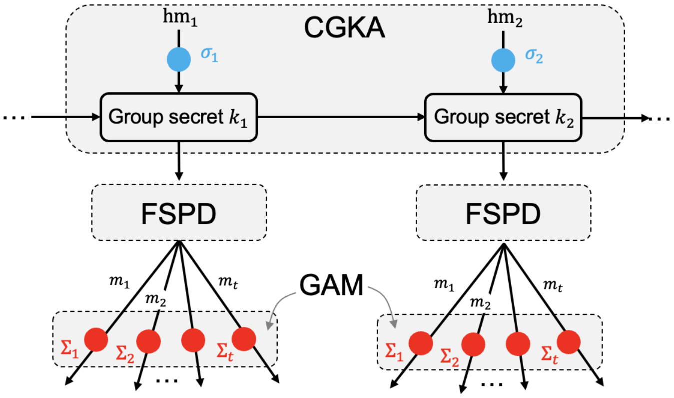

Figure 1: Relation between a CGKA, FSPD, and GAM protocols. hm denotes the handshake message used by the CGKA protocol. m_i denotes the output of the FSPD protocol; in MLS this is an *encryption* of the application message am. The blue and red circles indicate that the handshake and application messages are authenticated.

Group Authenticated Messaging Protocol. In Sec. 3, we propose the new notion of *group authenticated messaging* (GAM) protocol, allowing us to focus solely on the authenticity of the application messages while abstracting the confidentiality guarantees. More specifically, the FSPD protocol already entails the confidentiality of application messages and our GAM protocol can be viewed as adding authenticity guarantees to them. Fig. 1 gives an illustration on how a GAM protocol interacts with the CGKA and FSPD protocols. For instance, in MLS, m and Σ_1 are the encryptions of the application message am and signature sig_u on am, respectively (i.e., Enc-Sign mode).

New Authentication Modes. In Secs. 4 and 5, we introduce five new GAM protocols: `COSMOS`, `COSMAC`, `QUASAR`, `STARS`, and `GEMSTARS`. All are based on generic building blocks such as one-way functions (OWFs), message authentication codes (MACs), and key encapsulation mechanisms (KEMs) that are instantiable from both classical and post-quantum assump-

---

[4] Alwen et al. [5] uses the term forward-secure group AEAD instead of FSPD.

Table 1: Comparison between different authentication modes for secure messaging protocols. N and T denote the size of the group and the number of messages each user sends. “Communication Cost Overhead per Msg per User” is defined as the sum of 1 (offline/online) upload cost and $(N-1)$ (offline/online) download cost for each user normalized by NT. For readability, we use the simplification $(N+1)/N \approx 1$. sig, osig, and gsig denote a standard signature, a one-time signature, and a group signature, respectively. ovk denotes the verification key of a one-time signature. ct denotes a KEM ciphertext. κ denotes the security parameter, set to 128 bits. ✓$^{(*)}$ denotes that it satisfies a weaker notion of unforgeability compared to ✓ (see Sec. 2.3). “State Updates” comes with “-”, “local”, and “global”, where “-” means no state update is necessary (see Remark 3.2). `COSMOS` and `COSMAC` come with an optimized variants indicated by $(^{+})$ and $(^{++})$, whose respective total communication cost overheads and state updates are provided in parentheses.

Authentication Modes	Anon.	Unf.	Anonymous Blocklistable	Tracing Soundness	Comm. Cost Overhead per Msg per User	State Updates
Enc-Sign [12]	✓	✓	✗	✗	$\lvert\mathsf{sig}\rvert$	-
Sign-Enc-Sign [53]	✓	✓	✓	✗	$2\cdot\lvert\mathsf{sig}\rvert$	-
`COSMOS`$(^{+},^{++})$ (Secs. 4)	✗	✓$^{(*)}$	✓	✓	$3\cdot\kappa\ \ (3\cdot\kappa\,,\ (2+\frac{3}{T})\cdot\kappa)$	local (- , local)
`COSMAC`$(^{+},^{++})$ (Secs. 4)	✓	✓$^{(*)}$	✓	✗	$4\cdot\kappa\ \ (4\cdot\kappa\,,\ (3+\frac{4}{T})\cdot\kappa)$	local (-, local)
`QUASAR` (Sec. 5.1)	✓	✓$^{(*)}$	✓	✓	$6\cdot\kappa+\frac{2\cdot(\kappa+\lvert\mathsf{ct}\rvert)}{T}$	global
`STARS` (Sec. 5.2)	✓	✓	✓	✓	$\lvert\mathsf{ovk}\rvert+2\cdot\lvert\mathsf{osig}\rvert+\frac{\kappa+2\cdot\lvert\mathsf{ct}\rvert}{T}$	global
`GEMSTARS` (Sec. 5.2)	✓	✓	✓	✓	$\lvert\mathsf{sig}\rvert+\lvert\mathsf{gsig}\rvert$	-

tions. Each mode fills a specific part of the design space with strengths and weaknesses, summarized in Tab. 1. In particular, `COSMOS` and `COSMAC` do not rely on signatures and the overhead (for their optimized variants) is merely 32 B and 48 B, respectively. This offers a roughly 75x savings in the post-quantum communication overhead compared to MLS, though at the cost of slightly weakening the unforgeability guarantee; we assume the malicious server does not collude with the malicious insider. See Sec. 2.3 for more detail. We believe this significantly lower communication overhead makes it a worthwhile trade-off with security.

Efficiency Analysis. In Sec. 7, we instantiate our proposed GAM protocols from both classical and post-quantum assumptions and compare their efficiency. For completeness, we also detail in Sec. 6 how to use each of our proposed GAM protocols inside MLS. While it is mostly a simple drop in, there are minor issues that require some explanation, since the syntax of GAM protocols intentionally leaves out some functionality provided by MLS, such as what is typically captured by the CGKA protocol (e.g., welcoming group members).

Lastly, we leave it as an important future work to analyze MLS in its entirety when using our notion of GAM protocol as a building block. While Alwen et al. [5] analyze MLS by composing the CGKA and FSPD protocols with a PRF-PRNG, the output of the FSPD protocol is explicitly signed (and not encrypted); that is, they assume the vanilla (unencrypted) GAM protocol used by MLS. Replacing this with a general GAM protocol and analyzing MLS is an interesting future work. We discuss further open problems in Sec. 8.

Other related work and preliminaries are deferred to Apps. A and B, respectively.

2 Setting: Authentication in SGM

This work focuses on secure group messaging (SGM) protocols, where group users share a unique common group secret key. In this section we use MLS as our primary example, but all of our constructions apply equally to most MLS variants (e.g., [2, 4, 6, 7, 52, 53, 57]) that rely on a group secret key to exchange messages.

Below, we give a brief background on how MLS authenticates application messages. We then take a close look at different security notions under the umbrella of authenticity and formally categorize them. Building on the systematization provided in this section, we introduce the concept of a *group authenticated messaging* (GAM) protocol in Sec. 3 and formally define the relevant security notions.

2.1 Secure Group Messaging and Our Goal

Following Alwen et al. [5], we view MLS as a combination of the CGKA and FSPD protocols (see Fig. 1 for illustration). At a high level, one can draw a parallel to hybrid encryption, where the heavy public key operations are handled by the CGKA protocol and the exchange of application messages is handled by the lightweight FSPD protocol.

In more detail, the CGKA protocol allows a group of users to agree on a continuous sequence of shared (symmetric) group secret keys. By regularly updating the group secret key (and user specific keys), strong notions of *forward secrecy* and *post-compromise security* [4–6, 8, 39] are guaranteed. The protocol is also responsible for handling group operations such as adding and removing users. *Handshake messages* is an umbrella term for the exchanged messages by the CGKA protocol, used to achieve the above objectives. A handshake message, or an encryption of it, is signed using the user's signing key to authenticate the sender. This plays an important role in guaranteeing the consistency and integrity of the group state. As can also be seen from Fig. 1, the authenticity of handshake messages is analyzed implicitly as a means to show confidentiality of the group secret keys; this is similar to standard (two-user) authenticated key exchange protocols where authenticity guarantees are implicit [15, 32].

The FSPD protocol then uses the established group secret key by the CGKA protocol to securely exchange *application messages*, containing various types of payload such as chat texts, images, and stamps. Compared to the CGKA protocol, the FSPD protocol is much simpler since there are no group operations (i.e., static groups) and the objective is only confidentiality with forward secrecy; authenticity is not a security requirement. In MLS, the output of the FSPD protocol — an encryption of the application message — is then signed using a signature scheme and encrypted (i.e., Enc-Sign mode), adding the necessary authenticity guarantee. This rather ad hoc way of adding authenticity seems to be justified by the simple nature of the FSPD protocol, and indeed most works on MLS mainly focus on the security of the CGKA protocol [4, 6–8, 22, 29, 52, 53, 57, 78]. To the best of our knowledge, Alwen et al. [5] is the only prior work to analyze MLS in its entirety. They do so by modularly combining the CGKA and FSPD protocols with a PRF-PRNG, assuming the output of the FSPD protocol is authenticated by a digital signature.

The goal of our paper is thus to put a spotlight onto the authentication of application messages or, to be more precise, the output of the FSPD protocol (see Fig. 1). We introduce a new primitive called group authenticated messaging (GAM) protocol and aim to more clearly and systematically explore alternative choices to the currently (implicitly) used GAM protocol by MLS, which is the Enc-Sign mode.

2.2 Environment

We first explain the environment in which MLS operates in. This involves introducing the relevant entities and outlining the network model under consideration.

2.2.1 Entities

Group Users: The set of users in a group. Depending on the considered security notion, the users are modeled to either be all honest or some malicious. For instance, we consider the latter case when modeling a security notion where a malicious insider (e.g., [8]) tries to impersonate an honest user.

Server: Any *asynchronous* messaging protocol requires a server to curate the messages between the group users. We consider two types of servers: honest and malicious. While servers are typically considered to be malicious by default in prior work, this is because the focus is mainly on the confidentiality of the CGKA protocol. For authentication, it makes sense to consider honest servers as well. For example, the recent work by Hashimoto et al. [53] considers an honest server to anonymously block outsiders from injecting malicious messages to the group.

Outsiders: Any adversary that is not a group user or the server. For instance, a user of the secure messaging application not in the group.

2.2.2 Network Model

Due to asynchronicity, when group users exchange messages, they must upload and download these to and from the server. Depending on the anonymity guarantee we aim to achieve, there are two types of communication channels that can be used between the group users and the server.

Non-Anonymous: If the server is allowed to know the users in the group, then we assume a user-server authenticated channel is used. For instance, TLS or Noise [69] with user-side password-based authentication can be used.

Anonymous: If the group users are required to remain anonymous to the server, then we assume an authenticated anonymous channel such as TOR [42, 71] or a VPN is used.

We note dealing with authentication in the non-anonymous setting is trivial since the server can simply maintain the group list and explicitly authenticate the group users. In contrast, in the anonymous setting, such trivial solutions no longer exist and the issue of authentication becomes non-trivial. Indeed, prior works on *anonymous* secure messaging, e.g., [34, 53, 60, 75], overcome this by relying on some type of anonymous group authentication protocol.

2.3 Threat Model for Authentication

We now categorize authenticity into four different security notions: unforgeability, anonymity, anonymous blocklisting, and tracing soundness. This categorization of the application message is motivated by the security definitions used in well-studied anonymous authentication schemes, such as group signatures [13, 27, 35] and accountable ring signatures [18, 80].

Below, for each security notion, we explain *who* the adversary is, *what* the goal of the security notion is, and *why* we consider it. For simplicity, we leave outsider adversaries out of most security notions as they are strictly weaker than malicious servers and group users. The following security notions will be formalized in Sec. 3.3.

Goal 1: Unforgeability.
Adversary: Malicious group users and/or a malicious server.
Goal: No adversary can forge a signature[5] of an honest user.

---

[5] Throughout this section, we use the term "signature" loosely and note that signatures are not the only way to authenticate. Using the terminology of our GAM protocol, this is more formally an "authentication token".

This is the default notion that any secure messaging protocol must ensure. We can consider two levels of unforgeability: we call it unforgeable if the set of malicious group users and the malicious server can collude, and *non-colluding* unforgeable otherwise. The former guarantees that even a colluding malicious insider and server cannot forge a signature of an honest group user. In contrast, the latter restricts the adversary to be either the set of malicious group users or the malicious server; that is, unforgeability holds only if there is no collusion. While (standard) unforgeability is the more secure notion, sacrificing security against collusion of a malicious insider and server could be a reasonable compromise for better efficiency.

Goal 2: Anonymity.
Adversary: A malicious server.
Goal: The server cannot deanonymize and link the activity of the group users. E.g., the server cannot distinguish whether two uploaded messages came from the same user or from two different users.

For this security notion we must rely on an anonymous network model as, otherwise, communication will be linkable at the network level. We further assume all the group users to be honest, since a malicious user can always inform the server of who is in the group or who authored a message.

Goal 3: Anonymous Blocklisting.
Adversary: An outsider.
Goal: An honest server can block any outsider trying to upload messages on behalf of the group.

Observe that non-anonymous blocklisting is trivial to satisfy, since the server can perform access control by explicitly authenticating the group users. We therefore use the term "anonymous" to emphasize that the motivation of the server is to blocklist non-group users while preserving the anonymity of the users. The purpose of anonymous blocklisting is for the server to be able to prevent outsiders from launching a DoS attack on the group. Importantly, although group users can verify the authenticity of the messages by downloading them from the server, we require the server to directly reject invalid messages on behalf of the group. This is satisfied for example by Sealed Sender [60] used in the Signal protocol and the metadata-hiding MLS protocol by Hashimoto et al. [53].

Goal 4: Tracing Soundness.
Adversary: Malicious group users.
Goal: The set of honest group users can trace any (possibly maliciously crafted) signature back to a *unique* group user; if an honest user traces a signature back to a user u in the group, then all other honest users trace it back to the same user u.

Tracing soundness allows to keep the view of the honest users consistent. For instance, consider a malicious insider mounting a DoS attack against the group by spamming garbage application messages. With tracing soundness, the honest users can unanimously agree on who the malicious insider was and remove him from the group. One can draw a parallel to anonymous blocklisting, that prevents such attacks from outsiders. Moreover, while similar, it is worth noting that tracing soundness is an orthogonal notion to unforgeability. Consider a malicious insider u that modifies the signature of an honest user v in such a way that for half of the group members it traces back to u, but for the other half traces back to v. While this does not contradict unforgeability, as the malicious user is effectively just "repurposing" somebody's message, it clearly breaks the consistency of the group's view.

2.4 Modeling Choices and Simplifications

Before introducing our GAM protocol in the next section, we clarify the modeling choices and simplifications we make.

Trusted Setup. The GAM protocol assumes the states of both the group users and the server are generated honestly by an initialization phase. This simplification is justified for protocols like MLS, since users are assumed to start the GAM protocol with the group secret key, derived from the CGKA protocol, already in their states.

Static Groups. The GAM protocol assumes static groups, following the way in which MLS' FSPD protocol operates. Recall that in MLS a new FSPD protocol for a static group is initialized every time group membership changes, as this will trigger a new CGKA protocol epoch (see Fig. 1). More generally, though, we could consider a *continuous* GAM protocol where we do not need to reinitialize the protocol with every group change, similarly to a CGKA protocol. However, such a definition must be intertwined with that of the CGKA protocol responsible for group state updates, rendering the definition to be as complex as modeling MLS in its entirety. As a study investigating new security goals of authentication, we opt for making the security notions tractable and to improve the overall readability. Nonetheless, we explain in Sec. 6.2 with concrete examples on how each of our proposed GAM protocols can handle dynamic operations.

Out-of-Order Messages. In our work, we do not model authentication when messages arrive out-of-order. While this is arguably important for a comprehensive model, we highlight that, unlike confidentiality, lack of authentication does not harm the usability of the FSPD protocol. In the context of the MLS protocol, *immediate decryption* of the messages will still be maintained. The only difference between MLS is that we may lose *immediate authentication* when messages arrive out-of-order. Importantly, though security is lost while some messages are missing, assuming that every message eventually arrives, then out-of-order messages do not affect security. Instead, if some messages are permanently dropped, we can allow the recipients to fetch this missing authentication information, which they can do assuming the proper indexing of the messages required by out-of-order decryption. We note that in MLS [20, Section 5.2], whether messages eventually arrive or not is controlled by the application that sets the policy.

3 Group Authenticated Messaging Protocol

We introduce *group authenticated messaging* (GAM) protocols and the associated security requirements.

3.1 Definition

A GAM protocol is defined between a server and a group G of users. As explained above, there exists an initialization algorithm Init that prepares the initial state for the group users, possibly further preparing a secret key for the server. To send a message m (e.g., the output of a FSPD protocol), a user $u \in \mathsf{G}$ runs the Send algorithm, outputting a *group* authentication token Σ_{G}. A server verifies $(\mathsf{m}, \Sigma_{\mathsf{G}})$ using the Verify algorithm and prepares *user* authentication tokens $(\sigma_i)_{i\in[N]}$, where $N = |\mathsf{G}|$. For example, in the context of the Sign mode in MLS (see Footnote 3), Σ_{G} is simply u's signature and $\sigma_i := \Sigma_{\mathsf{G}}$. To capture anonymity, we assume the server only knows the size of the group G[6] and assume a bijective map $\mathsf{idx} : \mathsf{G} \to [N]$ is secretly known by the group users. Namely, a user u such that $i = \mathsf{idx}(u)$ fetches σ_i from the server. It then runs the $\mathsf{Receive}$ algorithm to verify (m, σ_i) and traces the purported user $v \in \mathsf{G}$ that generated σ_i. It is worth highlighting that we make a distinction between a group authentication token Σ_{G} and a user authentication token σ_i to capture an optimization technique called *selective downloading* [7, 52]. This technique allows the server to sanitize the group authentication token Σ_{G} in a straightforward manner by delivering to each group user just the strictly necessary amount of data σ_i, while maintaining the same level of (dis)trust.

Finally, we endow a GAM protocol with an *offline-online* feature. In the offline phase, when the message is still unknown, a user can perform a possibly heavy state update, and share the update with the server and the group via the $\mathsf{UpdSend}$ algorithm. This algorithm is accompanied by algorithms $\mathsf{UpdVerify}$ and $\mathsf{UpdReceive}$ similarly to above. Once the message is known in the online phase, the user can send it using its updated state.[7] Formally, we have the following.

Definition 3.1. *A* GAM *protocol for message space* $\mathcal{M}$ *between a server* Sv *and a set of users in a group* G *consists of the following algorithms, where* $\mathsf{idx} : \mathsf{G} \to [N]$ *is a bijective function with* $N := |\mathsf{G}|$*. Below, if an algorithm outputs* $\perp$*, we assume it reverts to the state before running the algorithm.*

$\mathsf{Init}(1^\kappa, \mathsf{G}) \to \big(\mathsf{pp}, \mathsf{sk}_{\mathsf{Sv}}, (\mathsf{st}_u)_{u\in\mathsf{G}}\big)$: *On input the security parameter* 1^κ *and group information* $\mathsf{G} \subset \{0,1\}^*$*, it outputs public parameters* pp*, a secret key* $\mathsf{sk}_{\mathsf{Sv}}$ *for the server* Sv*, and an initial state* st_u *for all users* $u \in \mathsf{G}$*. We assume* $\mathsf{G} \in \mathsf{st}_u$*.*

$\mathsf{Send}(\mathsf{st}_u, \mathsf{m}) \to (\mathsf{st}'_u, \Sigma_{\mathsf{G}})$ *or* $\perp$: *On input a state* st_u *for user* $u \in \mathsf{G}$ *and a message* $\mathsf{m} \in \mathcal{M}$*, user* u *outputs an updated state* st'_u *and a* group authentication token Σ_{G}*, or* $\perp$*.*

$\mathsf{Verify}(\mathsf{pp}, \mathsf{sk}_{\mathsf{Sv}}, \Sigma_{\mathsf{G}}, \mathsf{m}) \to (\mathsf{pp}', (\sigma_i)_{i\in[N]})$ *or* $\perp$: *On input public parameters* pp*, a server secret key* $\mathsf{sk}_{\mathsf{Sv}}$*, a group authentication token* Σ_{G}*, and a message* $\mathsf{m} \in \mathcal{M}$*, the server* Sv *outputs updated public parameters* pp' *and* N user authentication tokens $(\sigma_i)_{i\in[N]}$*, or* $\perp$*.*

$\mathsf{Receive}(\mathsf{st}_u, \sigma, \mathsf{m}) \to (\mathsf{st}'_u, b \in \{\top, \perp\}, v \in \mathsf{G} \cup \{\perp\})$: *On input a state* st_u *for user* $u \in \mathsf{G}$*, a user authentication token* σ*, and a message* $\mathsf{m} \in \mathcal{M}$*, user* u *outputs an updated state* st'_u*, a bit* b *indicating whether the token was valid* ($b = \top$) *or invalid* ($b = \perp$)*, and a purported user* $v \in \mathsf{G} \cup \{\perp\}$*, where* $v = \perp$ *if tracing fails.*

$\mathsf{UpdSend}(\mathsf{st}_u) \to (\mathsf{st}'_u, \widehat{\Sigma}_{\mathsf{G}}, \widehat{\mathsf{ct}}_{\mathsf{G}})$: *On input a state* st_u *for user* $u \in \mathsf{G}$*, user* u *outputs an updated state* st'_u*, a* group update authentication token $\widehat{\Sigma}_{\mathsf{G}}$*, and* group update information $\widehat{\mathsf{ct}}_{\mathsf{G}}$*.*

$\mathsf{UpdVerify}(\mathsf{pp}, \mathsf{sk}_{\mathsf{Sv}}, \widehat{\Sigma}_{\mathsf{G}}, \widehat{\mathsf{ct}}_{\mathsf{G}}) \to \big(\mathsf{pp}', (\widehat{\sigma}_i, \widehat{\mathsf{ct}}_i)_{i\in[N]}\big)$ *or* $\perp$: *On input public parameters* pp*, a server secret key* $\mathsf{sk}_{\mathsf{Sv}}$*, a group update authentication token* $\widehat{\Sigma}_{\mathsf{G}}$*, and group update information* $\widehat{\mathsf{ct}}_{\mathsf{G}}$*, the server* Sv *outputs updated public parameters* pp' *and a list of* user update authentication tokens *and* user update information $\big(\widehat{\sigma}_i, \widehat{\mathsf{ct}}_i\big)_{i\in[N]}$*, or* $\perp$*.*

$\mathsf{UpdReceive}(\mathsf{st}_u, \widehat{\sigma}, \widehat{\mathsf{ct}}) \to (\mathsf{st}'_u, b \in \{\top, \perp\}, v \in \mathsf{G} \cup \{\perp\})$: *On input a state* st_u *for user* $u \in \mathsf{G}$*, a user update authentication token* $\widehat{\sigma}$*, and user update information* $\widehat{\mathsf{ct}}$*, user* u *outputs an updated state* state'_u*, a bit* b *indicating whether the token was valid* ($b = \top$) *or invalid* ($b = \perp$)*, and a purported user* $v \in \mathsf{G} \cup \{\perp\}$*, where* $v = \perp$ *if tracing fails.*

Remark 3.2 (Local and Global State Updates). For some protocols the user state may only allow signing up to T messages, and it may need to be updated before the user can sign again. There are two ways to perform state updates: *locally* and *globally*. In the former, a user regains the ability to send messages once it has updated its own state. In the latter, a user regains the ability to send messages only after every user in the group updates their states. Since global state updates are much more costly than local state updates, they are only useful if one state update allows to send a large number of messages T. Further, global updates can only guarantee security if users are online, a clear disadvantage over local updates. For the schemes presented in this paper there are no risks of a deadlock — i.e., a situation where a global state update cannot be completed and users are prevented to keep sending messages — as long as the users perform updates once coming online. However,

---

[6] While we could consider further hiding the size of the group to the server, we choose not to since it would resort in an inefficient padding strategy. This is the same level of anonymity satisfied by previous anonymous SGM protocols e.g., [34, 53].

[7] Naturally, protocols need not have such a differentiation and can simply only perform online state updates. This optimization allows us to improve the real-world usability of those protocols that do, as they can more evenly distribute their computation and communication over time.

the general definition of global updates does not guarantee that such a deadlock does not occur.

3.2 Correctness

We define two types of signing correctness. One for signing messages and the other for signing updates. They stipulate that an honestly generated user authentication token is always valid and traceable.

Definition 3.3 (Signing Correctness). *For any* $\kappa \in \mathbb{N}$, $\mathsf{G} \subset \{0,1\}^*$, *and* $\left(\mathsf{pp}, \mathsf{sk}_{\mathsf{Sv}}, (\mathsf{st}_u)_{u\in\mathsf{G}}\right) \in \mathsf{Init}(1^\kappa, \mathsf{G})$, *if we execute* $(\mathsf{Send}, \mathsf{Verify}, \mathsf{Receive}, \mathsf{UpdSend}, \mathsf{UpdVerify}, \mathsf{UpdReceive})$ *in an arbitrary but honest manner (i.e., we only run the algorithms on inputs that were output by another algorithm and run* Receive *(resp.* UpdReceive*) for all users after running* Verify *(resp.* UpdVerify*))*[8]*, then we have the following, where* pp' *and* $(\mathsf{st}'_u)_{u\in\mathsf{G}}$ *are arbitrary public parameters and states reachable from the initial* pp *and* $(\mathsf{st}_u)_{u\in\mathsf{G}}$*:*

Message Signing Correctness*: For any* $\mathsf{m} \in \mathcal{M}$ *and* $u \in \mathsf{G}$, *if we execute* $(\mathsf{st}''_u, \Sigma_{\mathsf{G}}) \leftarrow\!\!\$\, \mathsf{Send}(\mathsf{st}'_u, \mathsf{m})$, *redefine* $\mathsf{st}'_u := \mathsf{st}''_u$, *and execute* $(\mathsf{pp}'', (\sigma_i)_{i\in[N]}) \leftarrow \mathsf{Verify}(\mathsf{pp}', \mathsf{sk}_{\mathsf{Sv}}, \Sigma_{\mathsf{G}}, \mathsf{m})$, $(\mathsf{st}''_v, b_v, u_v) \leftarrow \mathsf{Receive}(\mathsf{st}'_v, \sigma_{\mathsf{idx}(v)}, \mathsf{m})$ *for all* $v \in \mathsf{G}$, *then conditioned on* $\Sigma_{\mathsf{G}} \neq \bot$, *we have* $(b_v, u_v) = (\top, u)$ *(i.e., every user accepts and traces the message back to u).*

Update Signing Correctness*: For any* $u \in \mathsf{G}$, *if we execute* $(\mathsf{st}''_u, \widehat{\Sigma}_{\mathsf{G}}, \widehat{\mathsf{ct}}_{\mathsf{G}}) \leftarrow\!\!\$\, \mathsf{UpdSend}(\mathsf{st}'_u)$, *redefine* $\mathsf{st}'_u := \mathsf{st}''_u$, *and execute* $(\mathsf{pp}'', (\widehat{\sigma}_i, \widehat{\mathsf{ct}}_i)_{v\in[N]}) \leftarrow \mathsf{UpdVerify}(\mathsf{pp}', \mathsf{sk}_{\mathsf{Sv}}, \widehat{\Sigma}_{\mathsf{G}}, \widehat{\mathsf{ct}}_{\mathsf{G}})$ *followed by* $(\mathsf{st}''_v, b_v, u_v) \leftarrow \mathsf{UpdReceive}(\mathsf{st}'_v, \widehat{\sigma}_{\mathsf{idx}(v)}, \widehat{\mathsf{ct}}_{\mathsf{idx}(v)})$ *for all* $v \in \mathsf{G}$, *then conditioned on* $\widehat{\Sigma}_{\mathsf{G}} \neq \bot$, *we have* $(b_v, u_v) = (\top, u)$ *(i.e., every user accepts and traces the update back to u).*

In some protocols, the states may occasionally need to be updated in order to regain the ability to send messages again. As explained in Remark 3.2, there are two ways a users can update their states. One is *local*, where it is sufficient that a user can simply update its state. The other one is *global*, where all the group users must update their states. Below, we define correctness of both state update modes.

Definition 3.4 (State-Update Correctness). *Assume the same precondition as in Def. 3.3. Then, we have* either *of the following:*

Local State-Update Correctness: *For any* $\mathsf{m} \in \mathcal{M}$ *and user* $u \in \mathsf{G}$, *if* $(\mathsf{st}''_u, \Sigma_{\mathsf{G}}) \leftarrow\!\!\$\, \mathsf{Send}(\mathsf{st}'_u, \mathsf{m})$ *such that* $\Sigma_{\mathsf{G}} = \bot$, *then if u executes* $(\mathsf{st}^*_u, \widehat{\Sigma}_{\mathsf{G}}, \widehat{\mathsf{ct}}_{\mathsf{G}}) \leftarrow\!\!\$\, \mathsf{UpdSend}(\mathsf{st}'_u)$, *the server* Sv *executes* $(\mathsf{pp}^*, (\widehat{\sigma}_i, \widehat{\mathsf{ct}}_i)_{i\in[N]}) \leftarrow \mathsf{UpdVerify}(\mathsf{pp}', \mathsf{sk}_{\mathsf{Sv}}, (\widehat{\Sigma}_{\mathsf{G}}, \widehat{\mathsf{ct}}_{\mathsf{G}}))$, *and every user* $v \in \mathsf{G}$ *executes* $\mathsf{UpdReceive}(\mathsf{st}'_v, (\widehat{\sigma}_{\mathsf{idx}(v)}, \widehat{\mathsf{ct}}_{\mathsf{idx}(v)}))$, *then the updated public parameters* pp^* *and state* st^*_u *allow user u to sign on* m, *that is,* $\Sigma_{\mathsf{G}} \neq \bot$ *and signing correctness holds (i.e., after user u sends a user update information* $\widehat{\mathsf{ct}}_{\mathsf{idx}(v)}$ *to every user v, then user u's state will be* refreshed*).*

State Update Correctness*: For any* $\mathsf{m} \in \mathcal{M}$ *and user* $u \in \mathsf{G}$, *if* $(\mathsf{st}''_u, \Sigma_{\mathsf{G}}) \leftarrow\!\!\$\, \mathsf{Send}(\mathsf{st}'_u, \mathsf{m})$ *such that* $\Sigma_{\mathsf{G}} = \bot$, *then if all users* $v \in \mathsf{G}$ *execute* $(\mathsf{st}''_v, \widehat{\Sigma}^v_{\mathsf{G}}, \widehat{\mathsf{ct}}^v_{\mathsf{G}}) \leftarrow\!\!\$\, \mathsf{UpdSend}(\mathsf{st}'_v)$, *the server* Sv *executes* $\mathsf{UpdVerify}$ *for all* $\left(\widehat{\Sigma}^v_{\mathsf{G}}, \widehat{\mathsf{ct}}^v_{\mathsf{G}}\right)_{v\in\mathsf{G}}$ *in an arbitrary order, and user u executes* $\mathsf{UpdReceive}$ *for all* $\left(\widehat{\sigma}^v_{\mathsf{idx}(u)}, \widehat{\mathsf{ct}}^v_{\mathsf{idx}(u)}\right)_{v\in\mathsf{G}}$ *output by* $\mathsf{UpdVerify}$ *in an arbitrary order, then the updated public parameters* pp^* *and state* st^*_u *allow user u to sign on* m, *that is,* $\Sigma_{\mathsf{G}} \neq \bot$ *and signing correctness holds (i.e., after every user v sends a user update information* $\widehat{\mathsf{ct}}^v_{\mathsf{idx}(u)}$ *to user u, then user u's state will be* refreshed*). Note that by symmetry, all users' state is refreshed.*

3.3 Security

We formalize the threat models explained in Sec. 2.3: unforgeability, anonymity, anonymous blocklisting, and tracing soundness, via a security game defined in Fig. 2. The probability of the game outputting 1 against an efficient adversary must be negligible for every game except for anonymity. For anonymity, as it is a distinguishing game, the game must output 1 with probability negligibly close to $\frac{1}{2}$.

For every game, the adversary is given access to oracles $\{\mathcal{O}_{\mathsf{Send}}, \mathcal{O}_{\mathsf{UpdSend}}\}$, allowing it to invoke honest users to create group (update) authentication tokens. The adversary is further given access to either $\{\mathcal{O}_{\mathsf{Receive}}, \mathcal{O}_{\mathsf{UpdReceive}}\}$ or $\{\mathcal{O}_{\mathsf{GroupReceive}}, \mathcal{O}_{\mathsf{GroupUpdReceive}}\}$. The former allows the adversary to directly invoke honest users to process (update) authentication tokens. This capture malicious server capabilities and is used by the unforgeability and anonymity games. In contrast, the latter only allows the adversary to query for *group* (update) authentication tokens. The oracle then individually invokes each honest users on the correctly processed (update) authentication tokens. Namely, this captures honest server behavior and models the fact that malicious users cannot directly send messages to group users. It is worth noting that, in this case, the authentication tokens created in $\{\mathcal{O}_{\mathsf{Send}}, \mathcal{O}_{\mathsf{UpdSend}}\}$ are directly processed by $\{\mathcal{O}_{\mathsf{GroupReceive}}, \mathcal{O}_{\mathsf{GroupUpdReceive}}\}$, modeling the fact that the communication channel between an honest user and server is secure.

To aid readability, we highlight some features of the security game. We model two types of unforgeability by $\mathsf{Game}^{\mathsf{X}}_{\mathcal{A}}$ with $\mathsf{X} \in \{\mathsf{ncUnf}, \mathsf{Unf}\}$. In standard unforgeability, as the adversary models both a malicious user and server, it has unrestricted access to all oracles. In contrast, for non-colluding unforgeability, we have two case distinctions depending on whether the set of corrupted users $\mathcal{C} = \emptyset$ or not.

---

[8] We impose the second condition to define correctness in a minimal yet well-defined manner. Without it, we must also include cases such as when only part of the users received a user update information.

In the former case the adversary is a malicious server, so the adversary is given the server secret key $\mathsf{sk}_{\mathsf{Sv}}$ and has access to $\{\, \mathcal{O}_{\mathsf{Receive}}, \mathcal{O}_{\mathsf{UpdReceive}} \,\}$. In the latter case the adversary is a set of malicious users, so the adversary is instead given the corrupted users' states and only has access to $\{\, \mathcal{O}_{\mathsf{GroupReceive}}, \mathcal{O}_{\mathsf{GroupUpdReceive}} \,\}$. For both types of unforgeability, an adversary wins if it can output a valid user (update) authentication token for an honest user that it has not seen before. For anonymity, we model a malicious server by giving the adversary the server secret key $\mathsf{sk}_{\mathsf{Sv}}$. The adversary outputs two users and messages and the game creates the group authentication tokens for both users. To non-trivialize the game, we restrict the (group) authentication tokens to be valid. To perform this check, the adversary needs to further output a (possibly malformed) public parameter $\overline{\mathsf{pp}}$ so the game can run algorithm Verify. The adversary can further perform oracle queries under the restriction that it does not query the receive oracles on the challenge authentication tokens.

We now provide the formal definition of non-colluding and standard unforgeability, anonymity, anonymous blocklisting, and tracing soundness and some more intuitions on how to understand them.

Unforgeability. As already discussed above, we model standard and non-colluding unforgeability by giving the adversary access to different oracles and running it with different inputs (i.e., with or without server and user states). The adversary wins if it outputs a valid authentication token such that $v_u \in \mathcal{H} \wedge (v_u, *, \bar{\mathsf{m}}) \notin L_{\mathsf{upd}}$ where $\bar{\mathsf{m}} \in \{\, \mathsf{m}, \widehat{\mathsf{ct}} \,\}$ holds, where recall v_u is the traced user (cf. unforgeability game, lines 17 and 23). The former checks that the user v_u is not malicious; without this check a malicious user can trivially win unforgeability. The latter checks that the honest user v_u did not sign $\bar{\mathsf{m}}$. Formally, we define unforgeability as follows.

Definition 3.5 (Unforgeability). *We define* $\mathsf{Game}^{\mathsf{ncUnf}}_{\mathcal{A}}(1^\kappa)$ *as in Fig. 2 for an adversary* $\mathcal{A}$*. We say a* GAM *protocol is* no-colluding unforgeable *if for any* $\mathsf{G} \subset \{0,1\}^*$*, injective function* $\mathsf{idx} : \mathsf{G} \to [N]$ *with* $N = |\mathsf{G}|$*, and any PPT adversary* $\mathcal{A}$*, we have*

$$\mathsf{Adv}^{\mathsf{ncUnf}}_{\mathcal{A}}(1^\kappa) := \Pr[\mathsf{Game}^{\mathsf{ncUnf}}_{\mathcal{A}}(1^\kappa) = 1] = \mathsf{negl}(\kappa).$$

We further say the scheme is (standard) unforgeable if the above holds for $\mathsf{Game}^{\mathsf{Unf}}_{\mathcal{A}}(1^\kappa)$ *as defined in Fig. 2.*

Anonymity. As briefly explained above, the game checks if the group authentication token Σ_{G} and the individual authentication tokens $(\sigma_i)_{i\in[N]}$ are valid. Without this check, an adversary may trivially break anonymity if the protocol requires state updates. Concretely, assume the adversary queries a user u_0 to oracle $\mathcal{O}_{\mathsf{Send}}$ until u_0 can no longer sign without performing an update. At this point, if the adversary challenges user u_0 and u_1, then it can trivially break anonymity as u_0 cannot produce a group authentication token while u_1 can. Moreover, while we can easily define anonymity for updates, we chose not to do so as updates are sent far less often compared to messages, and we opted for simplicity of the security game. Formally, we define anonymity as follows.

Definition 3.6 (Anonymity). *We define* $\mathsf{Game}^{\mathsf{Anon}}_{\mathcal{A}}(1^\kappa)$ *as in Fig. 2 for an adversary* $\mathcal{A}$*. We say a* GAM *protocol is* anonymous *if for any* $\mathsf{G} \subset \{0,1\}^*$*, injective function* $\mathsf{idx} : \mathsf{G} \to [N]$ *with* $N = |\mathsf{G}|$*, and any PPT adversary* $\mathcal{A}$*, we have*

$$\mathsf{Adv}^{\mathsf{Anon}}_{\mathcal{A}}(1^\kappa) := \left| \Pr[\mathsf{Game}^{\mathsf{Anon}}_{\mathcal{A}}(1^\kappa) = 1] - \frac{1}{2} \right| = \mathsf{negl}(\kappa).$$

Anonymous Blocklisting. This game is quite intuitive as the adversary is an outsider. The adversary wins the game if it's able to output a valid group authentication token that nobody in the group created. Although similar, we note that anonymous blocklisting is different from unforgeability. To win anonymous blocklisting, the adversary is required to output a *group* authentication token Σ_{G} that verifies. This entails the fact that the server can check the validity of Σ_{G} and immediately block malformed group authentication tokens on behalf of the group users. In contrast, unforgeability does not capture this type of blocking by the server. Formally, we define anonymous blocklisting as follows.

Definition 3.7 (Anonymous Blocklisting). *We define the security game* $\mathsf{Game}^{\mathsf{AnonBlock}}_{\mathcal{A}}(1^\kappa)$ *as in Fig. 2 for an adversary* $\mathcal{A}$*. We say a* GAM *protocol is* anonymous blocklistable *if for any* $\mathsf{G} \subset \{0,1\}^*$*, injective function* $\mathsf{idx} : \mathsf{G} \to [N]$ *with* $N = |\mathsf{G}|$*, and any PPT adversary* $\mathcal{A}$*, we have*

$$\mathsf{Adv}^{\mathsf{AnonBlock}}_{\mathcal{A}}(1^\kappa) := \Pr[\mathsf{Game}^{\mathsf{AnonBlock}}_{\mathcal{A}}(1^\kappa) = 1] = \mathsf{negl}(\kappa).$$

Tracing Soundness. As discussed in Sec. 2.3, the adversary wins if it outputs a group authentication token for which the set L_{tr} of traced users by the honest users is not of the form $L_{\mathsf{tr}} = \{\, v \,\}$ for some group user $v \in \mathsf{G}$. That is, if the group authentication token is valid, it must be traceable to some user in the group and this user must be unique among the honest users. Formally, we define tracing soundness as follows.

Definition 3.8 (Tracing Soundness). *We define* $\mathsf{Game}^{\mathsf{TraceSound}}_{\mathcal{A}}(1^\kappa)$ *as in Fig. 2 for an adversary* $\mathcal{A}$*. We say a* GAM *protocol is* tracing sound *if for any* $\mathsf{G} \subset \{0,1\}^*$*, injective function* $\mathsf{idx} : \mathsf{G} \to [N]$ *with* $N = |\mathsf{G}|$*, and any PPT adversary* $\mathcal{A}$*, we have*

$$\mathsf{Adv}^{\mathsf{TraceSound}}_{\mathcal{A}}(1^\kappa) := \Pr[\mathsf{Game}^{\mathsf{TraceSound}}_{\mathcal{A}}(1^\kappa) = 1] = \mathsf{negl}(\kappa).$$

Remark 3.9 (Transparency of Server). In any secure messaging protocol it may be important to have a transparent server, so as to limit the trust we put in it. In the context of a GAM protocol, notice that our current initialization algorithm Init

$\mathsf{Game}_{\mathcal{A}}^{\mathsf{X}}(1^{\kappa}) : \mathsf{X} \in \{\mathsf{ncUnf}, \mathsf{Unf}\}$

```
$\mathcal{C} \leftarrow\!\$\, \mathcal{A}(1^{\kappa})$
$\mathcal{H} := \mathsf{G} \setminus \mathcal{C}$
$L_{\mathsf{msg}}, L_{\mathsf{upd}} := \emptyset$   / Book keeping
$(\mathsf{pp}, \mathsf{sk}_{\mathsf{Sv}}, (\mathsf{st}_u)_{u \in \mathsf{G}}) \leftarrow\!\$\, \mathsf{Init}(1^{\kappa}, \mathsf{G})$
if $\mathsf{X} = \mathsf{Unf}$ then
    $(\mathtt{label}, \mathtt{obj}) \leftarrow\!\$\, \mathcal{A}^{O}(\mathsf{pp}, \mathsf{sk}_{\mathsf{Sv}}, (\mathsf{st}_u)_{u \in \mathcal{C}})$
else   / No collusion between malicious user and server
    if $\mathcal{C} = \emptyset$   / Honest users
        $(\mathtt{label}, \mathtt{obj}) \leftarrow\!\$\, \mathcal{A}^{O}(\mathsf{pp}, \mathsf{sk}_{\mathsf{Sv}})$
    else   / Honest server
        $(\mathtt{label}, \mathtt{obj}) \leftarrow\!\$\, \mathcal{A}^{O^{\star}}(\mathsf{pp}, (\mathsf{st}_u)_{u \in \mathcal{C}})$
if $\mathtt{label} = \mathtt{msg}$ then
    parse $(u, \sigma, \mathsf{m}) \leftarrow \mathtt{obj}$
    req $u \in \mathcal{H}$
    $(\mathsf{st}'_u, b_v, v_u) \leftarrow \mathsf{Receive}(\mathsf{st}_u, \sigma, \mathsf{m})$
    req $b_v = \top$   / Valid authentication token
    $b \leftarrow [\![ v_u \in \mathcal{H} \wedge (v_u, *, \mathsf{m}) \notin L_{\mathsf{msg}} ]\!]$
elseif $\mathtt{label} = \mathtt{upd}$ then
    parse $(u, \widehat{\sigma}, \widehat{\mathsf{ct}}) \leftarrow \mathtt{obj}$
    req $u \in \mathcal{H}$
    $(\mathsf{st}'_u, b_v, v_u) \leftarrow \mathsf{UpdReceive}(\mathsf{st}_u, \widehat{\sigma}, \widehat{\mathsf{ct}})$
    req $b_v = \top$   / Valid authentication token
    $b \leftarrow [\![ v_u \in \mathcal{H} \wedge (v_u, *, \widehat{\mathsf{ct}}) \notin L_{\mathsf{upd}} ]\!]$
return $b$
```

$\mathsf{Game}_{\mathcal{A}}^{\mathsf{Anon}}(1^{\kappa})$

```
$\mathcal{H} := \mathsf{G}$   / No corrupt users
$\mathsf{Chall}_{\mathsf{msg}} := \emptyset$
$\mathsf{coin} \leftarrow\!\$\, \{0,1\}$
$(\mathsf{pp}, \mathsf{sk}_{\mathsf{Sv}}, (\mathsf{st}_u)_{u \in \mathsf{G}}) \leftarrow\!\$\, \mathsf{Init}(1^{\kappa}, \mathsf{G})$
$(\overline{\mathsf{pp}}, u_0, u_1, \mathsf{m}_0, \mathsf{m}_1) \leftarrow\!\$\, \mathcal{A}^{O}(\mathsf{pp}, \mathsf{sk}_{\mathsf{Sv}})$
foreach $b \in \{0,1\}$ do
    $(\mathsf{st}'_{u_b}, \Sigma_{\mathsf{G}}^{b}) \leftarrow\!\$\, \mathsf{Send}(\mathsf{st}_{u_b}, \mathsf{m}_{b \oplus \mathsf{coin}})$
    $(\overline{\mathsf{pp}}', (\sigma_i^{b})_{i \in [N]})$
            $\leftarrow \mathsf{Verify}(\overline{\mathsf{pp}}, \mathsf{sk}_{\mathsf{Sv}}, \Sigma_{\mathsf{G}}^{b}, \mathsf{m}_{b \oplus \mathsf{coin}})$
    / Require the authentication token to be valid
    req $\overline{\mathsf{pp}}' \neq \bot$
    foreach $u \in \mathcal{H}$ do
        $(\mathsf{st}'_u, b_u, v_u) \leftarrow \mathsf{Receive}(\mathsf{st}_u, \sigma_{\mathsf{idx}(u)}^{b}, \mathsf{m}_{b \oplus \mathsf{coin}})$
        req $b_u \neq \bot$
    $\mathsf{Chall}_{\mathsf{msg}} \leftarrow \mathsf{Chall}_{\mathsf{msg}} \cup \{ \sigma_i^{b} \}_{i \in [N]}$
    $\overline{\mathsf{pp}} \leftarrow \overline{\mathsf{pp}}'$
$\widehat{\mathsf{coin}} \leftarrow\!\$\, \mathcal{A}^{O}(\mathsf{Chall}_{\mathsf{msg}})$
return $[\![ \mathsf{coin} = \widehat{\mathsf{coin}} ]\!]$
```

$\mathsf{Game}_{\mathcal{A}}^{\mathsf{AnonBlock}}(1^{\kappa})$

```
$\mathcal{H} := \mathsf{G}$   / No corrupt users
$L_{\mathsf{msg}}, L_{\mathsf{upd}} := \emptyset$   / Book keeping
$(\mathsf{pp}, \mathsf{sk}_{\mathsf{Sv}}, (\mathsf{st}_u)_{u \in \mathsf{G}}) \leftarrow\!\$\, \mathsf{Init}(1^{\kappa}, \mathsf{G})$
$(\mathtt{label}, \mathtt{obj}) \leftarrow\!\$\, \mathcal{A}^{O^{\star}}(\mathsf{pp})$   / Malicious outsides
if $\mathtt{label} = \mathtt{msg}$ then
    parse $(\Sigma_{\mathsf{G}}, \mathsf{m}) \leftarrow \mathtt{obj}$
    $(\mathsf{pp}', (\sigma_i)_{i \in [N]}) \leftarrow \mathsf{Verify}(\mathsf{pp}, \mathsf{sk}_{\mathsf{Sv}}, \Sigma_{\mathsf{G}}, \mathsf{m})$
    req $\mathsf{pp}' \neq \bot$   / Require Verify to succeed
    / Sv accepts new non-member token
    $b \leftarrow [\![ (*, \Sigma_{\mathsf{G}}, *) \notin L_{\mathsf{msg}} ]\!]$
elseif $\mathtt{label} = \mathtt{upd}$ then
    parse $(\widehat{\Sigma}_{\mathsf{G}}, \widehat{\mathsf{ct}}_{\mathsf{G}}) \leftarrow \mathtt{obj}$
    $\left(\mathsf{pp}', (\widehat{\sigma}_i, \widehat{\mathsf{ct}}_i)_{i \in [N]}\right)$
        $\leftarrow \mathsf{UpdVerify}(\mathsf{pp}, \mathsf{sk}_{\mathsf{Sv}}, \widehat{\Sigma}_{\mathsf{G}}, \widehat{\mathsf{ct}}_{\mathsf{G}})$
    req $\mathsf{pp}' \neq \bot$   / Require UpdVerify to succeed
    / Sv accepts new non-member token
    $b \leftarrow [\![ (*, \widehat{\Sigma}_{\mathsf{G}}, *) \notin L_{\mathsf{upd}} ]\!]$
return $b$
```

$\mathsf{Game}_{\mathcal{A}}^{\mathsf{TraceSound}}(1^{\kappa})$

```
$\mathcal{C} \leftarrow\!\$\, \mathcal{A}(1^{\kappa})$
$\mathcal{H} := \mathsf{G} \setminus \mathcal{C}$
$L_{\mathsf{tr}} := \emptyset$   / Book keeping
$(\mathsf{pp}, \mathsf{sk}_{\mathsf{Sv}}, (\mathsf{st}_u)_{u \in \mathsf{G}}) \leftarrow\!\$\, \mathsf{Init}(1^{\kappa}, \mathsf{G})$
$(\mathtt{label}, \mathtt{obj}) \leftarrow\!\$\, \mathcal{A}^{O^{\star}}(\mathsf{pp}, (\mathsf{st}_u)_{u \in \mathcal{C}})$
if $\mathtt{label} = \mathtt{msg}$ then
    parse $(\Sigma_{\mathsf{G}}, \mathsf{m}) \leftarrow \mathtt{obj}$
    $(\mathsf{pp}', (\sigma_i)_{i \in [N]}) \leftarrow \mathsf{Verify}(\mathsf{pp}, \mathsf{sk}_{\mathsf{Sv}}, \Sigma_{\mathsf{G}}, \mathsf{m})$
    req $\mathsf{pp}' \neq \bot$   / Require UpdVerify to succeed
    foreach $u \in \mathcal{H}$ do
        $(\mathsf{st}'_u, b_v, v_u) \leftarrow \mathsf{Receive}(\mathsf{st}_u, \sigma_{\mathsf{idx}(u)}, \mathsf{m})$
        if $b_v = \top$ then
            $L_{\mathsf{tr}} \leftarrow L_{\mathsf{tr}} \cup \{ v_u \}$   / If tracing fails, $v_u = \bot$
elseif $\mathtt{label} = \mathtt{upd}$ then
    parse $(\widehat{\Sigma}_{\mathsf{G}}, \widehat{\mathsf{ct}}_{\mathsf{G}}) \leftarrow \mathtt{obj}$
    $\left(\mathsf{pp}', (\widehat{\sigma}_i, \widehat{\mathsf{ct}}_i)_{i \in [N]}\right)$
        $\leftarrow \mathsf{UpdVerify}(\mathsf{pp}, \mathsf{sk}_{\mathsf{Sv}}, \widehat{\Sigma}_{\mathsf{G}}, \widehat{\mathsf{ct}}_{\mathsf{G}})$
    req $\mathsf{pp}' \neq \bot$   / Require UpdVerify to succeed
    foreach $u \in \mathcal{H}$ do
        $(\mathsf{st}'_u, b_v, v_u)$
            $\leftarrow \mathsf{UpdReceive}(\mathsf{st}_u, \widehat{\sigma}_{\mathsf{idx}(u)}, \widehat{\mathsf{ct}}_{\mathsf{idx}(u)})$
        if $b_v = \top$ then
            $L_{\mathsf{tr}} \leftarrow L_{\mathsf{tr}} \cup \{ v_u \}$   / If tracing fails, $v_u = \bot$
/ Does not uniquely trace user
$b \leftarrow [\![ \nexists v \in \mathsf{G} : L_{\mathsf{tr}} = \{ v \} ]\!]$
return $b$
```

Oracle $O_{\mathsf{Send}}(u \in \mathcal{H}, \mathsf{m})$

```
$s_{\mathsf{Rec}} := \emptyset$
$(\mathsf{st}'_u, \Sigma_{\mathsf{G}}) \leftarrow\!\$\, \mathsf{Send}(\mathsf{st}_u, \mathsf{m})$
$L_{\mathsf{msg}} \leftarrow L_{\mathsf{msg}} \cup \{ (u, \Sigma_{\mathsf{G}}, \mathsf{m}) \}$
/ Server honestly processes group authentication
if $\mathcal{A}$ has access to $O^{\star}$ then
    $s_{\mathsf{Rec}} \leftarrow O_{\mathsf{GroupReceive}}(\Sigma_{\mathsf{G}}, \mathsf{m})$
return $(\Sigma_{\mathsf{G}}, s_{\mathsf{Rec}})$
```

Oracle $O_{\mathsf{Receive}}(u \in \mathcal{H}, \sigma, \mathsf{m})$

```
req $\sigma \notin \mathsf{Chall}_{\mathsf{msg}}$   / Only used by anonymity
$(\mathsf{st}'_u, b_u, v_u) \leftarrow \mathsf{Receive}(\mathsf{st}_u, \sigma_{\mathsf{idx}(u)}, \mathsf{m})$
return $(b_u, v_u)$
```

Oracle $O_{\mathsf{GroupReceive}}(\Sigma_{\mathsf{G}}, \mathsf{m})$

```
$(\mathsf{pp}', (\sigma_i)_{i \in [N]}) \leftarrow \mathsf{Verify}(\mathsf{pp}, \mathsf{sk}_{\mathsf{Sv}}, \Sigma_{\mathsf{G}}, \mathsf{m})$
if $\mathsf{pp}' = \bot$ then return $\bot$
foreach $u \in \mathcal{H}$ do
    $(\mathsf{st}'_u, b_u, v_u) \leftarrow \mathsf{Receive}(\mathsf{st}_u, \sigma_{\mathsf{idx}(u)}, \mathsf{m})$
return $(b_u, v_u)_{u \in \mathcal{H}}$
```

Oracle $O_{\mathsf{UpdSend}}(u \in \mathcal{H})$

```
$s_{\mathsf{UpdRec}} := \emptyset$
$(\mathsf{st}'_u, \widehat{\Sigma}_{\mathsf{G}}, \widehat{\mathsf{ct}}_{\mathsf{G}}) \leftarrow\!\$\, \mathsf{UpdSend}(\mathsf{st}_u)$
$L_{\mathsf{upd}} \leftarrow L_{\mathsf{upd}} \cup \{ (u, \widehat{\Sigma}_{\mathsf{G}}, \widehat{\mathsf{ct}}_{\mathsf{G}}) \}$
/ Server honestly processes group authentication
if $\mathcal{A}$ has access to $O^{\star}$ then
    $s_{\mathsf{UpdRec}} \leftarrow O_{\mathsf{GroupUpdReceive}}(\widehat{\Sigma}_{\mathsf{G}}, \widehat{\mathsf{ct}}_{\mathsf{G}})$
return $((\widehat{\Sigma}_{\mathsf{G}}, \widehat{\mathsf{ct}}_{\mathsf{G}}), s_{\mathsf{UpdRec}})$
```

Oracle $O_{\mathsf{UpdReceive}}(u \in \mathcal{H}, \widehat{\sigma}, \widehat{\mathsf{ct}})$

```
$(\mathsf{st}'_u, b_u, v_u)$
        $\leftarrow \mathsf{UpdReceive}(\mathsf{st}_u, \widehat{\sigma}_{\mathsf{idx}(u)}, \widehat{\mathsf{ct}}_{\mathsf{idx}(u)})$
return $(b_u, v_u)$
```

Oracle $O_{\mathsf{GroupUpdReceive}}(\widehat{\Sigma}_{\mathsf{G}}, \widehat{\mathsf{ct}}_{\mathsf{G}})$

```
$\left(\mathsf{pp}', (\widehat{\sigma}_i, \widehat{\mathsf{ct}}_i)_{i \in [N]}\right)$
        $\leftarrow \mathsf{UpdVerify}(\mathsf{pp}, \mathsf{sk}_{\mathsf{Sv}}, \widehat{\Sigma}_{\mathsf{G}}, \widehat{\mathsf{ct}}_{\mathsf{G}})$
if $\mathsf{pp}' = \bot$ then return $\bot$
foreach $u \in \mathcal{H}$ do
    $(\mathsf{st}'_u, b_u, v_u)$
            $\leftarrow \mathsf{UpdReceive}(\mathsf{st}_u, \widehat{\sigma}_{\mathsf{idx}(u)}, \widehat{\mathsf{ct}}_{\mathsf{idx}(u)})$
return $(b_u, v_u)_{u \in \mathcal{H}}$
```

Figure 2: Security games for (non-colluding) unforgeability, anonymity, anonymous blocklisting, and tracing soundness. We define a set of oracles $O := \{ O_{\mathsf{Send}}, O_{\mathsf{Receive}}, O_{\mathsf{UpdSend}}, O_{\mathsf{UpdReceive}} \}$ and $O^{\star} := \{ O_{\mathsf{Send}}, O_{\mathsf{GroupReceive}}, O_{\mathsf{UpdSend}}, O_{\mathsf{GroupUpdReceive}} \}$. We assume the game maintains the public parameter pp and (secret) user states st_u. Moreover, we assume the updated state st'_u is implicitly set as st_u and omit the substitution $\mathsf{st}'_u \leftarrow \mathsf{st}_u$ for readability. When the condition in **req** does not hold, we assume the game outputs a random bit in the anonymity game and 0 in all other games. Lastly, for readability, we sometimes ignore creating the lists $L_{\mathsf{tr}}, L_{\mathsf{msg}}, L_{\mathsf{upd}}$ when they are not required by the game.

includes a server secret key $\mathsf{sk}_{\mathsf{Sv}}$; our security says nothing if $\mathsf{sk}_{\mathsf{Sv}}$ is *maliciously* generated.

One way to handle this issue of transparency of the server is to enforce that the server's secret key $\mathsf{sk}_{\mathsf{Sv}}$ can be *deterministically* derived from any user state st_u. With such a restriction, any user can locally run the server's algorithms Verify and UpdVerify, and potentially audit the server's behavior. Indeed, all of the protocols proposed in this paper will have such a property as $\mathsf{sk}_{\mathsf{Sv}}$ is derived from the group secret key generated by the CGKA protocol.

4 COSMOS: Authentication with One-Time Tokens

In this section we propose a GAM protocol named COSMOS (Compact authenticated Secure Messaging with randomized One-time tokenS). When anonymity is not necessary, COSMOS is the most efficient and simplest protocol among all our proposed protocols. The additional total communication overhead is only 3κ compared to a protocol where messages are sent without any authentication, where κ is the security parameter. Additionally, we show a simple method to bootstrap COSMOS to satisfy anonymity and anonymous blocklisting, which we name COSMAC. The added overhead to COSMOS is a single MAC tag. Lastly, we show how to optimize both protocols by batching sends and updates together.

4.1 Construction of COSMOS

The high level idea is as follows: each group user mints tokens $(x_i, y_i) \in \{0,1\}^\kappa \times \{0,1\}^\kappa$ for $i \in [T]$ such that $y_i = \mathsf{OWF}(x_i)$; stores the private tokens $(x_i)_{i\in[T]}$ in its state; uploads the public tokens $(y_i)_{i\in[T]}$ to the server in an *offline* phase; and ideally sends (x_i, m_i) to the server once the message m_i is defined in an *online* phase, where x_i acts as the authentication token, and delete x_i from its state. However, since x_i is not cryptographically tied to m_i, this is insecure. Thus, the user additionally MACs (x_i, m_i) using a MAC key only known among the group users. We highlight that such a MAC key can be generated from the *common* group secret key gsk maintained by the CGKA protocol.

More formally, after the initialization phase, each user $u \in \mathsf{G}$ and server maintain a list of public tokens $\mathsf{PubTOKEN} \in (\{0,1\}^\kappa)^{NT}$, where $N = |\mathsf{G}|$ and T is the number of messages a user can send before needing to update its state. PubTOKEN is a list such that, for each user $u \in \mathsf{G}$, $\mathsf{PubTOKEN}[u] \in (\{0,1\}^\kappa)^T$ stores the T public tokens $(y_i)_{i\in[T]}$ used by user u. User u also maintains a list of private tokens $\mathsf{PrivTOKEN}_u \in (\{0,1\}^\kappa)^T$ storing the T private tokens $(x_i)_{i\in[T]}$.

To send a message m, user u retrieves an unused private token x from $\mathsf{PrivTOKEN}_u$, along with the counter $\mathsf{ctr} \in [T]$ such that $\mathsf{PubTOKEN}[u][\mathsf{ctr}] = \mathsf{OWF}(x)$, and sends $(x, \mathsf{ctr}, \Sigma_{\mathsf{MAC}})$ as the group authentication token Σ_{G} to the server, where Σ_{MAC} is a MAC tag using $\mathsf{k}_{\mathsf{MAC}}$. The server then checks if the token x is valid (i.e., $y_{\mathsf{ctr}} = \mathsf{OWF}(x)$) and relays $(x, \mathsf{ctr}, \Sigma_{\mathsf{MAC}})$ as the user authentication token σ_i to all the users. Here, Σ_{MAC} does not need to (nor can it) be verified by the server. Now, since PubTOKEN and $\mathsf{k}_{\mathsf{MAC}}$ is shared among the group, the users can verify the MAC tag and trace the user u that sent σ_i.

When only one private token x is left, user u performs a state update and mints new tokens. It generates a new batch of T tokens $(x_i, y_i)_{i\in[T]}$ and uploads $(y_i)_{i\in[T]}$ using the final token x along with a MAC tag. The server and users check that the newly minted tokens are from user u by validating x and update $\mathsf{PubTOKEN}[u] \leftarrow (y_i)_{i\in[T]}$. Once user u's state is updated, u can send T messages again. Importantly, COSMOS is *locally* state-updatable since a user can start sending messages once they update their state. We provide the formal description of COSMOS in Fig. 3.

Lastly, COSMOS satisfies all the security notions except for anonymity: non-colluding unforgeability, (anonymous) blocklisting, and tracing soundness. At a high level, we argue non-colluding unforgeability by considering two cases: against a malicious server the authentication token is unforgeable as $\mathsf{k}_{\mathsf{MAC}}$ is unknown. Importantly, the same authentication token $(x, \mathsf{ctr}, \Sigma_{\mathsf{MAC}})$ cannot be reused by the malicious server since the users have already deleted the associated public token y when it receives x the second time. Otherwise, against a malicious user, it is unforgeable as the private token x is unknown. In the latter, we use the fact that an honest server correctly processes the private token sent from an honest user (i.e., delete it from the server), preventing a malicious user from replaying it. One can check that it is not *standard* unforgeable since if a malicious server and insider collude, both $\mathsf{k}_{\mathsf{MAC}}$ and private tokens x will be known to the adversary, allowing for a trivial forgery. Moreover, we note that even though the MAC tag attached to the group authentication token cannot be verified by the server, and hence can be stealthily modified to a garbage MAC tag, this will not harm tracing soundness as we only use the private tokens for tracing. The formal security proof is deferred to App. C.1.

4.2 COSMAC: An Anonymous COSMOS with Anonymous Blocklisting

While COSMOS is efficient, it lacks anonymity. A server can link two tokens by looking at their corresponding locations in PubTOKEN. We present a simple method to transform COSMOS to have anonymity and anonymous blocklisting at an overhead of only one MAC tag. We name this GAM protocol COSMAC (COSMOS with MAC). Note that in exchange for anonymity, COSMAC loses tracing soundness.

The high level idea is for each user in the group to additionally derive a unique MAC key $\overline{\mathsf{k}_{\mathsf{MAC}}}$ and a symmetric-key encryption (SKE) key $\mathsf{k}_{\mathsf{SKE}}$ from gsk where, unlike in COSMOS, $\overline{\mathsf{k}_{\mathsf{MAC}}}$ is uploaded to the server. When a group user uploads

```
$\mathsf{Init}(1^\kappa, \mathsf{G})$
$\mathsf{gsk} \leftarrow\$ \{0,1\}^\kappa$   / Group secret key
$\mathsf{k_{MAC}} \leftarrow \mathsf{PRF}(\mathsf{gsk}, 0)$   / MAC key for group
/ Prepare empty lists
foreach $u \in \mathsf{G}$ do
    $\mathsf{PubTOKEN}_u[*] := \bot$
    $\mathsf{PrivTOKEN}_u[*] := \bot$
foreach $u \in \mathsf{G}$ do
    $(X_u, Y_u) \leftarrow$ *gen-auth-token$(\mathsf{G}, u)$
    $\mathsf{PrivTOKEN}_u \leftarrow X_u$   / $X_u \in (\{0,1\}^\kappa)^T$
    foreach $v \in \mathsf{G}$ do
        $\mathsf{PubTOKEN}_v[u] \leftarrow Y_u$   / $Y_u \in (\{0,1\}^\kappa)^T$
foreach $u \in \mathsf{G}$ do
    $\mathsf{TOKEN}_u := (\mathsf{PrivTOKEN}_u, \mathsf{PubTOKEN}_u)$
    $\mathsf{st}_u \leftarrow (\mathsf{G}, \mathsf{k_{MAC}}, 1, \mathsf{TOKEN}_u)$
$\mathsf{DB}[*] := \bot$   / Prepare empty database for Sv
for $u \in \mathsf{G}$ do
    $\mathsf{DB}[u] \leftarrow Y_u$
$\mathsf{pp} \leftarrow \mathsf{DB}$   / $\mathsf{DB} \in (\{0,1\}^\kappa)^{N \times T}$
return $\left(\mathsf{pp}, (\mathsf{st}_u)_{u \in \mathsf{G}}\right)$
```

```
$\mathsf{Send}(\mathsf{st}_u, \mathsf{m})$
parse $(\mathsf{G}, \mathsf{k_{MAC}}, \mathsf{ctr}, \mathsf{TOKEN}_u) \leftarrow \mathsf{st}_u$
if $\mathsf{ctr} \geq T - 1$ then return $\bot$   / Need to update tokens
$\mathsf{ctr}' \leftarrow \mathsf{ctr} + 1$
$\Sigma_\mathsf{G} \leftarrow$ *attach-auth-token$(\mathsf{st}_u, \mathsf{m})$
$\mathsf{st}'_u \leftarrow (\mathsf{G}, \mathsf{k_{MAC}}, \mathsf{ctr}', \mathsf{TOKEN}_u)$
return $(\mathsf{st}'_u, \Sigma_\mathsf{G})$
```

```
$\mathsf{Verify}(\mathsf{pp}, \Sigma_\mathsf{G}, \mathsf{m})$
parse $\mathsf{DB} \leftarrow \mathsf{pp}$
try $(\mathsf{pp}', (\sigma_v)_{v \in \mathsf{G}}) \leftarrow$ *verify-auth-token$(\mathsf{pp}, \Sigma_\mathsf{G})$
return $(\mathsf{pp}', (\sigma_v)_{v \in \mathsf{G}})$
```

```
$\mathsf{Receive}(\mathsf{st}_u, \sigma, \mathsf{m})$
try $(\mathsf{st}'_u, b, v) \leftarrow$ *trace-sender$(\mathsf{st}_u, \sigma, \mathsf{m})$
return $(\mathsf{st}'_u, b, v)$
```

```
$\mathsf{UpdSend}(\mathsf{st}_u)$
parse $(\mathsf{G}, \mathsf{k_{MAC}}, \mathsf{ctr}, \mathsf{TOKEN}_u) \leftarrow \mathsf{st}_u$
parse $(\mathsf{PrivTOKEN}_u, \mathsf{PubTOKEN}_u) \leftarrow \mathsf{TOKEN}_u$
$(X_u, Y_u) \leftarrow$ *gen-auth-token$(\mathsf{G}, u)$
$\mathsf{PrivTOKEN}_u \leftarrow X_u$
$\mathsf{PubTOKEN}_u[u] \leftarrow Y_u$
$\mathsf{TOKEN}_u \leftarrow (\mathsf{PrivTOKEN}_u, \mathsf{PubTOKEN}_u)$
/ Refresh counter to 1
$\mathsf{st}'_u \leftarrow (\mathsf{G}, \mathsf{k_{MAC}}, 1, \mathsf{TOKEN}_u)$
$\widehat{\mathsf{ct}}_\mathsf{G} \leftarrow (u, Y_u)$
$\widehat{\Sigma}_\mathsf{G} \leftarrow$ *attach-auth-token$(\mathsf{st}_u, \widehat{\mathsf{ct}}_\mathsf{G})$
return $(\mathsf{st}'_u, \widehat{\Sigma}_\mathsf{G}, \widehat{\mathsf{ct}}_\mathsf{G})$
```

```
$\mathsf{UpdVerify}(\mathsf{pp}, \widehat{\Sigma}_\mathsf{G}, \widehat{\mathsf{ct}}_\mathsf{G})$
parse $\mathsf{DB} \leftarrow \mathsf{pp}$
try $(\mathsf{pp}', (\widehat{\sigma}_v)_{v \in \mathsf{G}}) \leftarrow$ *verify-auth-token$(\mathsf{pp}, \widehat{\Sigma}_\mathsf{G})$
parse $(u, Y_u) \leftarrow \widehat{\mathsf{ct}}_\mathsf{G}$
$\mathsf{DB}[u] \leftarrow Y_u$
foreach $v \in \mathsf{G}$ do
    $\widehat{\mathsf{ct}}_v \leftarrow (u, Y_u)$
return $(\mathsf{pp}', (\widehat{\sigma}_v, \widehat{\mathsf{ct}}_v)_{v \in \mathsf{G}})$
```

```
$\mathsf{UpdReceive}(\mathsf{st}_u, \widehat{\sigma}, \widehat{\mathsf{ct}})$
parse $(\mathsf{G}, \mathsf{k_{MAC}}, \mathsf{ctr}, \mathsf{TOKEN}_u) \leftarrow \mathsf{st}_u$
parse $(\mathsf{PrivTOKEN}_u, \mathsf{PubTOKEN}_u) \leftarrow \mathsf{TOKEN}_u$
try $(\mathsf{st}_u, b, v) \leftarrow$ *trace-sender$(\mathsf{st}_u, \widehat{\sigma}, \widehat{\mathsf{ct}})$
parse $(v', Y) \leftarrow \widehat{\mathsf{ct}}$
req $v = v'$
$\mathsf{PubTOKEN}_u[v] \leftarrow Y$   / Update user $v$'s tokens
$\mathsf{TOKEN}_u \leftarrow (\mathsf{PrivTOKEN}_u, \mathsf{PubTOKEN}_u)$
$\mathsf{st}'_u \leftarrow (\mathsf{G}, \mathsf{k_{MAC}}, \mathsf{ctr}, \mathsf{TOKEN}_u)$
return $(\mathsf{st}'_u, b, v)$
```

Figure 3: `COSMOS`: A group authenticated messaging protocol with one-time tokens. The server Sv is assumed to implicitly apply the inverse of idx, i.e., it uses $u \in \mathsf{G}$ rather than $i = \mathsf{idx}(u) \in [N]$. The helper algorithms used above are detailed in Fig. 4.

```
Func *gen-auth-token($\mathsf{G}, u$)
    $X[*], Y[*] := \bot$
    foreach $t \in [T]$ do
        $X[t] \leftarrow\$ \{0,1\}^\kappa$
        $Y[t] \leftarrow \mathsf{OWF}(X[t])$
    return $(X, Y)$
```

```
Func *attach-auth-token($\mathsf{st}_u, \mathsf{m}$)
    parse $(\mathsf{G}, \mathsf{k}_{\mathsf{MAC}}, \mathsf{ctr}, \mathsf{TOKEN}_u) \leftarrow \mathsf{st}_u$
    parse $(\mathsf{PrivTOKEN}_u, \mathsf{PubTOKEN}_u) \leftarrow \mathsf{TOKEN}_u$
    $x_u^{(\mathsf{ctr})} \leftarrow \mathsf{PrivTOKEN}_u[\mathsf{ctr}]$
    $\mathsf{PrivTOKEN}_u[\mathsf{ctr}] \leftarrow \bot$
    $\mathsf{TOKEN}_u \leftarrow (\mathsf{PrivTOKEN}_u, \mathsf{PubTOKEN}_u)$
    $\Sigma_{\mathsf{MAC}} \leftarrow\$ \mathsf{MAC.TagGen}(\mathsf{k}_{\mathsf{MAC}}, (u, \mathsf{ctr}, x_u^{(\mathsf{ctr})}, \mathsf{m}))$
    return $\left(u, \mathsf{ctr}, x_u^{(\mathsf{ctr})}, \Sigma_{\mathsf{MAC}}\right)$
```

```
Func *verify-auth-token($\mathsf{pp}, \Sigma_{\mathsf{G}}$)
    $\mathsf{DB} \leftarrow \mathsf{pp}$    / $\mathsf{DB} \in (\{0,1\}^\kappa)^{N \times T}$
    parse $(u, \mathsf{ctr}, x, \Sigma_{\mathsf{MAC}}) \leftarrow \Sigma_{\mathsf{G}}$
    if $\mathsf{DB}[u][\mathsf{ctr}] \neq \mathsf{OWF}(x)$ then
        return $\bot$
    / If check passes, set user authentication tokens and
    / delete entry from DB
    foreach $v \in \mathsf{G}$ do
        $\sigma_v \leftarrow (u, \mathsf{ctr}, x, \Sigma_{\mathsf{MAC}})$
    $\mathsf{DB}[u][\mathsf{ctr}] \leftarrow \bot$
    $\mathsf{pp}' \leftarrow \mathsf{DB}$
    return $(\mathsf{pp}', (\sigma_v)_{v \in \mathsf{G}})$
```

```
Func *trace-sender($\mathsf{st}_u, \sigma, \mathsf{m}$)
    parse $(\mathsf{G}, \mathsf{k}_{\mathsf{MAC}}, \mathsf{ctr}, \mathsf{TOKEN}_u) \leftarrow \mathsf{st}_u$
    parse $(\mathsf{PrivTOKEN}_u, \mathsf{PubTOKEN}_u) \leftarrow \mathsf{TOKEN}_u$
    parse $(v, \mathsf{ctr}_v, x, \Sigma_{\mathsf{MAC}}) \leftarrow \sigma$
    if $\mathsf{PubTOKEN}_u[v][\mathsf{ctr}_v] \neq \mathsf{OWF}(x)$
            $\vee\ \mathsf{MAC.Verify}(\mathsf{k}_{\mathsf{MAC}}, (v, \mathsf{ctr}_v, x, \mathsf{m}), \Sigma_{\mathsf{MAC}}) = \bot$ then
        return $(\mathsf{st}_u, \bot, \bot)$
    $\mathsf{PubTOKEN}_u[v][\mathsf{ctr}_v] \leftarrow \bot$
    $\mathsf{TOKEN}_u \leftarrow (\mathsf{PrivTOKEN}_u, \mathsf{PubTOKEN}_u)$
    $\mathsf{st}'_u \leftarrow (\mathsf{G}, \mathsf{k}_{\mathsf{MAC}}, \mathsf{ctr}, \mathsf{TOKEN}_u)$
    return $(\mathsf{st}'_u, \top, v)$
```

Figure 4: Helper functions used by `COSMOS`.

some content to the server, it runs the Send (resp. $\mathsf{UpdSend}$) algorithm of COSMOS, encrypts the group (resp. update) authentication token using $\mathsf{k}_{\mathsf{SKE}}$, and MACs the ciphertext with $\overline{\mathsf{k}_{\mathsf{MAC}}}$. The server only accepts contents that have a valid tag under $\overline{\mathsf{k}_{\mathsf{MAC}}}$. A group user can verify the user authentication token by first decrypting the ciphertext using $\mathsf{k}_{\mathsf{SKE}}$, followed by the same check as COSMOS. The protocol is formally given in Fig. 5. The protocol consists of COSMOS and a wrapper protocol that encrypts and MACs the authentication tokens output by COSMOS. The main difference between COSMAC and COSMOS is highlighted with a box in Fig. 5.

Observe that the authentication tokens are now encrypted and the server no longer learns the identity of the user. This is how anonymity is achieved. Non-colluding unforgeability almost immediately follows from the non-colluding unforgeability of COSMOS. This is because from the users point of view, COSMAC and COSMOS are almost identical. The only difference is that COSMAC requires to first perform a decryption using $\mathsf{k}_{\mathsf{SKE}}$; this does not make forging anymore easier for the adversary. Indeed, prove non-colluding unforgeability of COSMAC assuming the non-colluding unforgeability of COSMOS. Moreover, COSMAC satisfies anonymous blocklisting since an outsider without knowledge of $\overline{\mathsf{k}_{\mathsf{MAC}}}$ cannot upload contents which the server will accept. Lastly, on the other hand, unlike COSMOS, a malicious user can now stealthily perform a DoS attack on the group since the server can only check the validity of the MAC tag and not the content. In particular, COSMAC loses tracing soundness, as the content, which can now be a malformed ciphertext, may not include the sender's identity. We show in App. C.2 that COSMAC is non-colluding unforgeable, anonymous, and anonymous blocklistable.

4.3 Optimizations of COSMOS and COSMAC

We take advantage of the fact that COSMOS (and COSMAC) have an efficient *local* state update and apply two optimizations leading to $\texttt{COSMOS}^{+}$ and $\texttt{COSMOS}^{++}$ (and $\texttt{COSMAC}^{+}$ and $\texttt{COSMAC}^{++}$, respectively). We focus on COSMOS as the case for COSMAC is almost identical.

Removing Local Updates: $\texttt{COSMOS}^{+}$**.** Notice that local state-updates allow a user to execute the $\mathsf{UpdSend}$ algorithm as part of the Send algorithm. That is, users can send a message and perform an update *at the same time*. Concretely, this requires to maintain just one public token y per user $u \in \mathsf{G}$. To send a message m, u first mints a new token (x', y') and uploads both the message and public token (m, y') using the private token x, along with a MAC tag binding (m, x, y') together. The server and other group members replace y with y'. User u can repeat the process using the new private token x'. Effectively, the protocol now consists of only running an online phase, since the update is implicitly performed during a send. Compared to COSMOS, $\texttt{COSMOS}^{+}$ balances the throughput of the user without harming the total communication cost $3 \cdot \kappa$, while also reducing the storage cost of public tokens.

Minimizing Communication Cost: $\texttt{COSMOS}^{++}$**.** This optimization reduces the communication cost of $\texttt{COSMOS}^{+}$ by $1/3$ while keeping the local update of COSMOS.[9] The main idea is to make the private authentication token become the public token for the next message. As in $\texttt{COSMOS}^{+}$, the server maintains a single public token $y_{i,c}$ per user, where $c \in \mathbb{N}$ will be the number of times user u ran $\mathsf{UpdSend}$. As a result of running $\mathsf{UpdSend}$ the user will upload a public token $y_{0,c} = \mathsf{OWF}^{T}(x_{T,c})$, i.e., T^{th} invocation of OWF. This updated public token can be used to send T messages. To authenticate the i-th message m_i, user u simply sends token $x_{i,c} = \mathsf{OWF}^{T-i}(x_{T,c})$ along with a MAC tag on $(m_i, x_{i,c})$. The server and other group members update the public token to $y_{i,c} := x_{i,c}$. Since the public token generated in the offline phase is useful for sending T messages, the amortized cost of sending one message is $(2 + \frac{1}{T}) \cdot \kappa$. For a sufficiently large T, this reduces the communication cost of $\texttt{COSMOS}^{+}$ by $1/3$.

5 Anonymous and Tracing Sound GAMs

In this section we introduce three GAM protocols that simultaneously achieve anonymity and tracing soundness. These are the first authentication modes in the literature to do so. The first two protocols: QUASAR (Quick Authenticated Secure *Anonymous* messaging with Randomized one-time tokens) and STARS (Strongly-Authenticated anonymous messaging with Randomized one-time Signatures) satisfy these stronger authenticity guarantees at the cost of being only *global* as opposed to *local* state-updatable like COSMOS and COSMAC. Once a user $u \in \mathsf{G}$ exhausts its private tokens, it must wait till all other users perform an update before being able to send a message again. We discuss some ideas to mitigate the shortcoming of global state updates in App. D.3. Our third protocol GEMSTARS (Group Signature Modified STARS), eliminates updates altogether by relying on group signatures. Below we give intuitive overviews of the protocols, deferring the formal descriptions and security proofs to Apps. D and E.

5.1 QUASAR: Anonymous Authentication with Tokens

We first consider a non-anonymous variant of QUASAR and add anonymity later. Its core idea is to perform a relatively expensive *offline* phase (i.e., $\mathsf{UpdSend}$) to make the *online* phase (i.e., Send) very cheap.

Basic Idea. Assume a group $\mathsf{G} = (u_i)_{i \in [N]}$. Each user u_i mints tokens $(x^{(t)}_{j \to i}, y^{(t)}_{j \to i}) \in \{0,1\}^{\kappa} \times \{0,1\}^{\kappa}$ for $(j,t) \in [N] \times [T]$ such that $y^{(t)}_{j \to i} = \mathsf{OWF}(x^{(t)}_{j \to i})$. Here, $j \to i$ indicates that user

---

[9] This optimization was suggested to us by an anonymous reviewer; afterwards, another reviewer informed us that a similar idea is used in the S/KEY one-time password authentication protocol [48, 49].

```
$\mathsf{Init}(1^\kappa, \mathsf{G})$
$\mathsf{gsk} \leftarrow\$ \{0,1\}^\kappa$   / Group secret key
$\mathsf{k_{MAC}} \leftarrow \mathsf{PRF}(\mathsf{gsk}, 0)$   / MAC key for group
$\boxed{\mathsf{sk_{Sv}} := \overline{\mathsf{k_{MAC}}} \leftarrow \mathsf{PRF}(\mathsf{gsk}, 1) \quad \text{/ MAC key for Sv and group}}$
$\boxed{\mathsf{k_{SKE}} \leftarrow \mathsf{PRF}(\mathsf{gsk}, 2) \quad \text{/ SKE key for group}}$
/ Prepare empty lists
foreach $u \in \mathsf{G}$ do
    $\mathsf{PubTOKEN}_u[*] := \bot$
    $\mathsf{PrivTOKEN}_u[*] := \bot$
foreach $u \in \mathsf{G}$ do
    $(X_u, Y_u) \leftarrow$ *gen-auth-token$(\mathsf{G}, u)$
    $\mathsf{PrivTOKEN}_u \leftarrow X_u$   / $X_u \in (\{0,1\}^\kappa)^T$
    foreach $v \in \mathsf{G}$ do
        $\mathsf{PubTOKEN}_v[u] \leftarrow Y_u$   / $Y_u \in (\{0,1\}^\kappa)^T$
foreach $u \in \mathsf{G}$ do
    $\mathsf{TOKEN}_u := (\mathsf{PrivTOKEN}_u, \mathsf{PubTOKEN}_u)$
    $\mathsf{st}_u \leftarrow \left(\mathsf{G}, \mathsf{k_{MAC}}, \boxed{\overline{\mathsf{k_{MAC}}}, \mathsf{k_{SKE}}}, 1, \mathsf{TOKEN}_u\right)$
$\boxed{\mathsf{pp} := \bot \quad \text{/ No pp maintained by Sv}}$
return $(\mathsf{pp}, \boxed{\mathsf{sk_{Sv}}}, (\mathsf{st}_u)_{u \in \mathsf{G}})$
```

```
$\mathsf{Send}(\mathsf{st}_u, \mathsf{m})$
parse $\left(\mathsf{G}, \mathsf{k_{MAC}}, \boxed{\overline{\mathsf{k_{MAC}}}, \mathsf{k_{SKE}}}, \mathsf{ctr}, \mathsf{TOKEN}_u\right) \leftarrow \mathsf{st}_u$
if $\mathsf{ctr} \geq T - 1$ then return $\bot$   / Need to update tokens
$\mathsf{ctr}' \leftarrow \mathsf{ctr} + 1$
$\Sigma_\mathsf{G} \leftarrow$ *attach-auth-token$(\mathsf{st}_u, \mathsf{m})$
$\mathsf{st}'_u \leftarrow \left(\mathsf{G}, \mathsf{k_{MAC}}, \boxed{\overline{\mathsf{k_{MAC}}}, \mathsf{k_{SKE}}}, \mathsf{ctr}', \mathsf{TOKEN}_u\right)$
return $(\mathsf{st}'_u, \Sigma_\mathsf{G})$
```

```
$\mathsf{Verify}(\mathsf{pp}, \boxed{\mathsf{sk_{Sv}}}, \Sigma_\mathsf{G}, \mathsf{m})$
try $(\bot, (\sigma_i)_{i \in [N]})$
        $\leftarrow$ *verify-auth-token$(\bot, \boxed{\mathsf{sk_{Sv}}}, \Sigma_\mathsf{G})$
return $(\mathsf{pp}', (\sigma_i)_{i \in [N]})$
```

```
$\mathsf{Receive}(\mathsf{st}_u, \sigma, \mathsf{m})$
try $(\mathsf{st}'_u, b, v) \leftarrow$ *trace-sender$(\mathsf{st}_u, \sigma, \mathsf{m})$
return $(\mathsf{st}'_u, b, v)$
```

```
$\mathsf{UpdSend}(\mathsf{st}_u)$
parse $\left(\mathsf{G}, \mathsf{k_{MAC}}, \boxed{\overline{\mathsf{k_{MAC}}}, \mathsf{k_{SKE}}}, \mathsf{ctr}, \mathsf{TOKEN}_u\right) \leftarrow \mathsf{st}_u$
parse $(\mathsf{PrivTOKEN}_u, \mathsf{PubTOKEN}_u) \leftarrow \mathsf{TOKEN}_u$
$(X_u, Y_u) \leftarrow$ *gen-auth-token$(\mathsf{G}, u)$
$\widehat{\Sigma}_\mathsf{G} \leftarrow$ *attach-auth-token$(\mathsf{st}_u, Y_u)$
$\mathsf{PrivTOKEN}_u \leftarrow X_u$
$\mathsf{PubTOKEN}_u[u] \leftarrow Y_u$
$\mathsf{TOKEN}_u \leftarrow (\mathsf{PrivTOKEN}_u, \mathsf{PubTOKEN}_u)$
/ Refresh counter to 1
$\mathsf{st}'_u \leftarrow \left(\mathsf{G}, \mathsf{k_{MAC}}, \boxed{\overline{\mathsf{k_{MAC}}}, \mathsf{k_{SKE}}}, 1, \mathsf{TOKEN}_u\right)$
$\boxed{\widehat{\mathsf{ct}}_\mathsf{SKE} \leftarrow \mathsf{Enc}(\mathsf{k_{SKE}}, (u, Y_u))}$
$\boxed{\widehat{\mathsf{ct}}_\mathsf{G} \leftarrow \widehat{\mathsf{ct}}_\mathsf{SKE}}$
return $(\mathsf{st}'_u, \widehat{\Sigma}_\mathsf{G}, \widehat{\mathsf{ct}}_\mathsf{G})$
```

```
$\mathsf{UpdVerify}(\mathsf{pp}, \boxed{\mathsf{sk_{Sv}}}, \widehat{\Sigma}_\mathsf{G}, \widehat{\mathsf{ct}}_\mathsf{G})$
try $(\bot, (\widehat{\sigma}_i)_{i \in [N]}) \leftarrow$ *verify-auth-token$(\bot, \boxed{\mathsf{sk_{Sv}}}, \widehat{\Sigma}_\mathsf{G})$
foreach $i \in [N]$ do
    $\boxed{\widehat{\mathsf{ct}}_i \leftarrow \widehat{\mathsf{ct}}_\mathsf{G}}$
return $(\mathsf{pp}', (\widehat{\sigma}_i, \widehat{\mathsf{ct}}_i)_{i \in [N]})$
```

```
$\mathsf{UpdReceive}(\mathsf{st}_u, \widehat{\sigma}, \widehat{\mathsf{ct}})$
parse $\left(\mathsf{G}, \mathsf{k_{MAC}}, \boxed{\overline{\mathsf{k_{MAC}}}, \mathsf{k_{SKE}}}, \mathsf{ctr}, \mathsf{TOKEN}_u\right) \leftarrow \mathsf{st}_u$
parse $(\mathsf{PrivTOKEN}_u, \mathsf{PubTOKEN}_u) \leftarrow \mathsf{TOKEN}_u$
$\boxed{\textbf{parse } \widehat{\mathsf{ct}}_\mathsf{SKE} \leftarrow \widehat{\mathsf{ct}}}$
$\boxed{(v', Y) \leftarrow \mathsf{SKE.Dec}(\mathsf{k_{SKE}}, \widehat{\mathsf{ct}}_\mathsf{SKE})}$
try $(\mathsf{st}_u, b, v) \leftarrow$ *trace-sender$(\mathsf{st}_u, \widehat{\sigma}, Y)$
req $v = v'$
$\mathsf{PubTOKEN}_u[v] \leftarrow Y$   / Update user $v$'s tokens
$\mathsf{TOKEN}_u \leftarrow (\mathsf{PrivTOKEN}_u, \mathsf{PubTOKEN}_u)$
$\mathsf{st}'_u \leftarrow \left(\mathsf{G}, \mathsf{k_{MAC}}, \boxed{\overline{\mathsf{k_{MAC}}}, \mathsf{k_{SKE}}}, \mathsf{ctr}, \mathsf{TOKEN}_u\right)$
return $(\mathsf{st}'_u, \top, v)$
```

Figure 5: `COSMAC`: An anonymous group authenticated messaging protocol with one-time tokens. The main differences between `COSMOS` is highlighted by a box. The helper algorithms used above are detailed in Fig. 6.

```
Func *gen-auth-token$(\mathsf{G}, u)$
$X[*], Y[*] := \bot$
foreach $t \in [T]$ do
    $X[t] \leftarrow\$ \{0,1\}^{\kappa}$
    $Y[t] \leftarrow \mathsf{OWF}(X[t])$
return $(X, Y)$
```

```
Func *attach-auth-token$(\mathsf{st}_u, \mathsf{m})$
parse $\left(\mathsf{G}, \mathsf{k}_{\mathsf{MAC}}, \boxed{\overline{\mathsf{k}_{\mathsf{MAC}}}, \mathsf{k}_{\mathsf{SKE}}}, \mathsf{ctr}, \mathsf{TOKEN}_u\right) \leftarrow \mathsf{st}_u$
parse $(\mathsf{PrivTOKEN}_u, \mathsf{PubTOKEN}_u) \leftarrow \mathsf{TOKEN}_u$
$x_u^{(\mathsf{ctr})} \leftarrow \mathsf{PrivTOKEN}_u[\mathsf{ctr}]$
$\mathsf{PrivTOKEN}_u[\mathsf{ctr}] \leftarrow \bot$
$\mathsf{TOKEN}_u \leftarrow (\mathsf{PrivTOKEN}_u, \mathsf{PubTOKEN}_u)$
$\Sigma_{\mathsf{MAC}} \leftarrow\$ \mathsf{MAC.TagGen}(\mathsf{k}_{\mathsf{MAC}}, (u, \mathsf{ctr}, x_u^{(\mathsf{ctr})}, \mathsf{m}))$
$\boxed{\mathsf{ct}_{\mathsf{SKE}} \leftarrow\$ \mathsf{SKE.Enc}\left(\mathsf{k}_{\mathsf{SKE}}, \left(u, \mathsf{ctr}, x_u^{(\mathsf{ctr})}, \Sigma_{\mathsf{MAC}}\right)\right)}$
$\boxed{\overline{\Sigma}_{\mathsf{MAC}} \leftarrow\$ \mathsf{MAC.TagGen}(\overline{\mathsf{k}_{\mathsf{MAC}}}, \mathsf{ct}_{\mathsf{SKE}})}$
return $\boxed{(\mathsf{ct}_{\mathsf{SKE}}, \overline{\Sigma}_{\mathsf{MAC}})}$
```

```
Func *verify-auth-token$(\mathsf{pp} = \bot, \boxed{\mathsf{sk}_{\mathsf{Sv}}}, \Sigma_{\mathsf{G}})$
$\boxed{\textbf{parse}\ (\mathsf{ct}_{\mathsf{SKE}}, \overline{\Sigma}_{\mathsf{MAC}}) \leftarrow \Sigma_{\mathsf{G}}}$
$\boxed{\textbf{if}\ \mathsf{MAC.Verify}(\mathsf{sk}_{\mathsf{Sv}}, \mathsf{ct}_{\mathsf{SKE}}, \overline{\Sigma}_{\mathsf{MAC}}) = \bot\ \textbf{then}}$
    return $\bot$
/ If check passes, set user authentication tokens
foreach $i \in [N]$ do
    $\boxed{\sigma_i \leftarrow \mathsf{ct}_{\mathsf{SKE}}}$
return $(\bot, (\sigma_i)_{i \in [N]})$
```

```
Func *trace-sender$(\mathsf{st}_u, \sigma, \mathsf{m})$
parse $\left(\mathsf{G}, \mathsf{k}_{\mathsf{MAC}}, \boxed{\overline{\mathsf{k}_{\mathsf{MAC}}}, \mathsf{k}_{\mathsf{SKE}}}, \mathsf{ctr}, \mathsf{TOKEN}_u\right) \leftarrow \mathsf{st}_u$
parse $(\mathsf{PrivTOKEN}_u, \mathsf{PubTOKEN}_u) \leftarrow \mathsf{TOKEN}_u$
$\boxed{\textbf{parse}\ \mathsf{ct}_{\mathsf{SKE}} \leftarrow \sigma}$
$\boxed{(v, \mathsf{ctr}_v, x, \Sigma_{\mathsf{MAC}}) \leftarrow \mathsf{SKE.Dec}(\mathsf{k}_{\mathsf{SKE}}, \mathsf{ct}_{\mathsf{SKE}})}$
if $\mathsf{PubTOKEN}_u[v][\mathsf{ctr}_v] \neq \mathsf{OWF}(x)$
        $\vee\ \mathsf{MAC.Verify}(\mathsf{k}_{\mathsf{MAC}}, (v, \mathsf{ctr}_v, x, \mathsf{m}), \Sigma_{\mathsf{MAC}}) = \bot$ then
    return $(\mathsf{st}_u, \bot, \bot)$
$\mathsf{PubTOKEN}_u[v][\mathsf{ctr}_v] \leftarrow \bot$
$\mathsf{TOKEN}_u \leftarrow (\mathsf{PrivTOKEN}_u, \mathsf{PubTOKEN}_u)$
$\mathsf{st}'_u \leftarrow \left(\mathsf{G}, \mathsf{k}_{\mathsf{MAC}}, \boxed{\overline{\mathsf{k}_{\mathsf{MAC}}}, \mathsf{k}_{\mathsf{SKE}}}, \mathsf{ctr}, \mathsf{TOKEN}_u\right)$
return $(\mathsf{st}'_u, \top, v)$
```

Figure 6: Helper functions used by `COSMAC`. The main differences between `COSMOS` is highlighted by a box.

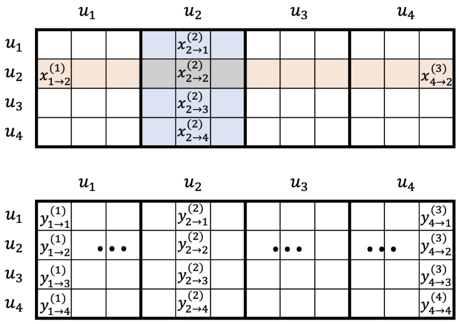

Figure 7: A toy example of a non-anonymous variant of `QUASAR`. $\mathsf{G} = (u_i)_{i\in[4]}$ and $T = 3$. The upper box stores the private tokens $(x^{(t)}_{i\to j})_{t\in[3]}$ for $i, j \in [4]$. The blue columns are tokens that user u_2 uses to send messages and the red rows are tokens that user u_2 minted. The bottom box stores the public tokens $(y^{(t)}_{i\to j})_{t\in[3]}$ for $i, j \in [4]$ held by the server.

u_i approves u_j to send a message to him. User u_i sends the private tokens $(x^{(t)}_{j\to i})_{t\in[T]}$ to u_j by encrypting it with a public-key encryption (PKE) scheme using u_j's public key. Moreover, u_i uploads the public tokens $(y^{(t)}_{j\to i})_{(j,t)\in[N]\times[T]}$ to the server. A toy example is provided in Fig. 7. Throughout the protocol, each user u_i maintains two types of tokens: *sender* tokens $(x^{(t)}_{i\to j})_{(j,t)\in[N]\times[T]}$ (blue box in Fig. 7) and *receiver* tokens $(x^{(t)}_{j\to i})_{(j,t)\in[N]\times[T]}$ (red box in Fig. 7). To send the t^*-th ($t^* < T$) message m, u_i retrieves the sender tokens $(x^{(t^*)}_{i\to j})_{j\in[N]}$ and uploads this as the group authentication token Σ_{G} along with m. Similarly to `COSMOS` and `COSMAC`, Σ_{G} will include N MAC tags for each message tuple $(x^{(t^*)}_{i\to j}, \mathsf{m})$, binding the tokens to m. The server checks that Σ_{G} maps to a specific column of public tokens in its database. If so, it parses Σ_{G}, sets the user authentication token σ_j for user u_j as $x^{(t^*)}_{i\to j}$ and the j-th MAC tag. User u_j can verify and trace σ_j back to u_i by searching through its receiver tokens. Once the users exhaust their tokens, they perform an update by minting new one-time tokens and distributing them to the group as done above. Note that this is where we require global state updates: once the boxes in Fig. 7 become empty, every user has to update in order to fill them back again.

Informally, user traceability holds since the group authentication token Σ_{G} corresponds to a unique column in the database held by the server. All honest users can use this unique column index to trace the same sender.

Amortized Efficiency. To reduce the communication cost of the offline phase and make the dependence on the ciphertext size minimal (particularly important in the post-quantum regime), we replace the private tokens $(x^{(t)}_{j\to i})_{t\in[T]}$ by one PRF seed $\mathsf{seed}_{j\to i}$, allowing each user to locally derive the corresponding tokens. This reduces the number of ciphertexts by a factor of T, making the overhead in the offline communication cost to be $2\cdot\left(\frac{|\mathsf{ct}|}{T} + \kappa\right)$ per message, where ct denotes a PKE ciphertext and κ is the bit-length of the public tokens.

Adding Anonymity. Fig. 7 illustrates how if, e.g., u_2 sends two messages using private tokens from the blue box, the server will be able to link these two messages together. Our final description of `QUASAR` fixes this by permuting the column indices using a permutation key derived from the group secret key. The server stills checks the group authentication tokens Σ_{G} with respect to a single column, and the users can map this randomly permuted column index to a unique sender.

Similarly to all the GAM protocols so far, `QUASAR` is non-colluding unforgeable since each users maintain a database of the public token and user pair. The added complexity only comes from adding tracing soundness while maintaining anonmity. Moreover, `QUASAR` is not standard unforgeable as a malicious server and malicious insider can collude to impersonate any honest user.

5.2 `STARS` and `GEMSTARS`

<u>`STARS`</u>: This is almost equivalent to `QUASAR`, except that it additionally achieves *standard* unforgeability by replacing the usage of one-time tokens (i.e., private and public tokens (x,y) such that $\mathsf{OWF}(x) = y$) with *one-time signatures* (OTS). Importantly, the usage must remain *one-time* as otherwise two messages sent with the same signing key becomes linkable.
<u>`GEMSTARS`</u>: This is essentially `STARS` without state updates. The main idea is to use a *group signature* (GS) [13, 27, 35].[10] Informally, a GS consists of three entity types: a group manager, group tracers (also referred to as *opener*), and users. A group manager is unique and handles the registration of a new user to the system. When a user u wishes to join the system, the group manager provides u with a certificate cert_u attesting to the fact that u is a valid user in the system. User u can specify a group tracer I and use cert_u to *anonymously* sign a message on behalf of all the users in the system. The signature can be publicly verifiable, but importantly, it remains anonymous to any entity (including the group manager) except to the specified group tracer I, who can *trace* the signature back to user u.

An initial attempt to build a secure GAM protocol from a GS is as follows. The server and users of a GAM protocol are mapped to the group manager and users of a GS, respectively. Since the group users in G should be the only users that can trace the signature, we then would like to map each group G in the GAM protocol to a group tracer I. We achieve this by noticing that each group G shares a common group secret key gsk. Thus, the group generates the keys $(\mathsf{gtvk}_I, \mathsf{gtsk}_I)$ for the group tracer I by executing the key generation algorithm of the group tracer using randomness derived from gsk. When a user sends a message to the group G, it runs the signing

---

[10] Since we consider multiple group tracers, we can view it as an accountable ring signature as well [18, 80].

algorithm of GS by specifying gtvk_I as the group tracer's key. The server then relays the message to the group G only if the signature verifies. The group users can locally run the tracing algorithm of I on the signature since they hold the group tracer's secret key gtsk_I.

This GAM protocol inherits the anonymity and unforgeability of GS. Unfortunately, it does not achieve anonymous blocklisting nor tracing soundness. The issue is that *any* user in the system, even those outside of the group G, can sign on behalf of G — this stems from the mismatch in the scope of "groups" considered by the GAM protocol and GS. In particular, an adversary can mount the following Sybil attack on the group: The adversary creates fake users u' and obtains credentials $\mathsf{cert}_{u'}$ by joining the system. It then specifies gtvk_I as the group tracer's key and signs on behalf of the group G. While traceability of GS allows the group users to trace back to this outsider u', the server cannot block u' from uploading malicious contents to the group since it is a valid group signature.

We resolve this issue by proving group membership in a different layer, similarly to the metadata-hiding protocol of [53] and what we did with `COSMAC`. Using the group secret key gsk, the group further generates a unique signature key pair $(\mathsf{vk}, \mathsf{sk})$ and uploads vk to the server. To send a message to the group, a group user generates the group signature as before, but further attaches a signature generated by sk. Similarly to `COSMAC`, this additional signature guarantees anonymous blocklisting, while GS guarantees that we can trace the signer to a group user. Lastly, since GS allows to sign an unbounded number of messages, `GEMSTARS` requires no updates.

Although `GEMSTARS` removes state updates while satisfying all the desired security, it is quite inefficient in the post-quantum setting due to the lack of efficient GS.

6 Running GAM Protocols on MLS

In this section, we explain how to integrate a GAM protocol into MLS. While this is fairly straightforward, some discussion is required since our GAM protocol assumes a trusted initialization algorithm Init that prepares the group user states. For the particular GAM protocol (i.e., Enc-Sign mode) currently used by MLS this is not an issue, since the initial user state (i.e., the group secret key and verification keys of the group users) is implicitly provided by the CGKA protocol. However, in general, this may not be the case.

Below, we explain this integration in two steps. We first consider the base case where a GAM protocol operates on an already established set of user states. We then consider how our *specific* GAM protocols can handle the Init algorithm without a trusted setup. Recall in MLS, a single user u initializes a group $\mathsf{G} = \{u\}$ and then dynamically adds users by sending welcome messages. We follow this approach and explain how the Init algorithm of a GAM protocol can be implemented through these dynamic group operations. As explained in Sec. 2.4, a formal model for allowing dynamic groups in GAM protocols is left as an important future work.

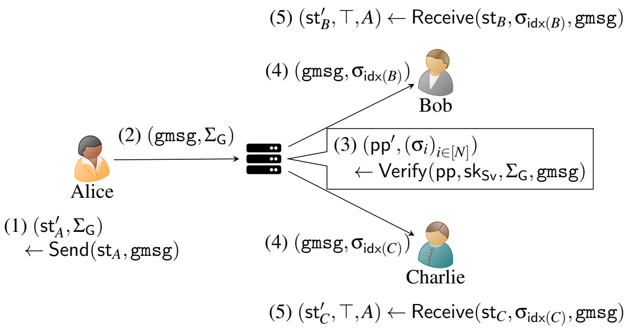

Figure 8: Using a GAM protocol on top of MLS' FSPD protocol. When retrieving a message, a user u specifies its index $\mathsf{idx}(u)$ to the server; the server then returns the user authentication token $\sigma_{\mathsf{idx}(u)}$ along with $\mathtt{gmsg}$.

6.1 Authentication in a Static Group

Assume the base case where the user states of the GAM protocol are already set up. Recall an application message of MLS' FSPD protocol have the following format [12, Sec. 6]:

$$(\mathsf{gid}, \mathsf{epoch}_\mathsf{CGKA}, \mathsf{ct}_\mathsf{senderID}, \mathsf{ct}_\mathsf{Contents}, \mathsf{AuthData}) \quad (1)$$

gid is the group identity; $\mathsf{epoch}_\mathsf{CGKA}$ is a counter[11]; $\mathsf{senderID}$ is the sender's identity; $\mathsf{Contents}$ stores the payload; ct_X is an SKE encryption of X under an SKE key derived from the group secret key; and $\mathsf{AuthData}$ is an authentication data field, including an encryption of the signature (i.e., Enc-Sign mode). Below, we denote $\mathtt{gmsg}$ as Eq. (1), excluding $\mathsf{AuthData}$.

With this structure in mind, using a GAM protocol on top of MLS' FSPD is straightforward. This is depicted in Fig. 8 (see also Fig. 1). To send a message $\mathtt{gmsg}$, user $u \in \mathsf{G}$ runs the Send algorithm to create a group authentication token Σ_G and uploads $(\mathtt{gmsg}, \mathsf{AuthData} := \Sigma_\mathsf{G})$ to the server. The server verifies Σ_G and prepares *user* authentication tokens $(\sigma_i)_{i \in [N]}$, where $N = |\mathsf{G}|$. When a user $v \in \mathsf{G}$ with index $i = \mathsf{idx}(v)$ contacts the server, the server returns $(\mathtt{gmsg}, \mathsf{AuthData}_i := \sigma_i)$. User v can then verify and trace user u by running the $\mathsf{Receive}$ algorithm of the GAM protocol on $(\mathtt{gmsg}, \sigma_i)$.

Lastly, in case the GAM protocol requires state updates (e.g., `QUASAR` and `STARS`), this can be simply run on top of FSPD by directly embedding the group update information $\widehat{\Sigma}_\mathsf{G}$ into $\mathsf{Contents}$ or $\mathsf{ct}_\mathsf{Contents}$ depending on the anonymity guarantee we require.

---

[11] Note that this $\mathsf{epoch}_\mathsf{CGKA}$ is maintained by the CGKA protocol and differs from those used by `QUASAR` and `STARS`.

6.2 Authentication in a Dynamic Group

We now discuss how to add users to a preexisting GAM protocol using the welcome message functionality provided by MLS (or the CGKA protocol to be precise). Similarly to how the CGKA protocol is implemented, this procedure can be used to execute the Init algorithm. Since the process is inherently protocol specific, we explain them individually below.

COSMOS. This is simple since each user state is independent of the others. When an outsider joins the group via a welcome message, it mints a new token (x_0, y_0) and uploads the public token y_0 to the server and group. The outsider can additionally authenticate y_0 by including it in its key package maintained by the Delivery Service [20, Sec. 5] or by directly signing it to the group with its long-term key maintained by the Authentication Service [20, Sec. 4]. Once y_0 is shared among the group, it can use x_0 to start sending messages. Moreover, the outsider can fetch the current public tokens of the other users from the server. In case they want to be sure that these public tokens were indeed minted by the respective users, they can obtain the tokens directly from the senders. Note that the process is very similar to the Enc-Sign mode currently used by MLS' FSPD protocol. The case for the optimized COSMOS is similar.

COSMAC. This is almost identical to COSMOS. The only difference is that the outsider, after processing the welcome message, recovers the current group secret key gsk. It then uses the gsk to derive the MAC key $\mathsf{k}_{\mathsf{MAC}}$ and an SKE key $\mathsf{k}_{\mathsf{SKE}}$. Note that we can update $\mathsf{k}_{\mathsf{MAC}}$ and $\mathsf{k}_{\mathsf{SKE}}$ anytime the CGKA protocol performs a commit, which would allow some form of post-compromise security of the anonymity and anonymous blocklisting properties (see Sec. 8). Moreover, if we need sender anonymity of the welcome message, we can rely on existing anonymous *two-party* messaging protocols, such as Sealed Sender [60] or Orca [75], where the latter protocol also provides user traceability.

QUASAR *and* STARS. These are the most involved due to global state updates. Since the protocol flows of QUASAR and STARS are identical, we only focus on QUASAR. Looking at the toy example from Fig. 7, we cannot simply append columns corresponding to the new users since the server can always link public tokens associated to these appended columns. Thus, to ensure anonymity, the group users must all update their state and refresh the public tokens held by the server. In more detail and similarly to COSMOS, the outsider o, after having processed the welcome message, mints their one-time tokens, uploads the public tokens $\left(y^{(t)}_{j\to o}\right)_{(j,t)\in[N]\times[T]}$ to the server, and sends the PRF seeds $\mathsf{seed}_{j\to o}$ to each group user so that they can recover the corresponding private tokens. After every other user updates their states by further minting T extra public tokens for the new user o, the user can start sending messages.

An optimization of the above approach will have each group member j send a new PRF seed $\mathsf{seed}_{o\to j}$ to o, but, for each other group member i, derive the corresponding seed for the new epoch from the seed $\mathsf{seed}_{j\to i}$ (respectively $\mathsf{seed}_{i\to j}$) for the previous epoch, and upload the corresponding public tokens $y^{(t)}_{*\to j}$ to the server. This could be done by setting the new seed to be $\mathsf{OWF}(\mathsf{seed}_{j\to i}\|\mathsf{epoch}\|o)$ (respectively $\mathsf{OWF}(\mathsf{seed}_{i\to j}\|\mathsf{epoch}\|o)$). The advantage of such an approach is that each group member is only required to send a single ciphertext, and download nothing, as opposed to uploading and downloading N ciphertexts.

GEMSTARS. This is the only protocol that requires the MLS server to additionally run a group signature scheme. When a user joins the secure group messaging application, the server provides the user with a certificate (see Sec. 5.2 for details). Assuming this step has been finished, then joining a group is straightforward. This is because the only thing the outsider requires to run GEMSTARS is the groups tracer key and signature key, which are both derived from the group secret key gsk. Hence, following the same discussion in COSMAC, we can easily add new users to GEMSTARS.

Lastly, we note that removing users is straightforward for all of our GAM protocols. Since COSMOS is not anonymous, the server can simply maintain a list of group members. COSMAC and GEMSTARS natively supports removal at the CGKA layer since, once a commit occurs, the MAC key $\mathsf{k}_{\mathsf{MAC}}$ is updated along with the group secret key gsk. Finally, for QUASAR, the users can simply remove the unused public tokens corresponding to the removed users from the server.

7 Bandwidth Efficiency Analysis

In this section we analyze the efficiency of our proposed GAM protocols and compare them with existing authentication modes. We are specifically interested in the bandwidth overhead incurred by each authentication mode compared to a messaging protocol where the application message (e.g., chat texts) is sent in the clear, i.e., without authentication.

7.1 Instantiation

We target the NIST Level I security[12] stating that breaking the protocol is no easier than key-recovery on a block cipher with a 128-bit key (e.g., AES-128). This provides a meaningful baseline to discuss post-quantum security and ignoring quantum attacks corresponds to a classical security level of 128 bits. The main cryptographic primitives used in our GAM protocols is summarized below. The sizes of the cryptographic artifacts used in our instantiations are shown in Tab. 2.

OWF: We use SHA-256 and truncate its output to 16 B.

---

[12] https://csrc.nist.gov/projects/post-quantum-cryptography/post-quantum-cryptography-standardization/evaluation-criteria/security-(evaluation-criteria)

Table 2: Instantiation of the building blocks for our GAM protocols. SIG and GS stand for signature schemes and group signatures. sig, osig, and gsig denote the signature a standard signature, an OTS, and a group signature, respectively. ovk denotes the verification key of an OTS. ek and ct denote a KEM encapsulation key and ciphertext, respectively. All sizes are given in bytes. † indicates the output of SHA-256 is truncated to 16 B. We use κ to denote the security parameter, set to 128 bits.

Primitives	Classic		Post-quantum		Related Auth. Modes
OWF	SHA-256:	$\kappa = 16^{\dagger}$	SHA-256:	$\kappa = 16^{\dagger}$	COSMOS, COSMAC, QUASAR
MAC	HMAC-SHA-256:	$\kappa = 16^{\dagger}$	HMAC-SHA-256:	$\kappa = 16^{\dagger}$	COSMAC
OTS	EdDSA:	$\lvert \mathsf{ovk} \rvert = 32$, $\lvert \mathsf{osig} \rvert = 64$	WOTS^{+}:	$\lvert \mathsf{ovk} \rvert = 1320$, $\lvert \mathsf{osig} \rvert = 1072$	STARS
SIG	EdDSA:	$\lvert \mathsf{sig} \rvert = 64$	Dilithium:	$\lvert \mathsf{sig} \rvert = 2420$	Sign, Enc-Sign, Sign-Enc-Sign, GEMSTARS
GS	BBS:	$\lvert \mathsf{gsig} \rvert = 336$	LNP:	$\lvert \mathsf{gsig} \rvert = 9.2 \times 10^{4}$	GEMSTARS
KEM	Hashed ElGamal:	$\lvert \mathsf{ek} \rvert = \lvert \mathsf{ct} \rvert = 32$	Kyber:	$\lvert \mathsf{ek} \rvert = 800$, $\lvert \mathsf{ct} \rvert = 768$	QUASAR, STARS

MAC and PRF: We use HMAC [58] with SHA-256 and truncate its output to 16 B. Note that a deterministic MAC can be viewed as a PRF.

Pseudo-random permutation: We require a PRP to permute the set of NT tokens. We have the option of using either FastPRP [73] or the Thorp shuffle [67].

One-time signature schemes: For classical security, we use EdDSA [17]. For post-quantum security, we use WOTS^{+} [54, 55] used as a building block of the NIST PQC standard signature SPHINCS+ [56]. We set the Winternitz parameter $w = 16$, and use SHA-256 as the underlying hash function.

Signature schemes: For classical security, we use EdDSA [17]. For post-quantum security, we use the NIST PQC standard Dilithium [61]. We did not consider Falcon [70], another PQC standard, since Dilithium is selected as the primary algorithm and NIST recommends it for most use cases.

Group signatures: For classical security, we use the pairing-based BBS scheme [26] with the BLS12-381 pairing-friendly curve. For post-quantum security, we use the lattice-based scheme proposed by Lyubashevsky, Nguyen, and Plançon (LNP) [62].

KEM schemes: For classical and post-quantum security, we use the Hashed ElGamal KEM [40] and the NIST PQC standard Kyber [72], respectively.

7.2 Efficiency

Cost Metric. Following [7, 53], analyzing the bandwidth efficiency of MLS (and its variants), we analyze our GAM protocol through three metrics: the upload and download cost, and the total cost. Each of these costs are further broken down into offline and online costs; online (resp. offline) cost is associated to the cost of uploading and downloading contents generated by Send and Verify (resp. UpdSend and UpdVerify). We assume a user can send at most T messages once their states are updated. For protocols that require no updates, we can simply set $T = 1$ as there are no offline costs. In more detail, we have the following, where the costs are defined per user in a group of size N.

Total upload cost: The cost of uploading T outputs of Send (online) and one output of UpdSend (offline).

Total download cost: The cost of downloading NT outputs of Send (online) and N outputs of UpdSend (offline).

Total cost: The sum of the upload and download costs.

The download cost is a factor N times larger than the upload cost since each user has $N = \lvert G \rvert$ users to download from. Here, for COSMOS^{++}[13], COSMAC^{++}, and GEMSTARS, we can slightly optimize the download cost by allowing the users to not download the messages they upload. In contrast, for QUASAR and STARS, the users *must* download what they uploaded to preserve anonymity. At a high level, looking at Fig. 7, if a user does not download what it uploaded, then the server can link the (permuted) columns together. Lastly, observe that even if the offline cost is larger compared to the online cost, it gets amortized by T: as T grows larger, the total online cost starts to dominate the total offline cost.

Comparison of Communication Costs. We analyze the communication cost of our proposed GAM protocols and compare them with existing authentication modes. The number of exchanged cryptographic elements in the GAM protocols is summarized in Tab. 3. We classify the GAM protocols into the following three categories and compare them.

1. Non-anonymous protocols: Sign mode and COSMOS^{++}.
2. Anonymous protocols without tracing soundness: Enc-Sign mode, Sign-Enc-Sign mode, and COSMAC^{++}.
3. Anonymous protocols with tracing soundness: QUASAR, STARS, and GEMSTARS.

Table 3: The number of total cryptographic elements exchanged in GAM protocols. N is the group size and T is the number of online messages per one offline update. κ denotes the security parameter. sig, osig, and gsig denote a standard signature, a one-time signature, and a group signature, respectively. ovk denotes the verification key of a one-time signature. ct denotes a KEM ciphertext.

Auth. Mode	Offline						Online							
	Upload			Download			Upload				Download			
	κ	$\|ovk\|$	$\|ct\|$	κ	$\|ovk\|$	$\|ct\|$	κ	$\|osig\|$	$\|sig\|$	$\|gsig\|$	κ	$\|osig\|$	$\|sig\|$	$\|gsig\|$
Enc-Sign (MLS)									T				$(N-1)T$	
Sign-Enc-Sign [53]									T				$(N-1)T$	
`COSMOS`$^{++}$	3			$3(N-1)$			$2T$				$2(N-1)T$			
`COSMAC`$^{++}$	4			$4(N-1)$			$3T$				$3(N-1)T$			
`QUASAR`	$2NT$		N	$2N$		N	$2NT$				$2NT$			
`STARS`		NT	N	N		N		NT				NT		
`GEMSTARS`									T	T			$(N-1)T$	$(N-1)T$

Table 4: Communication cost of each GAM protocols. The sizes are in bytes. N is the group size and T is the number of online messages per one offline update. In Tabs. 4a and 4b, we ignore the $O(N)$ cost of the offline phase for readability. The column "Total" is normalized by NT, denoting the total overhead cost per message. The column "PQ?" is ✗ (resp. ✓) for classical (resp. post-quantum) security. The mode "Sign" in Tab. 4a corresponds to the naïve approach of simply signing messages.

a: Non-anonymous GAM protocols

Auth. Mode	Online		Total	PQ?
	Upload	Download		
Sign	$64 \cdot T$	$64 \cdot (N-1)T$	64	✗
	$2420 \cdot T$	$2420 \cdot (N-1)T$	2420	✓
`COSMOS`$^{++}$	$32 \cdot T$	$32 \cdot (N-1)T$	$16 \cdot (2 + \frac{3}{T})$	✓

b: Anonymous GAM protocols without tracing soundness

Auth. Mode	Online		Total	PQ?
	Upload	Download		
Enc-Sign (MLS)	$64 \cdot T$	$64 \cdot (N-1)T$	64	✗
	$2420 \cdot T$	$2420 \cdot (N-1)T$	2420	✓
Sign-Enc-Sign [53]	$128 \cdot T$	$128 \cdot (N-1)T$	128	✗
	$4840 \cdot T$	$4840 \cdot (N-1)T$	4840	✓
`COSMAC`$^{++}$	$48 \cdot T$	$48 \cdot (N-1)T$	$16 \cdot (3 + \frac{4}{T})$	✓

c: Anonymous GAM protocols with tracing soundness

Auth. Mode	Offline		Online		Total	PQ?
	Upload	Download	Upload	Download		
`QUASAR`	$32 \cdot NT + 32 \cdot N$	$64 \cdot N$	$64 \cdot NT$	$32 \cdot NT$	$\left(96 + \frac{96}{T}\right) \cdot \frac{N+1}{N}$	✗
	$32 \cdot NT + 768 \cdot N$	$800 \cdot N$	$32 \cdot NT$	$32 \cdot NT$	$\left(96 + \frac{1568}{T}\right) \cdot \frac{N+1}{N}$	✓
`STARS`	$32 \cdot NT + 32 \cdot N$	$48 \cdot N$	$64 \cdot NT$	$64 \cdot NT$	$\left(160 + \frac{80}{T}\right) \cdot \frac{N+1}{N}$	✗
	$1320 \cdot NT + 768 \cdot N$	$784 \cdot N$	$1072 \cdot NT$	$1072 \cdot NT$	$\left(3464 + \frac{1552}{T}\right) \cdot \frac{N+1}{N}$	✓
`GEMSTARS`			$400 \cdot T$	$400 \cdot (N-1)T$	400	✗
			$9.4 \times 10^4 \cdot T$	$9.4 \times 10^4 \cdot (N-1)T$	9.4×10^4	✓

Category (1). Tab. 4a compares the Sign mode (cf. Footnote 3) used in MLS and `COSMOS`$^{++}$. Technically, the Sign mode is used exclusively by the CGKA protocol in MLS, and not used to authenticate the output of the FSPD protocol. Nonetheless, this mode was used to analyze MLS by Alwen et al. [5] and we view it as the vanilla non-anonymous GAM protocol. Considering that any cryptographic element added to satisfy authentication should be at least 128-bits, `COSMOS`$^{++}$ is near optimal. Compared to the Sign mode, the total post-quantum communication cost is a factor 75x smaller. On the other hand, Sign mode achieves *standard* unforgeability while `COSMOS`$^{++}$ does not. We believe `COSMOS`$^{++}$ offers a worthwhile tradeoff between efficiency and security.

Category (2). Tab. 4b compares the Enc-Sign mode used in MLS, Sign-Enc-Sign mode [53], and `COSMAC`$^{++}$. Recall Enc-Sign mode does not achieve anonymous blocklisting while Sign-Enc-Sign and `COSMAC`$^{++}$ do (see Tab. 1). Out of the three protocols, `COSMAC`$^{++}$ has the lowest communication cost. Compared to the Enc-Sign mode used in MLS, the total post-quantum communication cost is a factor 50x smaller. As with `COSMOS`$^{++}$, `COSMAC`$^{++}$ only achieves non-colluding unforgeability while Enc-Sign and Sign-Enc-Sign modes do.

Category (3). Tab. 4c compares `QUASAR`, `STARS`, and `GEMSTARS`. These are the only anonymous GAM protocols with tracing soundness. `QUASAR` and `STARS` have a variable total communication cost that becomes smaller as T (and N) increases. This is because the KEM ciphertext encrypting the

---

[13] We only consider the most efficient version of `COSMOS` and `COSMAC` here.

PRF seed, exchanged during the offline phase, can be used to mint T tokens. Specifically, the cost of sending a large ciphertext is amortized by the number of messages T sent in the online phase. By setting $T = 1000$, the cost of sending a KEM ciphertext relative to the total cost is only 2 B or less per message, even in the post-quantum setting.

Out of the three protocols, `QUASAR` provides the most total-cost-efficient protocol. In fact, `QUASAR` is even comparable to $\texttt{COSMAC}^{++}$ that has no tracing soundness, e.g., it is 106 B when $(N,T) = (10,1000)$. While larger than `QUASAR`, `STARS` also offers a relatively small total overhead, albeit more computationally expensive due to running an OTS. The benefit of using `STARS` over `QUASAR` is that it achieves *standard* unforgeability. Lastly, while both `QUASAR` and `STARS` have an $O(N)$ online upload cost (i.e., maximum bandwidth consumption) per message, the concrete cost is only 16 KB for `QUASAR` even for a relatively large group of $N = 1024$. `STARS` uploads 64 KB and 1 MB of data in the classical and post-quantum settings, respectively. Lastly, recall one of the weakness of `QUASAR` and `STARS` are that they are only *globally* state-updatable. `GEMSTARS` removes updates altogether, with the cost of a larger total communication overhead; in the post-quantum setting, it is 94 KB.

8 Open Problems and Future Work

Other than those discussed in Sec. 2.4, we consider the following as interesting future work.

FS and PCS. Both forward secrecy (FS) and post-compromise security (PCS) are standard security notions in secure messaging. A natural question is then, *given the compromise of (one or more) users states to the adversary, what is the effect of this on the unforgeability, anonymity, anonymous blocklisting, and user traceability of past and future messages?* This opens interesting directions, both towards formalizing these notions and towards constructing authentication modes satisfying them.

Regarding FS, we first note that unforgeability, anonymous blocklisting, and user traceability are not relevant, as in our setting these notions are only concerned with the moment messages are processed by users.[14] Anonymity, on the other hand, is more interesting: can a state compromise allow an adversary to de-anonymize past messages from a user? The answer for both MLS and our proposals is "Yes", at least in some cases. Indeed, messages in each MLS' FSPD instantiation share an epoch and are thus all signed with the same signing key, and their sender identity and signature are encrypted with the shared secret in the epoch. In the latter, either key material is static (like in `GEMSTARS`), or anonymity relies on the key material that only gets rotated between CGKA epochs, like the MAC key used in `COSMAC`, or the permutation key used in `QUASAR` or `STARS`. Thus, natural questions are: *how do we formalize "forward anonymity"?* and *can we design authentication modes that satisfy it?*

The matter regarding PCS for authentication is more involved, since all the security notions make sense in this setting, as indicated by the original work on PCS by Cohn-Gordon, Cremers, and Garratt [39]. For unforgeability, the work of Cremers, Hale, and Kohbrok [41] introduces the notion of PCS signatures, where key-pairs can be evolved to "heal" from a compromise. For anonymity, ideas from unlinkable sanitizable signatures [30,45] could be useful. We leave concrete construction of a PCS GAM protocol as an interesting problem.

Optimal Security with PQ Efficiency. We provide several GAM protocols with different efficiency and security profiles, some of which offering much better post-quantum efficiency compared to the GAM protocol used in MLS, albeit weakening unforgeability. So far, the only GAM protocol satisfying optimal security (i.e., standard unforgeability, anonymity, anonymous blocklisting, tracing soundness) with no global state updates is `GEMSTARS`. However, this comes at a great cost as post-quantum group signatures are much more costly than signatures. We view it as an interesting open problem to find a GAM protocol achieving all the desireable properties while retaining efficiency.

Acknowledgments

The authors thank the anonymous CCS and USENIX reviewers. Their reviews really helped us make our paper better. This research was partially supported by JST CREST JPMJCR22M1, Japan and funded by the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie Grant Agreement No.665385.

Ethical Considerations

Our work follows the ethical guidelines of the conference. It proposes improvements for a popular and widely studied IETF standard: MLS. Most of the improvements target efficiency, and the formalization of authenticity notions, while new, does not allow for the creation of non-previously-known primitives that could be seen as bringing controversial or negative consequences, i.e., anonymous (the only notion that could be seen as potentially having a negative outcome) messaging systems already exist and, in particular, anonymity on the FSPD layer is already an aim of MLS. All the other authenticity notions enabled by this work are hard to imagine to have a negative outcome if implemented in the real world. Finally, our work did not include any experiments with live systems and does not lead to negative results for already used or implemented systems.

---

[14] The concept of *forward-secret signatures* [14], motivated by the will that the compromise of the current secret key does not enable an adversary to forge signatures pertaining to the past, is thus not relevant.

Open Science Policy

We do not have any artifacts (e.g., datasets, scripts, binaries) related to this paper.

References

[1] Joël Alwen, Benedikt Auerbach, Miguel Cueto Noval, Karen Klein, Guillermo Pascual-Perez, and Krzysztof Pietrzak. DeCAF: Decentralizable continuous group key agreement with fast healing. Cryptology ePrint Archive, Report 2022/559, 2022. `https://eprint.iacr.org/2022/559`.

[2] Joël Alwen, Benedikt Auerbach, Miguel Cueto Noval, Karen Klein, Guillermo Pascual-Perez, Krzysztof Pietrzak, and Michael Walter. CoCoA: Concurrent continuous group key agreement. In Dunkelman and Dziembowski [44], pages 815–844.

[3] Joël Alwen, Sandro Coretti, and Yevgeniy Dodis. The double ratchet: Security notions, proofs, and modularization for the Signal protocol. In Yuval Ishai and Vincent Rijmen, editors, *EUROCRYPT 2019, Part I*, volume 11476 of *LNCS*, pages 129–158. Springer, Heidelberg, May 2019.

[4] Joël Alwen, Sandro Coretti, Yevgeniy Dodis, and Yiannis Tselekounis. Security analysis and improvements for the IETF MLS standard for group messaging. In Daniele Micciancio and Thomas Ristenpart, editors, *CRYPTO 2020, Part I*, volume 12170 of *LNCS*, pages 248–277. Springer, Heidelberg, August 2020.

[5] Joël Alwen, Sandro Coretti, Yevgeniy Dodis, and Yiannis Tselekounis. Modular design of secure group messaging protocols and the security of MLS. In Vigna and Shi [76], pages 1463–1483.

[6] Joël Alwen, Sandro Coretti, Daniel Jost, and Marta Mularczyk. Continuous group key agreement with active security. In Pass and Pietrzak [68], pages 261–290.

[7] Joël Alwen, Dominik Hartmann, Eike Kiltz, and Marta Mularczyk. Server-aided continuous group key agreement. In Yin et al. [81], pages 69–82.

[8] Joël Alwen, Daniel Jost, and Marta Mularczyk. On the insider security of MLS. In Dodis and Shrimpton [43], pages 34–68.

[9] Michael Anastos, Benedikt Auerbach, Mirza Ahad Baig, Miguel Cueto Noval, Matthew Kwan, Guillermo Pascual-Perez, and Krzysztof Pietrzak. The cost of maintaining keys in dynamic groups with applications to multicast encryption and group messaging. In Elette Boyle and Mohammad Mahmoody, editors, *Theory of Cryptography*, pages 413–443, Cham, 2025. Springer Nature Switzerland.

[10] Benedikt Auerbach, Miguel Cueto Noval, Guillermo Pascual-Perez, and Krzysztof Pietrzak. On the Cost of Post-compromise Security in Concurrent Continuous Group-Key Agreement. In Guy Rothblum and Hoeteck Wee, editors, *Theory of Cryptography*, pages 271–300, Cham, 2023. Springer Nature Switzerland.

[11] David Balbás, Daniel Collins, and Serge Vaudenay. Cryptographic administration for secure group messaging. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 1253–1270, Anaheim, CA, August 2023. USENIX Association.

[12] Richard Barnes, Benjamin Beurdouche, Raphael Robert, Jon Millican, Emad Omara, and Katriel Cohn-Gordon. The Messaging Layer Security (MLS) Protocol. RFC 9420, July 2023.

[13] Mihir Bellare, Daniele Micciancio, and Bogdan Warinschi. Foundations of group signatures: Formal definitions, simplified requirements, and a construction based on general assumptions. In Eli Biham, editor, *EUROCRYPT 2003*, volume 2656 of *LNCS*, pages 614–629. Springer, Heidelberg, May 2003.

[14] Mihir Bellare and Sara K. Miner. A forward-secure digital signature scheme. In Michael J. Wiener, editor, *CRYPTO'99*, volume 1666 of *LNCS*, pages 431–448. Springer, Heidelberg, August 1999.

[15] Mihir Bellare and Phillip Rogaway. Entity authentication and key distribution. In Douglas R. Stinson, editor, *CRYPTO'93*, volume 773 of *LNCS*, pages 232–249. Springer, Heidelberg, August 1994.

[16] Mihir Bellare, Haixia Shi, and Chong Zhang. Foundations of group signatures: The case of dynamic groups. In Alfred Menezes, editor, *CT-RSA 2005*, volume 3376 of *LNCS*, pages 136–153. Springer, Heidelberg, February 2005.

[17] Daniel J. Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, and Bo-Yin Yang. High-speed high-security signatures. In Bart Preneel and Tsuyoshi Takagi, editors, *CHES 2011*, volume 6917 of *LNCS*, pages 124–142. Springer, Heidelberg, September / October 2011.

[18] Ward Beullens, Samuel Dobson, Shuichi Katsumata, Yi-Fu Lai, and Federico Pintore. Group signatures and more from isogenies and lattices: Generic, simple, and efficient. In Dunkelman and Dziembowski [44], pages 95–126.

[19] Benjamin Beurdouche, Eric Rescorla, Emad Omara, Srinivas Inguva, and Alan Duric. The Messaging Layer Security (MLS) Architecture. Internet-Draft draft-ietf-mls-architecture-10, Internet Engineering Task Force, December 2022. Work in Progress.

[20] Benjamin Beurdouche, Eric Rescorla, Emad Omara, Srinivas Inguva, and Alan Duric. The Messaging Layer Security (MLS) Architecture. Internet-Draft draft-ietf-mls-architecture-15, Internet Engineering Task Force, August 2024. Work in Progress.

[21] Karthikeyan Bhargavan, Richard Barnes, and Eric Rescorla. TreeKEM: Asynchronous Decentralized Key Management for Large Dynamic Groups A protocol proposal for Messaging Layer Security (MLS). Research report, Inria Paris, May 2018.

[22] Karthikeyan Bhargavan, Benjamin Beurdouche, and Prasad Naldurg. Formal Models and Verified Protocols for Group Messaging: Attacks and Proofs for IETF MLS. Research report, Inria Paris, December 2019.

[23] Alexander Bienstock, Yevgeniy Dodis, Sanjam Garg, Garrison Grogan, Mohammad Hajiabadi, and Paul Rösler. On the worst-case inefficiency of CGKA. In Eike Kiltz and Vinod Vaikuntanathan, editors, *TCC 2022, Part II*, volume 13748 of *LNCS*, pages 213–243. Springer, Heidelberg, November 2022.

[24] Alexander Bienstock, Yevgeniy Dodis, and Paul Rösler. On the price of concurrency in group ratcheting protocols. In Pass and Pietrzak [68], pages 198–228.

[25] Alexander Bienstock, Jaiden Fairoze, Sanjam Garg, Pratyay Mukherjee, and Srinivasan Raghuraman. A more complete analysis of the Signal double ratchet algorithm. In Yevgeniy Dodis and Thomas Shrimpton, editors, *CRYPTO 2022, Part I*, volume 13507 of *LNCS*, pages 784–813. Springer, Heidelberg, August 2022.

[26] Dan Boneh, Xavier Boyen, and Hovav Shacham. Short group signatures. In Matthew Franklin, editor, *CRYPTO 2004*, volume 3152 of *LNCS*, pages 41–55. Springer, Heidelberg, August 2004.

[27] Jonathan Bootle, Andrea Cerulli, Pyrros Chaidos, Essam Ghadafi, and Jens Groth. Foundations of fully dynamic group signatures. In Mark Manulis, Ahmad-Reza Sadeghi, and Steve Schneider, editors, *ACNS 16*, volume 9696 of *LNCS*, pages 117–136. Springer, Heidelberg, June 2016.

[28] Jacqueline Brendel, Rune Fiedler, Felix Günther, Christian Janson, and Douglas Stebila. Post-quantum asynchronous deniable key exchange and the signal handshake. In Goichiro Hanaoka, Junji Shikata, and Yohei Watanabe, editors, *PKC 2022, Part II*, volume 13178 of *LNCS*, pages 3–34. Springer, Heidelberg, March 2022.

[29] Chris Brzuska, Eric Cornelissen, and Konrad Kohbrok. Security analysis of the MLS key derivation. In *2022 IEEE Symposium on Security and Privacy*, pages 2535–2553. IEEE Computer Society Press, May 2022.

[30] Christina Brzuska, Marc Fischlin, Anja Lehmann, and Dominique Schröder. Unlinkability of sanitizable signatures. In Phong Q. Nguyen and David Pointcheval, editors, *PKC 2010*, volume 6056 of *LNCS*, pages 444–461. Springer, Heidelberg, May 2010.

[31] Ran Canetti, Palak Jain, Marika Swanberg, and Mayank Varia. Universally composable end-to-end secure messaging. In Dodis and Shrimpton [43], pages 3–33.

[32] Ran Canetti and Hugo Krawczyk. Security analysis of IKE's signature-based key-exchange protocol. In Moti Yung, editor, *CRYPTO 2002*, volume 2442 of *LNCS*, pages 143–161. Springer, Heidelberg, August 2002. `https://eprint.iacr.org/2002/120/`.

[33] Melissa Chase, Sarah Meiklejohn, and Greg Zaverucha. Algebraic MACs and keyed-verification anonymous credentials. In Gail-Joon Ahn, Moti Yung, and Ninghui Li, editors, *ACM CCS 2014*, pages 1205–1216. ACM Press, November 2014.

[34] Melissa Chase, Trevor Perrin, and Greg Zaverucha. The signal private group system and anonymous credentials supporting efficient verifiable encryption. In Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna, editors, *ACM CCS 2020*, pages 1445–1459. ACM Press, November 2020.

[35] David Chaum and Eugène van Heyst. Group signatures. In Donald W. Davies, editor, *EUROCRYPT'91*, volume 547 of *LNCS*, pages 257–265. Springer, Heidelberg, April 1991.

[36] Katriel Cohn-Gordon, Cas Cremers, Benjamin Dowling, Luke Garratt, and Douglas Stebila. A formal security analysis of the signal messaging protocol. In *IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 451–466, 2017.

[37] Katriel Cohn-Gordon, Cas Cremers, Benjamin Dowling, Luke Garratt, and Douglas Stebila. A formal security analysis of the signal messaging protocol. *Journal of Cryptology*, 33(4):1914–1983, October 2020.

[38] Katriel Cohn-Gordon, Cas Cremers, Luke Garratt, Jon Millican, and Kevin Milner. On ends-to-ends encryption: Asynchronous group messaging with strong security guarantees. In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *ACM CCS 2018*, pages 1802–1819. ACM Press, October 2018.

[39] Katriel Cohn-Gordon, Cas J. F. Cremers, and Luke Garratt. On post-compromise security. In Michael Hicks and Boris Köpf, editors, *CSF 2016 Computer Security Foundations Symposium*, pages 164–178. IEEE Computer Society Press, 2016.

[40] Ronald Cramer and Victor Shoup. Design and analysis of practical public-key encryption schemes secure against adaptive chosen ciphertext attack. *SIAM Journal on Computing*, 33(1):167–226, 2003.

[41] Cas Cremers, Britta Hale, and Konrad Kohbrok. The complexities of healing in secure group messaging: Why cross-group effects matter. In Michael Bailey and Rachel Greenstadt, editors, *USENIX Security 2021*, pages 1847–1864. USENIX Association, August 2021.

[42] 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.

[43] Yevgeniy Dodis and Thomas Shrimpton, editors. *CRYPTO 2022, Part II*, volume 13508 of *LNCS*. Springer, Heidelberg, August 2022.

[44] Orr Dunkelman and Stefan Dziembowski, editors. *EUROCRYPT 2022, Part II*, volume 13276 of *LNCS*. Springer, Heidelberg, May / June 2022.

[45] Nils Fleischhacker, Johannes Krupp, Giulio Malavolta, Jonas Schneider, Dominique Schröder, and Mark Simkin. Efficient unlinkable sanitizable signatures from signatures with re-randomizable keys. In Chen-Mou Cheng, Kai-Min Chung, Giuseppe Persiano, and Bo-Yin Yang, editors, *PKC 2016, Part I*, volume 9614 of *LNCS*, pages 301–330. Springer, Heidelberg, March 2016.

[46] Signal Foundation. Signal protocol: Technical documentation. (Accessed on 04/25/2023).

[47] GreenNet. Understanding file sizes. https://www.greennet.org.uk/support/understanding-file-sizes. (Accessed on 04/25/2023).

[48] Neil M. Haller. The S/KEY one-time password system. In *Proceedings of the ISOC Symposium on Network and Distributed System Security*, February 1994.

[49] Neil M. Haller. The S/KEY One-Time Password System. RFC 1760, February 1995.

[50] Keitaro Hashimoto, Shuichi Katsumata, Kris Kwiatkowski, and Thomas Prest. An efficient and generic construction for signal's handshake (X3DH): Post-quantum, state leakage secure, and deniable. In Juan Garay, editor, *PKC 2021, Part II*, volume 12711 of *LNCS*, pages 410–440. Springer, Heidelberg, May 2021.

[51] Keitaro Hashimoto, Shuichi Katsumata, Kris Kwiatkowski, and Thomas Prest. An efficient and generic construction for signal's handshake (X3DH): Post-quantum, state leakage secure, and deniable. *Journal of Cryptology*, 35(3):17, July 2022.

[52] Keitaro Hashimoto, Shuichi Katsumata, Eamonn Postlethwaite, Thomas Prest, and Bas Westerbaan. A concrete treatment of efficient continuous group key agreement via multi-recipient PKEs. In Vigna and Shi [76], pages 1441–1462.

[53] Keitaro Hashimoto, Shuichi Katsumata, and Thomas Prest. How to hide MetaData in MLS-like secure group messaging: Simple, modular, and post-quantum. In Yin et al. [81], pages 1399–1412.

[54] Andreas Hülsing. W-OTS+ - shorter signatures for hash-based signature schemes. In Amr Youssef, Abderrahmane Nitaj, and Aboul Ella Hassanien, editors, *AFRICACRYPT 13*, volume 7918 of *LNCS*, pages 173–188. Springer, Heidelberg, June 2013.

[55] Andreas Hülsing. WOTS+ – shorter signatures for hash-based signature schemes. Cryptology ePrint Archive, Report 2017/965, 2017. https://eprint.iacr.org/2017/965.

[56] Andreas Hulsing, Daniel J. Bernstein, Christoph Dobraunig, Maria Eichlseder, Scott Fluhrer, Stefan-Lukas Gazdag, Panos Kampanakis, Stefan Kolbl, Tanja Lange, Martin M Lauridsen, Florian Mendel, Ruben Niederhagen, Christian Rechberger, Joost Rijneveld, Peter Schwabe, Jean-Philippe Aumasson, Bas Westerbaan, and Ward Beullens. SPHINCS+. Technical report, National Institute of Standards and Technology, 2022. available at https://csrc.nist.gov/Projects/post-quantum-cryptography/selected-algorithms-2022.

[57] Karen Klein, Guillermo Pascual-Perez, Michael Walter, Chethan Kamath, Margarita Capretto, Miguel Cueto, Ilia Markov, Michelle Yeo, Joël Alwen, and Krzysztof Pietrzak. Keep the dirt: Tainted TreeKEM, adaptively and actively secure continuous group key agreement. In *2021 IEEE Symposium on Security and Privacy*, pages 268–284. IEEE Computer Society Press, May 2021.

[58] Hugo Krawczyk, Mihir Bellare, and Ran Canetti. HMAC: Keyed-hashing for message authentication. IETF Internet Request for Comments 2104, February 1997.

[59] Julia Len, Esha Ghosh, Paul Grubbs, and Paul Rösler. Interoperability in end-to-end encrypted messaging. Cryptology ePrint Archive, Report 2023/386, 2023. https://eprint.iacr.org/2023/386.

[60] Joshua Lund. Technology preview: Sealed sender for signal. https://signal.org/blog/sealed-sender/, October 2018. (Accessed on 04/19/2023).

[61] Vadim Lyubashevsky, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Peter Schwabe, Gregor Seiler, Damien Stehlé, and Shi Bai. CRYSTALS-DILITHIUM. Technical report, National Institute of Standards and Technology, 2022. available at https://csrc.nist.gov/Projects/post-quantum-cryptography/selected-algorithms-2022.

[62] Vadim Lyubashevsky, Ngoc Khanh Nguyen, and Maxime Plançon. Lattice-based zero-knowledge proofs and applications: Shorter, simpler, and more general. In Dodis and Shrimpton [43], pages 71–101.

[63] Moxie Marlinspike and Trevor Perrin. The double ratchet algorithm, November 2016. https://signal.org/docs/specifications/doubleratchet/.

[64] Moxie Marlinspike and Trevor Perrin. The X3DH key agreement protocol, November 2016. https://signal.org/docs/specifications/x3dh/.

[65] Jaroslav Martan. Webex meetings security. Technical report, September 2021. https://www.cisco.com/c/dam/m/cs_cz/training-events/webinars/tech-club-webinars/webex-meetings-security.pdf.

[66] Ian Martiny, Gabriel Kaptchuk, Adam J. Aviv, Daniel S. Roche, and Eric Wustrow. Improving signal's sealed sender. In *NDSS 2021*. The Internet Society, February 2021.

[67] Ben Morris, Phillip Rogaway, and Till Stegers. Deterministic encryption with the Thorp shuffle. *Journal of Cryptology*, 31(2):521–536, April 2018.

[68] Rafael Pass and Krzysztof Pietrzak, editors. *TCC 2020, Part II*, volume 12551 of *LNCS*. Springer, Heidelberg, November 2020.

[69] Trevor Perrin. The noise protocol framework, 2018. (Accessed on 04/23/2023).

[70] Thomas Prest, Pierre-Alain Fouque, Jeffrey Hoffstein, Paul Kirchner, Vadim Lyubashevsky, Thomas Pornin, Thomas Ricosset, Gregor Seiler, William Whyte, and Zhenfei Zhang. FALCON. Technical report, National Institute of Standards and Technology, 2022. available at https://csrc.nist.gov/Projects/post-quantum-cryptography/selected-algorithms-2022.

[71] Guardian Project. Orbot: Proxy with tor. (Accessed on 04/19/2023).

[72] Peter Schwabe, Roberto Avanzi, Joppe Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M. Schanck, Gregor Seiler, Damien Stehlé, and Jintai Ding. CRYSTALS-KYBER. Technical report, National Institute of Standards and Technology, 2022. available at https://csrc.nist.gov/Projects/post-quantum-cryptography/selected-algorithms-2022.

[73] Emil Stefanov and Elaine Shi. FastPRP: Fast pseudo-random permutations for small domains. Cryptology ePrint Archive, Report 2012/254, 2012. https://eprint.iacr.org/2012/254.

[74] Roman Tobe. Ringcentral expands end-to-end encryption to phone and messaging. https://www.ringcentral.com/us/en/blog/ringcentral-update-e2ee-2022/, 2022. (Accessed on 04/25/2023).

[75] Nirvan Tyagi, Julia Len, Ian Miers, and Thomas Ristenpart. Orca: Blocklisting in sender-anonymous messaging. In Kevin R. B. Butler and Kurt Thomas, editors, *USENIX Security 2022*, pages 2299–2316. USENIX Association, August 2022.

[76] Giovanni Vigna and Elaine Shi, editors. *ACM CCS 2021*. ACM Press, November 2021.

[77] Théophile Wallez, Jonathan Protzenko, Benjamin Beurdouche, and Karthikeyan Bhargavan. TreeSync: Authenticated group management for messaging layer security. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 1217–1233, Anaheim, CA, August 2023. USENIX Association.

[78] Matthew Weidner. Group messaging for secure asynchronous collaboration. Mphil dissertation, University of Cambridge, Cambridge, UK, 2019.

[79] Matthew Weidner, Martin Kleppmann, Daniel Hugenroth, and Alastair R. Beresford. Key agreement for decentralized secure group messaging with strong security guarantees. In Vigna and Shi [76], pages 2024–2045.

[80] Shouhuai Xu and Moti Yung. Accountable ring signatures: A smart card approach. In *Smart Card Research and Advanced Applications VI*, pages 271–286. Springer, 2004.

[81] Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi, editors. *ACM CCS 2022*. ACM Press, November 2022.

A Other Related Work

Secure Two-Party Messaging. A well-known secure two-party messaging protocols is the Signal protocol, which comprises two sub-protocols: X3DH protocol [64] and Double Ratchet protocol [63]. The full security of Signal protocol was analyzed by Cohn-Gordon et al. [36, 37]. Subsequently, these two sub-protocols were studied separately. The X3DH protocol, which is used to establish the initial secret key required to start a secure conversation, was analyzed by [28, 50, 51]. With respect to the Double Ratchet protocol, Alwen et al. [3] abstracted it as Continuous Key Agreement (CKA) (CGKA in two-party setting) and studied the security of Signal's Double Ratchet protocol in this abstruction. Recently, Bienstock et al. [25] and Canetti et al. [31] concurrently formalized an ideal functionality for the Signal protocol and proved its security using the Universally Composable (UC) framework.

Secure Group Messaging. Signal supports a group of more than two users by sending the same message to the users on the group with each of two-user messaging channels (i.e., pair-wise channels). It becomes less efficient as the group grows. To address this limitation, the Messaging Layer Security (MLS) protocol was developed in an IETF working group and is ready to be published as an RFC. MLS consists of two sub-protocols: Continuous Group Key Agreement (CGKA) protocol and Forward Secure Payload Delivery (FSPD) protocol. The former establishes a group secret key and continuously updates it, while the latter delivers payloads (a.k.a. application messages [12]) securely using the group secret key.

MLS uses TreeKEM [21] as a CGKA protocol, which is based on the Asynchronous Ratcheting Trees proposed by Cohn-Gordon et al. [38]. The security of TreeKEM as a standalone protocol was analyzed with game-based [4], simulation-based (UC framework) [8], and machine-checkable [22] security definitions. Alwen et al. [5] proposed a game-based security definitions for MLS, and analyzed MLS in its entirety. Their paper is the first to formally cast group messaging as the composition of CGKA and FSPD (termed FS-GAEAD in their model). They also identify a third component primitive, which takes the form of a hash function with particular properties. Bienstock et al. [23] proved a lower bound on the communication cost of any group messaging protocol, as well showing no optimal (with respect to distributions of group operations) protocol exists. Anastos et al. [9] prove a lower bound on the communication cost of replacing a set of users in CGKA, in particular showing MLS is optimal in this regard among protocols built in a black-box way from a standard set of primitives.

In addition to studying the standard TreeKEM in MLS, constructing new CGKA protocols is an active research topic. Re-randomized TreeKEM [4] and TreeKEM with active security [6] was proposed to improve forward security. Tainted TreeKEM [57] introduced an alternative approach to removing users which can be more efficient in certain settings, such as those with group administrators. Hashimoto et al. [52] and Alwen et al. [7] constructed communication-efficient CGKA protocols by allowing receivers to selectively download uploaded contents. Hashimoto et al. [52] proposed a CGKA protocol based on multi-recipient PKEs called Chained CmPKE, which is designed to reduce the total communication cost. Alwen et al. [7] proposed a server-aided CGKA protocol by extending MLS's TreeKEM, which is designed to reduce the upload and download costs per message. Weidner et al. [79] proposed a decentralized CGKA (DCGKA) protocol to realize secure group messaging for decentralized networks that have no central authority. An issue with CGKA protocols is how to handle concurrently uploaded key updates. To address this, CGKA protocols supporting concurrent key updates and exhibiting different trade-offs have been proposed [1, 2, 24, 78]. Bienstock et al. [24] also showed a lower bound on the communication costs of achieving PCS in two rounds of concurrent communication. This lower bound was extended by Auerbach et al. [10] to capture the more general setting of achieving PCS in a (potentially) higher number of rounds.

Authentication for Group States. In secure group messaging, group members share a common *group state* that includes information such as the list of group members and their cryptographic keys. When new members join the group, they must verify that the received group state is synchronized across all current members. If not, the group suffers from insider and outsider attacks e.g., double join attacks [22]. Signal uses an anonymous group management protocol called *private groups* [34]. In this protocol, the group membership list is encrypted with a group secret key and stored on the server. When members want to modify the list, the server anonymously authenticates them to ensure that they have the permission to do so. Further, a recent paper [11] introduces the notion of *administrated CGKA*, which enables a set of administrators to cryptographically manage group membership.

In MLS, new members receive the group state as a welcome message from an existing member. The group state includes the membership of the group and the structure of the public key tree for TreeKEM. MLS uses an authentication mechanism called *tree-signing* to allow new members to agree on and authenticate the group's membership and public key tree structure, without which powerful insider attacks could be performed [8]. Wallez et al. [77] formalized this authentication mechanism as *TreeSync* and provided a machine-checked formal specification for it.

It is worth noting that these works are orthogonal to the authentication we consider. Whereas they consider the integrity of *group states* and the authentication of values therein, we consider the authenticity of *conversation messages*.

Anonymous Secure Messaging. In the two-party setting, Signal provides sender-anonymous messaging through the Sealed Sender protocol [60]. Each user distributes an *access key* to their friends and registers it with the server in advance.

The server authenticates senders by checking whether the sender holds the access key of the intended receiver. Martiny et al. [66] demonstrated statistical analysis attacks that break sender anonymity in Sealed Sender, and suggested countermeasures to address their attacks. Tyagi et al. [75] pointed out that the access key distribution during the setup in Sealed Sender is not anonymous, and users could suffer from a DoS attack due to the lack of user traceability. To address these issues, they proposed a new sender-anonymous messaging protocol called Orca. Their protocol ensures that all communication, even in the setup phase, is anonymous and provides user traceability.

In the group setting, Signal already supports anonymous group messaging by using the sender-anonymous two-party protocol mentioned above as the underlying two-party messaging protocol. MLS also considers sender-anonymous group messaging according to its specification [12, Sec. 6.3.2.] (see also [19, Sec. 7.1.2]). Hashimoto et al. [53] studied the anonymity that MLS provides and defined a new security models that covers anonymity of senders and receivers against the server. They proposed a *metadata-hiding* CGKA protocol satisfying their security definition. Their protocol offers an implicit anonymous blocklisting property that allows the server to block access from outside the group, similar to Signal's Sealed Sender.

Group Signatures. Group signatures [13, 27, 35] can be used to achieve anonymous authentication with user traceability. They allow users to anonymously sign messages on behalf of a group, while allowing a special entity called the *group tracer* to trace the signature back to the user. In the context of secure messaging, we view the server as the group manager and verifier, and users as signers and group tracers. The server manages a group of all users in the system. Users join this group by setting up the account, and receiving a certificate from the server. With this set up in place, a sender can generate a signature with its certificate, and designate the intended receiver as tracer. The server can then verify the signature and, if verification is successful, deliver it to the intended receiver, who can then trace it back to the sender with its tracing key. In the context of group messaging, the set of users in a group is regarded as one instantiation of the group tracer, thus allowing all members to trace the sender. Tyagi et al. [75] and this work propose anonymous (group) messaging protocols following this framework.

To use group signatures in secure messaging as explained above, they must support dynamic groups [16] and multiple group tracers [75]. It is worth noting that the same key is used for both issuing certificates and verifying signature in this setting, since the server acts as both group manager and verifier. This property is called *keyed-verification* [33, 34, 75] in the literature, and it could be useful to construct efficient schemes. Tyagi et al. [75] constructed an efficient group signature satisfying these properties based on the Diffie-Hellman assumption.

B Preliminary: Cryptographic Tools

In this section, we define all the standard cryptographic tools used in our work.

One-Way Function. Let $\mathsf{OWF} : \mathcal{D} \rightarrow \mathcal{R}$ be an efficient function family with domain $\mathcal{D}$ and finite range $\mathcal{R}$. We define a one-way function as follows.

Definition B.1 (One-Way Function). *We say* OWF *is a* one-way function *if for all PPT adversary* $\mathcal{A}$*, we have*

$$\left|\Pr[x \leftarrow\!\$\, \mathcal{D}, x' \leftarrow\!\$\, \mathcal{A}(1^{\kappa}, \mathsf{OWF}(x)) : x = x']\right| \leq \mathsf{negl}(\kappa).$$

Lemma B.2. *Let* n, m *be integers such that* $n, m \geq \kappa$*. Then, a hash function* $\mathsf{H} : \{0,1\}^n \rightarrow \{0,1\}^m$ *instantiated as a random oracle is a* OWF*.*

Proof. Let $\mathcal{A}$ be a PPT adversary that makes at most $Q = \mathsf{poly}(\kappa)$ queries to the random oracle. First, the probability that $\mathcal{A}$ queries x is bounded by $Q/2^n$. Moreover, if it queries $z \neq x$, then the probability that $\mathsf{H}(z) = \mathsf{H}(x)$ is $Q/2^m$. Hence, the probability that $\mathcal{A}$ wins is upper bounded by $Q(1/2^n + 1/2^m) \leq Q/2^{\kappa-1} = \mathsf{negl}(\kappa)$. ☐

Pseudorandom Function. Let $\mathsf{PRF} : \mathcal{K} \times \mathcal{D} \rightarrow \mathcal{R}$ be an efficient function family with key space $\mathcal{K}$, domain $\mathcal{D}$ and finite range $\mathcal{R}$. We define a pseudorandom function as follows.

Definition B.3 (Pseudorandom Function). *We say* PRF *is a pseudo-random function if for all* PPT *adversary* $\mathcal{A}$*, we have*

$$\left|\Pr\left[\ b = b' \ :\ \begin{array}{c} b \leftarrow\!\$\, \{0,1\}; \mathsf{K} \leftarrow\!\$\, \mathcal{K}; \mathsf{RF} \leftarrow\!\$\, \mathcal{RF}; \\ b' \leftarrow\!\$\, \mathcal{A}^{\mathcal{F}(\cdot)}(1^{\kappa}) \end{array}\right] - \frac{1}{2}\right| \leq \mathsf{negl}(\kappa),$$

where $\mathcal{RF}$ *is a set of all functions with domain* $\mathcal{D}$ *and range* $\mathcal{R}$*, and* $\mathcal{F}(\cdot)$ *is defined as* $\mathsf{PRF}(\mathsf{K}, \cdot)$ *if* $b = 0$*, and* $\mathsf{RF}(\cdot)$ *otherwise.*

Pseudorandom Permutation. Let $\mathsf{PRP} : \mathcal{K} \times \mathcal{R} \rightarrow \mathcal{R}$ be an efficient function family of one-to-one functions from $\mathcal{R}$ to $\mathcal{R}$ with key space $\mathcal{K}$ for which PRP^{-1} is also efficiently computable given the first input (i.e., key). We define a pseudorandom permutation as follows.

Definition B.4 (Pseudorandom Permutation). *We say* PRP *is a pseudo-random permutation if for all* PPT *adversary* $\mathcal{A}$*, we have*

$$\left|\Pr\left[\ b = b' \ :\ \begin{array}{c} b \leftarrow\!\$\, \{0,1\}; \mathsf{K} \leftarrow\!\$\, \mathcal{K}; \mathsf{RP} \leftarrow\!\$\, \mathcal{RP}; \\ b' \leftarrow \mathcal{A}^{\mathcal{P}(\cdot), \mathcal{P}^{-1}(\cdot)}(1^{\kappa}) \end{array}\right] - \frac{1}{2}\right| \leq \mathsf{negl}(\kappa),$$

where $\mathcal{RP}$ *is the set of all permutations over* $\mathcal{R}$*, and* $\mathcal{P}(\cdot)$ *(resp.* $\mathcal{P}^{-1}(\cdot)$*) is defined as* $\mathsf{PRP}(\mathsf{K}, \cdot)$ *(resp.* $\mathsf{PRP}^{-1}(\mathsf{K}, \cdot)$*) if* $b = 0$*, and* $\mathsf{RP}(\cdot)$ *(resp.* $\mathsf{RP}^{-1}(\cdot)$*) otherwise.*

Secret Key Encryption. We provide the standard notion of secret key encryption (SKE).

Definition B.5 (Secret-Key Encryption). *A secret-key encryption (*SKE*) over key space* $\mathcal{K}$ *and message space* $\mathcal{M}$ *consists of the following two algorithms:*

$\mathsf{Enc}(\mathsf{k},\mathsf{m}) \to \mathsf{ct}$: *On input a secret key* $\mathsf{k} \in \mathcal{K}$ *and a message* $\mathsf{m} \in \mathcal{M}$*, it outputs a ciphertext* ct.

$\mathsf{Dec}(\mathsf{k},\mathsf{ct}) \to \mathsf{m}$ *or* $\perp$: *On input a secret key* k *and a ciphertext* ct*, it (deterministically) outputs either* $\mathsf{m} \in \mathcal{M}$ *or* $\perp \notin \mathcal{M}$.

Definition B.6 (Correctness). *An* SKE *is correct if* $\Pr[\mathsf{Dec}(\mathsf{k},\mathsf{Enc}(\mathsf{k},\mathsf{m})) = \mathsf{m}] = 1$ *holds for all* $\mathsf{m} \in \mathcal{M}$ *and* $\mathsf{k} \in \mathcal{K}$.

Definition B.7 ($\mathsf{IND\text{-}CCA}$**).** *An* SKE *is* $\mathsf{IND\text{-}CCA}$ secure *if for all PPT adversary* $\mathcal{A}$*, we have*

$$\left|\Pr\left[b=b' : \begin{array}{c}(b,\mathsf{k}) \leftarrow\$\, \{0,1\}\times\mathcal{K},\\ b' \leftarrow\$\, A^{\mathcal{C}(\cdot,\cdot),\mathcal{D}(\cdot)}(\mathsf{ct}^*)\end{array}\right]-\frac{1}{2}\right| \le \mathsf{negl}(\kappa),$$

where $\mathcal{C}(\mathsf{m}_0,\mathsf{m}_1)$ *outputs* $(\mathsf{ct}_0,\mathsf{ct}_1)$*, where* $\mathsf{ct}_i \leftarrow\$\,\mathsf{Enc}(\mathsf{k},\mathsf{m}_{i\oplus b})$ *for* $i \in \{0,1\}$*, and* $\mathcal{D}(\mathsf{ct})$ *returns* $\mathsf{Dec}(\mathsf{k},\mathsf{ct})$ *conditioned on* ct *not being an output of* $\mathcal{C}(\cdot,\cdot)$.

Message Authentication Code. We provide the standard notion of (deterministic) message authentication codes (MAC).

Definition B.8 (MAC**).** *A (deterministic) message authentication code* MAC *over key space* $\mathcal{K}$ *and message space* $\mathcal{M}$ *consists of the following algorithms:*

$\mathsf{TagGen}(\mathsf{k},\mathsf{m}) \to \mathsf{tag}$*: On input a key* $\mathsf{k} \in \mathcal{K}$ *and a message* $\mathsf{m} \in \mathcal{M}$*, it (deterministically) outputs a tag* tag.

$\mathsf{Verify}(\mathsf{k},\mathsf{tag},\mathsf{m}) \to \perp/\top$*: On input a key* k*, a tag* tag*, and a message* m*, it (deterministically) outputs* $\top$ *or* $\perp$.

Since the TagGen algorithm is deterministic, we can simply define Verify to run TagGen on (k,m) and check if the generated tag' is identical to the provided tag.

Definition B.9 (Correctness). *A* MAC *is correct if for all keys* $\mathsf{k} \in \mathcal{K}$ *and all messages* $\mathsf{m} \in \mathcal{M}$*,*

$$\Pr[\mathsf{Verify}(\mathsf{k},\mathsf{TagGen}(\mathsf{k},\mathsf{m}),\mathsf{m}) = \top] = 1.$$

Definition B.10 ($\mathsf{sEUF\text{-}CMA}$**).** *A* MAC *is* $\mathsf{sEUF\text{-}CMA}$ *secure if for all PPT adversary* $\mathcal{A}$*, we have*

$$\Pr\left[\begin{array}{l}\mathsf{Verify}(\mathsf{k},\mathsf{m}^*,\mathsf{tag}^*) = \top\\ \wedge(\mathsf{m}^*,\mathsf{tag}^*) \notin L^*\end{array} : \begin{array}{c}\mathsf{k} \leftarrow\$\,\mathcal{K};\\ (\mathsf{m}^*,\mathsf{tag}^*) \leftarrow\$\,\mathcal{A}^{\mathcal{T}(\cdot)}(1^\kappa)\end{array}\right] \le \mathsf{negl}(\kappa)$$

where $\mathcal{T}$ *is the MAC oracle which on input* m *returns* $\mathsf{TagGen}(\mathsf{k},\mathsf{m})$*, and* L^* *is the set of pairs of message and tag generated by the MAC oracle.*

(One Time) Signature We provide the standard notion of a signature scheme.

Definition B.11 (Signature Scheme). *A signature scheme* SIG *over a message space* $\mathcal{M}$ *consists of the following algorithms:*

$\mathsf{KeyGen}(1^\kappa) \to (\mathsf{vk},\mathsf{sk})$*: On input the security parameter* 1^κ*, it outputs a verification and signing key pair* $(\mathsf{vk},\mathsf{sk})$.

$\mathsf{Sign}(\mathsf{sk},\mathsf{m}) \to \sigma$*: On input a signing key* sk *and a message* $\mathsf{m} \in \mathcal{M}$*, it outputs a signature* σ.

$\mathsf{Verify}(\mathsf{vk},\sigma,\mathsf{m}) \to \perp/\top$*: On input a key* k*, a signature* σ*, and a message* m*, it (deterministically) outputs* $\top$ *or* $\perp$.

Definition B.12 (Correctness). *A signature scheme* SIG *is correct if for all* $\kappa \in \mathbb{N}$*,* $\mathsf{m} \in \mathcal{M}$*, and* $(\mathsf{vk},\mathsf{sk}) \in \mathsf{KeyGen}(1^\kappa)$*, we have*

$$\Pr[\mathsf{Verify}(\mathsf{vk},\sigma,\mathsf{m}) = \top : \sigma \leftarrow\$\,\mathsf{Sign}(\mathsf{sk},\mathsf{m})] = 1 - \mathsf{negl}(\kappa).$$

Definition B.13 ($\mathsf{EUF\text{-}CMA}$**).** *A signature scheme* SIG *is* $\mathsf{EUF\text{-}CMA}$ *secure if for all PPT adversary* $\mathcal{A}$*, we have*

$$\Pr\left[\begin{array}{l}\mathsf{Verify}(\mathsf{vk},\sigma^*,\mathsf{m}^*) = \top\\ \wedge\mathsf{m}^* \notin L^*\end{array} : \begin{array}{c}(\mathsf{vk},\mathsf{sk}) \leftarrow\$\,\mathsf{KeyGen}(1^\kappa);\\ (\mathsf{m}^*,\sigma^*) \leftarrow\$\,\mathcal{A}^{\mathcal{S}(\cdot)}(1^\kappa)\end{array}\right] \le \mathsf{negl}(\kappa)$$

where $\mathcal{S}$ *is the signing oracle which on input* m *returns* $\sigma \leftarrow\$\,\mathsf{Sign}(\mathsf{sk},\mathsf{m})$*, and* L^* *is the set of messages queried to the signing oracle. We say* SIG *is* one-time *secure if* $|L^*| = 1$.

Key Encapsulation Mechanism We provide the standard notion of a key encapsulation mechanism (KEM).

Definition B.14 (Key Encapsulation Mechanism). *A key encapsulation mechanism (*KEM*) with key space* $\mathcal{K}$ *consists of the following PPT algorithms:*

$\mathsf{KeyGen}(1^\kappa) \to (\mathsf{ek},\mathsf{dk})$: *On input a security parameter* 1^κ*, it outputs a pair of encryption and decryption keys* $(\mathsf{ek},\mathsf{dk})$.

$\mathsf{Enc}(\mathsf{ek}) \to \mathsf{ct}$: *On input an encryption key* ek*, it outputs a key* $\mathsf{k} \in \mathcal{K}$ *and a ciphertext* ct.

$\mathsf{Dec}(\mathsf{dk},\mathsf{ct}) \to \mathsf{m}/\perp$: *On input a decryption key* dk *and a ciphertext* ct*, it outputs a key* $\mathsf{k} \in \mathcal{K} \cup \{\perp\}$.

Definition B.15 ($(1-\delta)$**-Correctness).** *A* KEM *is* $(1-\delta)$-correct *if for all* $\kappa \in \mathbb{N}$*, we have*

$$(1-\delta) \le \Pr\left[\begin{array}{c}(\mathsf{ek},\mathsf{dk}) \leftarrow\$\,\mathsf{KeyGen}(1^\kappa),\\ (\mathsf{k},\mathsf{ct}) \leftarrow\$\,\mathsf{Enc}(\mathsf{ek})\end{array} : \mathsf{Dec}(\mathsf{dk},\mathsf{ct}) = \mathsf{k}\right].$$

Definition B.16 ($\mathsf{IND\text{-}CCA}$ **Security).** *A* KEM *is* $\mathsf{IND\text{-}CCA}$ *secure if for all PPT adversary* $\mathcal{A}$ *we have*

$$\left|\Pr\left[b=b' : \begin{array}{c}(\mathsf{ek},\mathsf{dk}) \leftarrow\$\,\mathsf{KeyGen}(1^\kappa),\\ (b,\mathsf{k}_0^*) \leftarrow\$\,\{0,1\}\times\mathcal{K},\\ (\mathsf{k}_1^*,\mathsf{ct}^*) \leftarrow\$\,\mathsf{Enc}(\mathsf{ek}),\\ b' \leftarrow\$\,A^{\mathcal{D}(\cdot)}(\mathsf{ek},\mathsf{ct}^*,k_b)\end{array}\right]-\frac{1}{2}\right| \le \mathsf{negl}(\kappa),$$

where $\mathcal{D}(\mathsf{ct})$ *outputs* $\mathsf{Dec}(\mathsf{dk},\mathsf{ct})$ *conditioned on* $\mathsf{ct} \neq \mathsf{ct}^*$.

C Security Proofs for COSMOS and COSMAC

In this section, we provide the security proofs for COSMOS and COSMAC.

C.1 Security Proof of COSMOS

We prove that COSMOS is correct and secure. Here, note that COSMOS is not anonymous since the list of public tokens $\mathsf{PubTOKEN}$ is indexed by the users $u \in \mathsf{G}$ and when user u updates its tokens $(x_i, y_i)_{i \in [T]}$, the list of public tokens $\mathsf{PubTOKEN}[u]$ is updated to $(y_i)_{i \in [T]}$. Any two messages with private tokens x and x' such that $\mathsf{OWF}(x), \mathsf{OWF}(x') \in (y_i)_{i \in [T]}$ can thus be linked together. Also, recall that if a GAM protocol is not anonymous, then blocklisting becomes trivial (see Secs. 2.2.2 and 2.3).

Formally, we have the following theorem.

Theorem C.1. *The* GAM *protocol* COSMOS *in Fig. 3 is signing correct, local state update correct, non-colluding unforgeable, and tracing sound assuming* OWF *is one-way,* PRF *is pseudorandom, and* MAC *is* sEUF-CMA.

Proof. Correctness of signing and local state-updates is clear from construction. In the following, we prove COSMOS is non-colluding unforgeable and tracing sound in Lems. C.2 and C.3, respectively. ☐

C.1.1 Proof of Lem. C.2

Lemma C.2. COSMOS *is non-colluding unforgeable assuming* OWF *is one-way,* PRF *is pseudorandom, and* MAC *is* sEUF-CMA.

Proof. There are two cases we must consider for non-colluding unforgeability: $\mathcal{C} \neq \emptyset$ (i.e., the server is honest and $\mathcal{A}$ has access to $\mathcal{O}^\star$) and $\mathcal{C} = \emptyset$ (i.e., the server is malicious and $\mathcal{A}$ has access to $\mathcal{O}$).

We first consider the first case. Let us assume $\mathcal{A}$ outputs $(\mathtt{label}, \mathtt{obj}) = (\mathtt{msg}, (v, \sigma, \mathsf{m}))$ for $v \in \mathcal{H}$; the other case when $\mathtt{label} = \mathtt{upd}$ is proven identically. Since it is a valid adversary, the authentication token σ is valid and traces back to some honest user $u^* \in \mathcal{H}$. That is, σ is the form $(u^*, \mathsf{ctr}, x, \Sigma_{\mathsf{MAC}})$ and $(u^*, *, \mathsf{m}) \notin L_{\mathsf{msg}}$. Now notice that a group authentication token of the form $\Sigma_{\mathsf{G}} = (u^*, \mathsf{ctr}, x, *)$ could not have been output by $\mathcal{O}_{\mathsf{Send}}$ since when $\mathcal{A}$ has access to $\mathcal{O}^\star$, Σ_{G} is immediately processed by $\mathcal{O}_{\mathsf{GroupReceive}}$. In particular, the honest server and group users delete the public token $y = \mathsf{OWF}(x)$ from DB and will not verify the second time it is used. Thus, for $\mathcal{A}$ to forge u^* it must output an unknown private token x for a public token $y = \mathsf{OWF}(x)$ without querying $\mathcal{O}_{\mathsf{Send}}$ on u^*. Due to the one-wayness of OWF, this can be done with at most negligible probability.

We next consider the second case. Similarly as before, let us assume $\mathcal{A}$ outputs $(\mathtt{label}, \mathtt{obj}) = (\mathtt{msg}, (v, \sigma, \mathsf{m}))$ for $v \in \mathcal{H}$ and traces to some honest user $u^* \in \mathcal{H}$. Moreover, due to the winning condition, we have $\sigma = (u^*, \mathsf{ctr}, x, \Sigma_{\mathsf{MAC}})$ with $\mathsf{MAC.Verify}(\mathsf{k}_{\mathsf{MAC}}, (u^*, \mathsf{ctr}x, \mathsf{m}), \Sigma_{\mathsf{MAC}}) = \top$ and $(u^*, *, \mathsf{m}) \notin L_{\mathsf{msg}}$. However, since the malicious server does not know the MAC key $\mathsf{k}_{\mathsf{MAC}}$, this can occur with at most negligible probability assuming sEUF-CMA security of the MAC. Concretely, if $(u^*, *, \mathsf{m}) \notin L_{\mathsf{msg}}$, then the game will have never signed the tuple $(u^*, \mathsf{ctr}x, \mathsf{m})$. Therefore, if an adversary outputs a MAC tag Σ_{MAC} for such a tuple, it breaks sEUF-CMA security. Here, to simulate the sEUF-CMA security game, we rely on the pseudorandomness of the PRF to sample the MAC key $\mathsf{k}_{\mathsf{MAC}}$ directly without running PRF. ☐

It is worth mentioning that in the above proof, we crucially rely on the fact that either the set of group users or the server is honest. When considering *standard* unforgeability, a malicious user and server can easily collude to break unforgeability as secrecy of both the OWF inputs and MAC key $\mathsf{k}_{\mathsf{MAC}}$ are known to the adversary.

C.1.2 Proof of Lem. C.3

Lemma C.3. COSMOS *is (unconditionally) tracing sound.*

Proof. Assume the adversary $\mathcal{A}$ outputs $(\mathtt{label}, \mathtt{obj}) = (\mathtt{msg}, (\Sigma_{\mathsf{G}}, \mathsf{m}))$. The game then runs $(\mathsf{pp}', (\sigma_i)_{i \in [N]}) \leftarrow \mathsf{Verify}(\mathsf{pp}, \Sigma_{\mathsf{G}}, \mathsf{m})$. Since pp' verifies, $\Sigma_{\mathsf{G}} = (u, \mathsf{ctr}, x, \Sigma_{\mathsf{MAC}})$ for some $u \in \mathsf{G}$, $\mathsf{ctr} \in [T]$, and $\mathsf{DB}[u][\mathsf{ctr}] = \mathsf{OWF}(x)$, where DB is the database stored in the public parameter pp. Then, a user authentication token for user $v \in \mathsf{G} \cap \mathcal{H}$ is set as $\sigma_v = (u, \mathsf{ctr}, x, \Sigma_{\mathsf{MAC}})$. Next, notice that all $v \in \mathsf{G} \cap \mathcal{H}$ include the same PubTOKEN in their state as those included in DB. Then, by the description of `*trace-sender` in Fig. 4, all honest users v either all reject the authentication token because Σ_{MAC} is invalid or uniquely traces u given σ_v. The case when $\mathcal{A}$ outputs $(\mathtt{label}, \mathtt{obj}) = (\mathtt{upd}, (\widehat{\Sigma}_{\mathsf{G}}, \widehat{\mathsf{ct}}_{\mathsf{G}}))$ is proven identically. This completes the proof of the lemma. ☐

C.1.3 Security of the Optimizations

Lifting the security proofs of COSMOS from App. C to the optimized schemes $\mathtt{COSMOS}^{+}$ and $\mathtt{COSMOS}^{++}$ are straightforward. Indeed, it is clear from the similarity of the construction that $\mathtt{COSMOS}^{+}$ will satisfy the same security guarantees. When it comes to $\mathtt{COSMOS}^{++}$, we note that tracing soundness immediately follows from the original proof in Lemma C.3. The proof for non-colluding unforgeability in Lemma C.2 shows that, in order to forge, an adversary must either break sEUF-CMA security of the MAC or find a pre-image of the employed OWF. Relying on the same argument and noting that any revealed private token becomes a pre-image challenge for the adversary, it follows that $\mathtt{COSMOS}^{++}$ is non-colluding unforgeable as well.

C.2 Security Proof of `COSMAC`

We show in the following theorem that `COSMAC` is non-colluding unforgeable, anonymous, and anonymous blocklistable. We note that `COSMAC` is not tracing sound since any malicious user can upload a valid MAC tag along with a ciphertext that does not decrypt to any valid authentication tokens of `COSMOS`. The optimized schemes `COSMAC`$^+$ and `COSMAC`$^{++}$ can be proven almost identically as per the discussion in App. C.1.3.

Theorem C.4. *The* GAM *protocol* `COSMAC` *in Fig. 5 is signing correct, local state update correct, non-colluding unforgeable, anonymous, and anonymous blocklistable assuming* OWF *is one-way,* PRF *is pseudorandom,* MAC *is* sEUF-CMA *secure, and* SKE *is IND-CCA secure.*

Proof. Correctness of signing and local state-updates are inherited from `COSMOS`, additionally relying on the fact that the MAC and SKE are correct. In the following we prove `COSMAC` is non-colluding unforgeable, anonymous, and anonymous blocklistable in Lems. C.5 to C.7, respectively. Proving these lemmas completes the proof of the theorem. □

C.2.1 Proof of Lem. C.5

Lemma C.5. `COSMAC` *is non-colluding unforgeable assuming* OWF *is one-way,* PRF *is pseudorandom, and* MAC *is* sEUF-CMA.

Proof. An adversary $\mathcal{A}$ that breaks the non-colluding unforgeability of `COSMAC` can be used to construct an adversary $\mathcal{B}$ that breaks the non-colluding unforgeability of `COSMOS`. Concretely, $\mathcal{B}$ generates $\overline{\mathsf{k}_{\mathsf{MAC}}}$ and $\mathsf{k}_{\mathsf{SKE}}$ randomly over their respective domains without invoking the PRF; this is indistinguishable assuming the pseudorandomness of PRF. To simulate O_{Send} and O_{UpdSend} to $\mathcal{A}$, $\mathcal{B}$ calls its own oracle; encrypts its output using $\mathsf{k}_{\mathsf{SKE}}$; and signs it using $\overline{\mathsf{k}_{\mathsf{MAC}}}$ as specified by `COSMAC`. To simulate O_{Receive}, $O_{\mathsf{UpdReceive}}$, $O_{\mathsf{GroupReceive}}$, and $O_{\mathsf{GroupUpdReceive}}$ to $\mathcal{A}$, $\mathcal{B}$ checks if the attached tag and content verifies and decrypts under $\overline{\mathsf{k}_{\mathsf{MAC}}}$ and $\mathsf{k}_{\mathsf{SKE}}$, respectively. If so, it queries its own oracle on the decrypted message. Finally, if $\mathcal{A}$ outputs a valid forgery, $\mathcal{B}$ removes the tag, decrypts it using $\mathsf{k}_{\mathsf{SKE}}$, and submits it as its own forgery. Thus, assuming that `COSMOS` is non-colluding unforgeable (which is established in Lem. C.2), `COSMAC` is non-colluding unforgeable. □

C.2.2 Proof of Lem. C.6

Lemma C.6. `COSMAC` *is anonymous assuming* PRF *is pseudorandom and* SKE *is* IND-CCA *secure.*

Proof. We first use the pseudorandomness of the PRF to modify the security game so that $\mathsf{k}_{\mathsf{MAC}}$, $\overline{\mathsf{k}_{\mathsf{MAC}}}$, and $\mathsf{k}_{\mathsf{SKE}}$ are sampled independently. We show that an adversary $\mathcal{A}$ that breaks anonymity of this modified security game can be used to construct an adversary $\mathcal{B}$ that breaks IND-CCA security of the SKE.

Concretely, $\mathcal{B}$ samples $\mathsf{sk}_{\mathsf{Sv}} = \overline{\mathsf{k}_{\mathsf{MAC}}}$ and provides it to $\mathcal{A}$. It then prepares an empty list L_{ct}. When $\mathcal{A}$ queries O_{Send} and O_{UpdSend}, $\mathcal{B}$ runs `COSMAC` except that it queries $(\bar{\mathsf{m}}, \bar{\mathsf{m}})$ to its encryption oracle $\mathcal{C}(\cdot,\cdot)$ where $\bar{\mathsf{m}} = (u, \mathsf{ctr}, x_u^{(\mathsf{ctr})}, \Sigma_{\mathsf{MAC}})$, rather than generating it itself. On receiving $(\mathsf{ct}_{\mathsf{SKE},0}, \mathsf{ct}_{\mathsf{SKE},1})$ from the oracle, $\mathcal{B}$ uses $\mathsf{ct}_{\mathsf{SKE},0}$ and $\overline{\mathsf{k}_{\mathsf{MAC}}}$ to create the authentication tokens. It then updates $L_{\mathsf{ct}} \leftarrow L_{\mathsf{ct}} \cup (\mathsf{m}, \mathsf{ct}_{\mathsf{SKE},0})$, where m is the message for which `*attach-auth-token` is invoked when $\mathcal{A}$ queries O_{Send} and O_{UpdSend}. This simulates `COSMAC` perfectly.

When $\mathcal{A}$ makes a challenge query $(u_0, u_1, \mathsf{m}_0, \mathsf{m}_1)$, $\mathcal{B}$ runs `COSMAC` except that it queries $((u_0, \mathsf{ctr}_0, x_{u_0}^{(\mathsf{ctr}_0)}, \Sigma_{\mathsf{MAC}_0}), (u_1, \mathsf{ctr}_1, x_{u_1}^{(\mathsf{ctr}_1)}, \Sigma_{\mathsf{MAC},1}))$ to its encryption oracle $\mathcal{C}(\cdot,\cdot)$. Here, note that we ignore the public parameter $\overline{\mathsf{pp}}$ from $\mathcal{A}$'s output as it is empty for `COSMAC`. On receiving $(\mathsf{ct}^*_{\mathsf{SKE},0}, \mathsf{ct}^*_{\mathsf{SKE},1})$ from the oracle, $\mathcal{B}$ generates MAC tags $(\overline{\Sigma}^*_{\mathsf{MAC},0}, \overline{\Sigma}^*_{\mathsf{MAC},1})$ using $\overline{\mathsf{k}_{\mathsf{MAC}}}$, respectively, and then outputs $\Big(((\mathsf{ct}^*_{\mathsf{SKE},0}, \overline{\Sigma}^*_{\mathsf{MAC},0}), \mathsf{m}_0), ((\mathsf{ct}^*_{\mathsf{SKE},1}, \overline{\Sigma}^*_{\mathsf{MAC},1}), \mathsf{m}_1)\Big)$. Finally, $\mathcal{B}$ executes the Verify and Receive algorithms to check the resulting (group) authentication tokens are valid. Here, we note that $\mathcal{B}$ can ignore decrypting the ciphertext $(\mathsf{ct}^*_{\mathsf{SKE},0}, \mathsf{ct}^*_{\mathsf{SKE},1})$ in the Receive algorithm assuming correctness of the SKE. It can be checked that the challenge bit of the IND-CCA security game is consistent with the challenge bit coin implicitly sampled by $\mathcal{B}$. Hence, this simulates `COSMAC` perfectly.

Lastly, when $\mathcal{A}$ queries O_{Receive} and $O_{\mathsf{UpdReceive}}$ on an authentication token σ, $\mathcal{B}$ checks $\sigma = \mathsf{ct}'_{\mathsf{SKE}} \notin \mathsf{Chall}_{\mathsf{msg}}$, where recall $\mathsf{Chall}_{\mathsf{msg}}$ is the $2N$ authentication tokens generated by Verify above. If the ciphertext $\mathsf{ct}'_{\mathsf{SKE}}$ satisfies $(\mathsf{m}', \mathsf{ct}'_{\mathsf{SKE}}) \in L_{\mathsf{ct}}$ for some m', it runs `COSMAC` except that it uses m'. Otherwise, if $\mathsf{ct}'_{\mathsf{SKE}} \notin L_{\mathsf{ct}} \cup \{\mathsf{ct}^*_{\mathsf{SKE},0}, \mathsf{ct}^*_{\mathsf{SKE},1}\}$, it runs `COSMAC` except that it decrypts $\mathsf{ct}'_{\mathsf{SKE}}$ using its decryption oracle $\mathcal{D}(\cdot)$. Finally, we never have the case $\mathsf{ct}'_{\mathsf{SKE}} \in \{\mathsf{ct}^*_{\mathsf{SKE},0}, \mathsf{ct}^*_{\mathsf{SKE},1}\}$ due to the condition $\sigma \notin \mathsf{Chall}_{\mathsf{msg}}$. Thus, this simulates `COSMAC` perfectly.

At the end of the game, when $\mathcal{A}$ outputs a guess $\widehat{\mathsf{coin}}$, $\mathcal{B}$ outputs this as its guess. Since $\mathcal{B}$ perfectly simulates the game to $\mathcal{A}$, if $\mathcal{A}$ breaks anonymity of `COSMAC`, then $\mathcal{B}$ breaks IND-CCA security of SKE. This completes the proof. □

C.2.3 Proof of Lem. C.7

Lemma C.7. `COSMAC` *is anonymous blocklistable assuming* PRF *is pseudorandom and* MAC *is* sEUF-CMA *secure.*

Proof. We first use the pseudorandomness of the PRF to modify the security game so that $\mathsf{k}_{\mathsf{MAC}}$, $\overline{\mathsf{k}_{\mathsf{MAC}}}$, and $\mathsf{k}_{\mathsf{SKE}}$ are sampled independently. We show that an adversary $\mathcal{A}$ that breaks anonymous blocklisting of this modified security game can be used to construct an adversary $\mathcal{B}$ that breaks sEUF-CMA security of the MAC.

Concretely, $\mathcal{B}$ samples everything as in the anonymous blocklistable security game except for implicitly setting $\mathsf{sk}_{\mathsf{Sv}} = \overline{\mathsf{k}_{\mathsf{MAC}}}$ as the MAC key used by the sEUF-CMA security game. It then proceeds as follows. When $\mathcal{A}$ queries $\mathcal{O}_{\mathsf{Send}}$ and $\mathcal{O}_{\mathsf{UpdSend}}$, $\mathcal{B}$ runs `COSMAC` except that it queries the MAC oracle $\mathsf{TG}(\cdot)$ on $\mathsf{ct}_{\mathsf{SKE}}$, rather than generating the MAC tag on its own. This perfectly simulates the view to $\mathcal{A}$ since MAC is perfectly correct.

When $\mathcal{A}$ queries $\mathcal{O}_{\mathsf{GroupReceive}}$ and $\mathcal{O}_{\mathsf{GroupUpdReceive}}$, it queries its MAC oracle on $\mathsf{ct}_{\mathsf{SKE}}$ attached to $\mathcal{A}$'s query. If the returned MAC tag is identical to the MAC tag $\overline{\Sigma}_{\mathsf{MAC}}$ attached to $\mathcal{A}$'s query, it completes `COSMAC`. Since MAC is deterministic, this perfectly simulates the view to $\mathcal{A}$. (Note that we can rely on sEUF-CMA security instead if we assume a randomized MAC). Here, without loss of generality, we can assume $\mathsf{ct}_{\mathsf{SKE}}$ was never queried to the MAC oracle by $\mathcal{B}$. Otherwise, $\mathcal{A}$ can use the pair $(\mathsf{ct}_{\mathsf{SKE}}, \overline{\Sigma}_{\mathsf{MAC}})$ to win the anonymous blocklisting security game.

Finally, when $\mathcal{A}$ outputs a forgery $(\Sigma_{\mathsf{G}}, \mathsf{m})$ or $(\widehat{\Sigma}_{\mathsf{G}}, \widehat{\mathsf{ct}}_{\mathsf{G}})$, it parses the authentication token as $(\mathsf{ct}^*_{\mathsf{SKE}}, \overline{\Sigma}^*_{\mathsf{MAC}})$ and submits it as its forgery. Due to the winning condition of the security game, the tuple $(\mathsf{ct}^*_{\mathsf{SKE}}, \overline{\Sigma}^*_{\mathsf{MAC}})$ was never output by $\mathcal{O}_{\mathsf{Send}}$ or $\mathcal{O}_{\mathsf{UpdSend}}$. Moreover, as discussed above, we can assume this tuple was never queried to $\mathcal{O}_{\mathsf{GroupReceive}}$ or $\mathcal{O}_{\mathsf{GroupUpdReceive}}$. This implies that $\mathcal{B}$ never queried to its MAC oracle. Thus the tuple $(\mathsf{ct}^*_{\mathsf{SKE}}, \overline{\Sigma}^*_{\mathsf{MAC}})$ forms a valid forgery against the sEUF-CMA security game as desired. This completes the proof. ☐

D More Details on `QUASAR`

In this section, we provide the omitted details from Sec. 5.1.

D.1 Formal Description of `QUASAR`

We provide the formal description of `QUASAR` in Fig. 9.

D.2 Security Proof of `QUASAR`

The following establishes the correctness and security of `QUASAR`.

Theorem D.1. *The* GAM *protocol* `QUASAR` *in Fig. 9 is signing correct, global state-update correct, non-colluding unforgeable, anonymous, anonymous blocklistable, and tracing sound assuming* OWF *is one-way,* PRF *and* PRP *are pseudorandom,* MAC *is* sEUF-CMA *secure, and* KEM *is* IND-CCA *secure.*

Proof. Correctness of signing follows from the construction. In the following, we prove `QUASAR` is global state-update correct, non-colluding unforgeable, anonymous, anonymous blocklistable, and tracing sound in Lems. D.2 to D.6, respectively. Proving these lemmas completes the proof of the theorem. ☐

D.2.1 Proof of Lem. D.2

Lemma D.2. `QUASAR` *is global state-update correct.*

Proof. Assume user $u \in \mathsf{G}$ used up all of its tokens in epoch, i.e., $\Sigma_{\mathsf{G}} = \bot$. After each user (including u) run $\mathsf{UpdSend}$; the server runs $\mathsf{UpdVerify}$ on all group update authentication tokens; and the users run $\mathsf{UpdReceive}$ on all user update authentication tokens, each user v's $\mathsf{SendSEED}_v[\mathsf{epoch}+1]$ will be updated and hence the epoch in their states will be incremented by one and the counter ctr will be refreshed to one. Therefore, every user (including u) will be able to run Send after ever users update. This shows global state-update correctness. ☐

D.2.2 Proof of Lem. D.3

Lemma D.3. `QUASAR` *is non-colluding unforgeable assuming* OWF *is one-way,* PRF *is pseudorandom,* MAC *is* sEUF-CMA *secure, and* KEM *is* IND-CCA *secure.*

Proof. We first focus on the case the adversary $\mathcal{A}$ outputs $(\mathtt{label}, \mathtt{obj}) = (\mathtt{msg}, (v, \sigma, \mathsf{m}))$ for $v \in \mathcal{H}$; the other case when $\mathtt{label} = \mathtt{upd}$ is proven identically. Since it is a valid adversary, the authentication token σ is valid and traces back to some honest user $u^* \in \mathcal{H}$. That is, σ is the form $(\mathsf{epoch}^*, \widetilde{\mathsf{id}}_{\mathsf{send}}, \overline{x}^{(t)}_{u^* \to v}, \Sigma_{\mathsf{MAC}})$ where $\overline{x}^{(t)}_{u^* \to v} = \mathsf{PRF}(\mathsf{seed}_{u^* \to v}, \mathsf{epoch}^* \| t \| v)$, $(\mathsf{seed}_{u^* \to v}, t) = \mathsf{ReceiveSEED}_v[\mathsf{epoch}^*][u^*]$, and $\mathsf{MAC.Verify}(\mathsf{k}_{\mathsf{MAC}}, (\widetilde{\mathsf{id}}_{\mathsf{send}}, \overline{x}^{(t)}_{u^* \to v}, \mathsf{m}), \Sigma_{\mathsf{MAC}}) = \top$. We also have $(u^*, *, \mathsf{m}) \notin L_{\mathsf{msg}}$.

Recall there are two cases we must consider for non-colluding unforgeability: $\mathcal{C} \neq \emptyset$ (i.e., the server is honest and $\mathcal{A}$ has access to $\mathcal{O}^\star$) and $\mathcal{C} = \emptyset$ (i.e., the server is malicious and $\mathcal{A}$ has access to $\mathcal{O}$). Below, our goal is to argue that no $\mathcal{A}$ can guess x in the first case and no $\mathcal{A}$ can create Σ_{MAC} in the second case. Towards this goal, we first invoke IND-CCA security of the KEM to argue that $\mathsf{seed}_{u^* \to v}$ for $u^*, v \in \mathcal{H}$ is distributed uniformly random over $\{0,1\}^\kappa$ from the view of $\mathcal{A}$.

Game 1: This is the real non-colluding unforgeability game.

Game 2: In this game, when $\mathcal{A}$ invokes $\mathcal{O}_{\mathsf{UpdSend}}(v)$ on $\mathsf{epoch}^* - 1$, it stores a random $\overline{\mathsf{seed}}_{u^* \to v}$ in $\mathsf{ReceiveSEED}_v[\mathsf{epoch}^*][u^*]$. Note that in the previous game, it generated a user update information $(\mathsf{seed}_{u^* \to v}, \mathsf{ct}_{u^*}) \leftarrow\$ \mathsf{KEM.Enc}(\mathsf{ek}_{u^*})$ and stored $\mathsf{seed}_{u^* \to v}$ in $\mathsf{ReceiveSEED}_v[\mathsf{epoch}^*][u^*]$. Moreover, in this game, when user u^* is invoked on $\mathsf{UpdReceive}$ with user update information ct_{u^*}, it simply uses $\overline{\mathsf{seed}}_{u^* \to v}$ rather than decrypting it by $\mathsf{KEM.Dec}(\mathsf{dk}_{u^*}, \mathsf{ct}_{u^*})$. Otherwise, the game proceeds identically to the previous game.

```
$\mathsf{Init}(1^\kappa, \mathsf{G})$
$\mathsf{gsk} \leftarrow_\$ \{0,1\}^\kappa$   / Group secret key
$\mathsf{k_{MAC}} \leftarrow \mathsf{PRF}(\mathsf{gsk}, 0)$   / MAC key for group
$\mathsf{K} \leftarrow \mathsf{PRF}(\mathsf{gsk}, 1)$   / PRF key for group
/ Prepare empty list and public key for users
foreach $u \in \mathsf{G}$ do
    $\mathsf{SendSEED}_u[*], \mathsf{ReceiveSEED}_u[*] := \perp$
    $(\mathsf{ek}_u, \mathsf{dk}_u) \leftarrow \mathsf{KEM.KeyGen}(1^\kappa)$
$\mathsf{EK} \leftarrow (u, \mathsf{ek}_u)_{u \in \mathsf{G}}$   / $\mathsf{EK}[u] = \mathsf{ek}_u$
$\mathsf{epoch} := 0$
$\mathsf{k_{epoch}} \leftarrow \mathsf{PRF}(\mathsf{K}, \mathsf{epoch})$   / Permutation keys for tokens
foreach $u \in \mathsf{G}$ do
    $(\mathsf{ReceiveSEED}_u, Y_u, (\mathsf{ct}_v)_{v \in \mathsf{G}})$
        $\leftarrow$ *gen-auth-token$(\mathsf{G}, \mathsf{EK}, \mathsf{epoch}, u, \mathsf{ReceiveSEED}_u)$
    foreach $v \in \mathsf{G}$ do
        $(\mathsf{seed}_{v \to u}, 1) \leftarrow \mathsf{ReceiveSEED}_u[\mathsf{epoch}][v]$
        $\mathsf{SendSEED}_v[\mathsf{epoch}][u] \leftarrow \mathsf{seed}_{v \to u}$
foreach $u \in \mathsf{G}$ do
    $\mathsf{SEED}_u := (\mathsf{SendSEED}_u, \mathsf{ReceiveSEED}_u)$
    $\mathsf{st}_u \leftarrow (\mathsf{G}, \mathsf{k_{MAC}}, \mathsf{K}, \mathsf{EK}, \mathsf{dk}_u, \mathsf{epoch}, 1, \mathsf{SEED}_u)$
$\mathsf{DB}[*] := \perp$   / Prepare empty database for Sv
for $(u, j) \in \mathsf{G} \times [NT]$ do
    $\widetilde{j} \leftarrow \mathsf{PRP}(\mathsf{k_{epoch}}, j)$   / Reorder public tokens and store it in DB
    $\mathsf{DB}[\mathsf{epoch}][\mathsf{idx}(u)][\widetilde{j}] \leftarrow Y_u[j]$
$\mathsf{pp} \leftarrow \mathsf{DB}$   / $\mathsf{DB}[0] \in (\{0,1\}^\kappa)^{N \times NT}$
return $(\mathsf{pp}, (\mathsf{st}_u)_{u \in \mathsf{G}})$
```

```
$\mathsf{Send}(\mathsf{st}_u, \mathsf{m})$
parse $(\mathsf{G}, \mathsf{k_{MAC}}, \mathsf{K}, \mathsf{EK}, \mathsf{dk}_u, \mathsf{epoch}, \mathsf{ctr}, \mathsf{SEED}_u) \leftarrow \mathsf{st}_u$
if $\mathsf{ctr} \geq T - 1$ then return $\perp$   / Need to update tokens
$\mathsf{ctr}' \leftarrow \mathsf{ctr} + 1$
$\Sigma_\mathsf{G} \leftarrow (\mathsf{epoch}$, *attach-auth-token$(\mathsf{st}_u$, msg :: $\mathsf{m}))$
$\mathsf{st}'_u \leftarrow (\mathsf{G}, \mathsf{k_{MAC}}, \mathsf{K}, \mathsf{EK}, \mathsf{dk}_u, \mathsf{epoch}, \mathsf{ctr}', \mathsf{SEED}_u)$
return $(\mathsf{st}'_u, \Sigma_\mathsf{G})$
```

```
$\mathsf{Verify}(\mathsf{pp}, \Sigma_\mathsf{G}, \mathsf{m})$
parse $\mathsf{DB} \leftarrow \mathsf{pp}$
try $(\mathsf{pp}', (\sigma_i)_{i \in [N]}) \leftarrow$ *verify-auth-token$(\mathsf{pp}, \Sigma_\mathsf{G})$
return $(\mathsf{pp}', (\sigma_i)_{i \in [N]})$
```

```
$\mathsf{Receive}(\mathsf{st}_u, \sigma, \mathsf{m})$
try $(\mathsf{st}'_u, b, v) \leftarrow$ *trace-sender$(\mathsf{st}_u, \sigma)$
return $(\mathsf{st}'_u, b, v)$
```

```
$\mathsf{UpdSend}(\mathsf{st}_u)$
parse $(\mathsf{G}, \mathsf{k_{MAC}}, \mathsf{K}, \mathsf{EK}, \mathsf{dk}_u, \mathsf{epoch}, \mathsf{ctr}, \mathsf{SEED}_u) \leftarrow \mathsf{st}_u$
/ Cannot update again if others haven't
if $\mathsf{ctr} = T$ then return $\perp$
parse $(\mathsf{SendSEED}_u, \mathsf{ReceiveSEED}_u) \leftarrow \mathsf{SEED}_u$
$\mathsf{epoch}' \leftarrow \mathsf{epoch} + 1$   / Move to new epoch
$\mathsf{k_{epoch'}} \leftarrow \mathsf{PRF}(\mathsf{K}, \mathsf{epoch}')$
/ Create seeds/tokens for new epoch
$(\mathsf{ReceiveSEED}_u, Y_u, (\mathsf{ct}_v)_{v \in \mathsf{G}})$
    $\leftarrow$ *gen-auth-token$(\mathsf{G}, \mathsf{EK}, \mathsf{epoch}', u, \mathsf{ReceiveSEED}_u)$
$\widetilde{Y}_u[*] := \perp$   / Reorder public tokens to be uploaded to Sv
for $j \in [NT]$ do
    $\widetilde{j} \leftarrow \mathsf{PRP}(\mathsf{k_{epoch}}, j)$
    $\widetilde{Y}_u[\widetilde{j}] \leftarrow Y_u[j]$
$\widehat{\mathsf{ct}}_\mathsf{G} \leftarrow (\mathsf{idx}(u), \widetilde{Y}_u, (\mathsf{ct}_v)_{v \in \mathsf{G}})$
$\widehat{\Sigma}_\mathsf{G} \leftarrow (\mathsf{epoch}$, *attach-auth-token$(\mathsf{st}_u$, upd :: $(\mathsf{ct}_v)_{v \in \mathsf{G}}))$
$\mathsf{SendSEED}_u[\mathsf{epoch}] \leftarrow \perp$   / Remove seeds from previous epoch
/ Update own seed/private tokens for next epoch
$\mathsf{SendSEED}_u[\mathsf{epoch}'][u] \leftarrow \mathsf{ReceiveSEED}_u[\mathsf{epoch}'][u]$
$\mathsf{SEED}_u \leftarrow (\mathsf{SendSEED}_u, \mathsf{ReceiveSEED}_u)$
/ Max out counter to $T$ but do not increment epoch yet
$\mathsf{st}'_u \leftarrow (\mathsf{G}, \mathsf{k_{MAC}}, \mathsf{K}, \mathsf{EK}, \mathsf{dk}_u, \mathsf{epoch}, T, \mathsf{SEED}_u)$
return $(\mathsf{st}'_u, \widehat{\Sigma}_\mathsf{G}, \widehat{\mathsf{ct}}_\mathsf{G})$
```

```
$\mathsf{UpdVerify}(\mathsf{pp}, \widehat{\Sigma}_\mathsf{G}, \widehat{\mathsf{ct}}_\mathsf{G})$
parse $\mathsf{DB} \leftarrow \mathsf{pp}$
try $(\mathsf{pp}', (\widehat{\sigma}_i)_{i \in [N]}) \leftarrow$ *verify-auth-token$(\mathsf{pp}$, upd :: $(\widehat{\Sigma}_\mathsf{G}, \widehat{\mathsf{ct}}_\mathsf{G}))$
parse $\left(\mathsf{epoch}, \widetilde{\mathsf{id}}_\mathsf{send}, (x_i, \Sigma_{\mathsf{MAC},i})\right)_{i \in [N]} \leftarrow \widehat{\Sigma}_\mathsf{G}$
parse $(\mathsf{id_{receive}}, \widetilde{Y}, (\mathsf{ct}_i)_{i \in [N]}) \leftarrow \widehat{\mathsf{ct}}_\mathsf{G}$
req $\mathsf{DB}[\mathsf{epoch} + 1][\mathsf{id_{receive}}] = \perp$   / No public token for $\mathsf{id_{receive}}$ in $\mathsf{epoch} + 1$
$\mathsf{DB}[\mathsf{epoch} + 1][\mathsf{id_{receive}}] \leftarrow \widetilde{Y}$   / Update DB for next epoch
foreach $i \in [N]$ do
    $\widehat{\mathsf{ct}}_i \leftarrow \mathsf{ct}_i$
return $(\mathsf{pp}', (\widehat{\sigma}_i, \widehat{\mathsf{ct}}_i)_{i \in [N]})$
```

```
$\mathsf{UpdReceive}(\mathsf{st}_u, \widehat{\sigma}, \widehat{\mathsf{ct}})$
parse $(\mathsf{G}, \mathsf{k_{MAC}}, \mathsf{K}, \mathsf{EK}, \mathsf{dk}_u, \mathsf{epoch}, \mathsf{ctr}, \mathsf{SEED}_u) \leftarrow \mathsf{st}_u$
parse $(\mathsf{SendSEED}_u, \mathsf{ReceiveSEED}_u) \leftarrow \mathsf{SEED}_u$
try $(\mathsf{st}'_u, b, v) \leftarrow$ *trace-sender$(\mathsf{st}_u, \widehat{\sigma}, \widehat{\mathsf{ct}})$
if $v = u$ then return $(\mathsf{st}_u, \top, u)$   / $\mathsf{SendSEED}_u$ is already updated with $\mathsf{seed}_{u \to u}$
req $\mathsf{SendSEED}_u[\mathsf{epoch} + 1][v] = \perp$   / $v$ hasn't updated yet in $\mathsf{epoch} + 1$
$\mathsf{seed}_{u \to v} \leftarrow \mathsf{KEM.Dec}(\mathsf{dk}_u, \widehat{\mathsf{ct}})$   / Seed used by $u$ to send message to $v$ in $\mathsf{epoch} + 1$
$\mathsf{SendSEED}_u[\mathsf{epoch} + 1][v] \leftarrow \mathsf{seed}_{u \to v}$
$\mathsf{SEED}_u \leftarrow (\mathsf{SendSEED}_u, \mathsf{ReceiveSEED}_u)$
if $\forall w \in \mathsf{G}, \mathsf{SendSEED}_u[\mathsf{epoch} + 1][w] \neq \perp$ then
    / Increment epoch and refresh counter to 1 if everybody updated
    $\mathsf{st}'_u \leftarrow (\mathsf{G}, \mathsf{k_{MAC}}, \mathsf{K}, \mathsf{EK}, \mathsf{dk}_u, \mathsf{epoch} + 1, 1, \mathsf{SEED}_u)$
else $\mathsf{st}'_u \leftarrow (\mathsf{G}, \mathsf{k_{MAC}}, \mathsf{K}, \mathsf{EK}, \mathsf{dk}_u, \mathsf{epoch}, \mathsf{ctr}, \mathsf{SEED}_u)$
return $(\mathsf{st}'_u, \top, v)$
```

Figure 9: `QUASAR`: An anonymous group authenticated messaging protocol with tracing soundness. `label` :: `obj` denotes that `obj` has a type `label`, where `label` is a special string used nowhere else. The helper algorithms used above are detailed in Fig. 10

```
Func *gen-auth-token$(\mathsf{G}, \mathsf{EK}, \mathsf{epoch}, u, \mathsf{ReceiveSEED})$
req $\mathsf{ReceiveSEED}[\mathsf{epoch}] = \bot$
foreach $v \in \mathsf{G}$ do
  $(\mathsf{seed}_{v \to u}, \mathsf{ct}_v) \leftarrow\$ \mathsf{KEM.Enc}(\mathsf{EK}[v])$
  $\mathsf{ReceiveSEED}[\mathsf{epoch}][v] \leftarrow (\mathsf{seed}_{v \to u}, 1)$
  foreach $t \in [T]$ do
    / $t$-th private token $v$ sends to $u$ in epoch
    $x^{(t)}_{v \to u} \leftarrow \mathsf{PRF}(\mathsf{seed}_{v \to u}, \mathsf{epoch} \| t \| u)$
    $y^{(t)}_{v \to u} \leftarrow \mathsf{OWF}(x^{(t)}_{v \to u})$   / Corresponding $t$-th public token
/ Arrange all public tokens in order of users $\mathsf{G} = (u_1, \cdots, u_N)$
$Y \leftarrow \Big(y^{(1)}_{u_1 \to u}, \ldots, y^{(T)}_{u_1 \to u},$
          $\ldots, y^{(1)}_{u_N \to u}, \ldots, y^{(T)}_{u_N \to u}\Big) \in (\{0,1\}^{\kappa})^{NT}$
return $(\mathsf{ReceiveSEED}, Y, (\mathsf{ct}_v)_{v \in \mathsf{G}})$
```

```
Func *attach-auth-token$(\mathsf{st}_u, \texttt{label} :: \texttt{obj})$
parse $(\mathsf{G}, \mathsf{k_{MAC}}, \mathsf{K}, \mathsf{EK}, \mathsf{dk}_u, \mathsf{epoch}, \mathsf{ctr}, \mathsf{SEED}_u) \leftarrow \mathsf{st}_u$
parse $(\mathsf{SendSEED}_u, \mathsf{ReceiveSEED}_u) \leftarrow \mathsf{SEED}_u$
$\mathsf{k_{epoch}} \leftarrow \mathsf{PRF}(\mathsf{K}, \mathsf{epoch})$
foreach $v \in \mathsf{G}$ do
  $\mathsf{seed}_{u \to v} \leftarrow \mathsf{SendSEED}_u[\mathsf{epoch}][v]$
  $x^{(\mathsf{ctr})}_{u \to v} \leftarrow \mathsf{PRF}(\mathsf{seed}_{u \to v}, \mathsf{epoch} \| \mathsf{ctr} \| v)$   / $\mathsf{ctr}$-th private token $u$ sends to $v$
  if $\texttt{label} = \texttt{msg}$ then
    parse $\mathsf{m} \leftarrow \texttt{obj}$
    $\Sigma_{\mathsf{MAC},v} \leftarrow\$ \mathsf{MAC.TagGen}\Big(\mathsf{k_{MAC}}, \Big(x^{(\mathsf{ctr})}_{u \to v}, \mathsf{m}\Big)\Big)$
  elseif $\texttt{label} = \texttt{obj}$ then
    parse $(\mathsf{ct}_v)_{v \in \mathsf{G}} \leftarrow \texttt{obj}$
    / Sign $v$'s ciphertext using otsk from $v$
    $\Sigma_{\mathsf{MAC},v} \leftarrow\$ \mathsf{MAC.TagGen}\Big(\mathsf{k_{MAC}}, \Big(x^{(\mathsf{ctr})}_{u \to v}, \mathsf{ct}_v\Big)\Big)$
/ Compute permuted index of the corresponding public token in DB
$\mathsf{id_{send}} \leftarrow \mathsf{ctr} + (\mathsf{idx}(u) - 1) \cdot T$
$\widetilde{\mathsf{id}}_{\mathsf{send}} \leftarrow \mathsf{PRP}(\mathsf{k_{epoch}}, \mathsf{id_{send}})$   / $\mathsf{id_{send}}, \widetilde{\mathsf{id}}_{\mathsf{send}} \in [NT]$
return $\Big(\widetilde{\mathsf{id}}_{\mathsf{send}}, \Big(x^{(\mathsf{ctr})}_{u \to v}, \Sigma_{\mathsf{MAC},v}\Big)_{v \in \mathsf{G}}\Big)$
```

```
Func *verify-auth-token$(\mathsf{pp}, \Sigma_{\mathsf{G}})$
$\mathsf{DB} \leftarrow \mathsf{pp}$   / $\mathbf{Y} \in (\{0,1\}^{\kappa})^{N \times NT}$
parse $\Big(\mathsf{epoch}, \widetilde{\mathsf{id}}_{\mathsf{send}}, (x_i, \Sigma_{\mathsf{MAC},i})\Big)_{i \in [N]} \leftarrow \Sigma_{\mathsf{G}}$
foreach $i \in [N]$ do
  if $\mathsf{DB}[\mathsf{epoch}][i][\widetilde{\mathsf{id}}_{\mathsf{send}}] \neq \mathsf{OWF}(x_i)$ then
    return $\bot$
/ If check passes, set user authentication tokens and
/ delete column to update DB
foreach $i \in [N]$ do
  $\sigma_i \leftarrow (\mathsf{epoch}, \widetilde{\mathsf{id}}_{\mathsf{send}}, x_i, \Sigma_{\mathsf{MAC},i})$
  $\mathsf{DB}[\mathsf{epoch}][i][\widetilde{\mathsf{id}}_{\mathsf{send}}] \leftarrow \bot$
$\mathsf{pp}' \leftarrow \mathsf{DB}$
return $(\mathsf{pp}', (\sigma_i)_{i \in [N]})$
```

```
Func *trace-sender$(\mathsf{st}_u, \sigma, \mathsf{m})$
parse $(\mathsf{G}, \mathsf{k_{MAC}}, \mathsf{K}, \mathsf{EK}, \mathsf{dk}_u, \mathsf{epoch}, \mathsf{ctr}, \mathsf{SEED}_u) \leftarrow \mathsf{st}_u$
parse $(\mathsf{SendSEED}_u, \mathsf{ReceiveSEED}_u) \leftarrow \mathsf{SEED}_u$
parse $(\mathsf{epoch}', \widetilde{\mathsf{id}}_{\mathsf{send}}, x, \Sigma_{\mathsf{MAC}}) \leftarrow \sigma$
req $\mathsf{epoch} = \mathsf{epoch}'$
$\mathsf{k_{epoch}} \leftarrow \mathsf{PRF}(\mathsf{K}, \mathsf{epoch})$
$\mathsf{id_{send}} \leftarrow \mathsf{PRP}^{-1}(\mathsf{k_{epoch}}, \widetilde{\mathsf{id}}_{\mathsf{send}})$
$i \leftarrow \lfloor \mathsf{id_{send}}/T \rfloor + 1$   / Index of the user who sent the message
$t \leftarrow \mathsf{id_{send}} - \lfloor \mathsf{id_{send}}/T \rfloor \cdot T$   / $t$-th token from user in epoch
$v \leftarrow i$-th user in $\mathsf{G}$   / $v \in \mathsf{G}$ and $\mathsf{id_{send}} = t + (\mathsf{idx}(v) - 1) \cdot T$
$(\mathsf{seed}_{v \to u}, \mathsf{ctr}) \leftarrow \mathsf{ReceiveSEED}_u[\mathsf{epoch}][v]$
req $\mathsf{ctr} = t$   / Require that token wasn't used yet
$x^{(t)}_{v \to u} \leftarrow \mathsf{PRF}(\mathsf{seed}_{v \to u}, \mathsf{epoch} \| t \| u)$
if $x^{(t)}_{v \to u} \neq x$
    $\vee\, \mathsf{MAC.Verify}(\mathsf{k_{MAC}}, \Big(\widetilde{\mathsf{id}}_{\mathsf{send}}, x^{(\mathsf{ctr})}_{v \to u}, \mathsf{m}\Big), \Sigma_{\mathsf{MAC}}) = \bot$ then
  return $(\mathsf{st}_u, \bot, \bot)$
$\mathsf{ReceiveSEED}_u[\mathsf{epoch}][v] \leftarrow (\mathsf{seed}_{v \to u}, \mathsf{ctr} + 1)$   / Mark $t$-th private token to be used
$\mathsf{SEED}_u \leftarrow (\mathsf{SendSEED}_u, \mathsf{ReceiveSEED}_u)$
$\mathsf{st}'_u \leftarrow (\mathsf{G}, \mathsf{k_{MAC}}, \mathsf{K}, \mathsf{EK}, \mathsf{dk}_u, \mathsf{epoch}, \mathsf{ctr}, \mathsf{SEED}_u)$
return $(\mathsf{st}'_u, \top, v)$
```

Figure 10: Helper functions used by `QUASAR`.

These two games are indistinguishable assuming the $\mathsf{IND\text{-}CCA}$ security of the KEM. Concretely, an adversary $\mathcal{B}$ against the $\mathsf{IND\text{-}CCA}$ security game is given $(\mathsf{ek}^*, \mathsf{ct}^*, \mathsf{seed}^*)$ as its challenge. It then embeds ek^* into ek_{u^*} and simulates the non-colluding unforgeability security game to $\mathcal{A}$. When $\mathcal{A}$ invokes $\mathcal{O}_{\mathsf{UpdSend}}(v)$ on epoch^*, it sets the user update information ct_{u^*} as ct^* and stores seed^* in $\mathsf{ReceiveSEED}_v[\mathsf{epoch}^*][u^*]$. Moreover, when user u^* is invoked on $\mathsf{UpdReceive}$ with user update information ct'_{u^*}, it queries its decryption oracle if $\mathsf{ct}'_{u^*} \neq \mathsf{ct}^*$. Otherwise, it simply sets the decapsulated value as seed^* without running the decapsulation algorithm. Assuming the $(1-\delta)$-correctness, $\mathcal{B}$ correctly simulates Game 1 (resp. Game 2) when the challenge bit is 1 (resp. 0) as desired.

We are now ready to invoke the pseudorandomness of the PRF.

Game 3: In this game, when $\mathcal{A}$ invokes $\mathcal{O}_{\mathsf{UpdSend}}(v)$ on $\mathsf{epoch}^* - 1$, it samples T random private tokens for u^*: $\overline{x}^{(t)}_{u^* \to v} \leftarrow\!\$ \{0,1\}^\kappa$ for $t \in [T]$ (cf. line 7 of `*gen-auth-token` in Fig. 10). Note that in the previous game, it generated them by using $\overline{\mathsf{seed}}_{u^* \to v}$ and the PRF. Moreover, in this game, when user u^* runs Send or $\mathsf{UpdSend}$ with $\mathsf{SendSEED}_{u^*}[\mathsf{epoch}][w] = \overline{\mathsf{seed}}_{u^* \to v}$ for some $w \in \mathsf{G}$, it uses $\left(\overline{x}^{(t)}_{u^* \to v}\right)_{t \in [T]}$ when $\mathsf{epoch} = \mathsf{epoch}^*$ rather than generating them via the PRF. Otherwise, if $\mathsf{epoch} \neq \mathsf{epoch}^*$, it samples T random fresh private tokens for this epoch and uses them. Here, note that we do not necessarily have $(w, \mathsf{epoch}) = (u, \mathsf{epoch}^*)$ since a malicious user w can reuse the user update information ct_{u^*} that v sent to u^* at some later $\mathsf{epoch} > \mathsf{epoch}^*$.

Finally, when `*trace-sender`$(\mathsf{st}_u, \sigma, \mathsf{m})$ with $\sigma = (\mathsf{epoch}^*, \widetilde{\mathsf{id}}_{\mathsf{send}}, x, \Sigma_{\mathsf{MAC}})$ is invoked it checks if $\widetilde{\mathsf{id}}_{\mathsf{send}}$ is the permuted index of user u^*'s ctr-th token (cf. lines 7 to 9 of `*trace-sender` in Fig. 10), where $(\mathsf{ReceiveSEED}_v[\mathsf{epoch}^*][u^*] = (\overline{\mathsf{seed}}_{u^* \to v}, \mathsf{ctr}))$. If not, it proceeds as in Game 2. Otherwise, it checks if $x = \overline{x}^{(\mathsf{ctr})}_{u^* \to v}$ rather than using the PRF. Otherwise, the game proceeds identically to the previous game.

Assuming the pseudorandomness of the PRF, it is clear that Game 2 and Game 3 are indistinguishable.

We now make two case distinctions based on $\mathcal{C} \neq \emptyset$ or not. Let us consider the first case where the adversary is the set of malicious sender and the server is honest. In Game 3, notice the private tokens $\left(\overline{x}^{(t)}_{u^* \to v}\right)_{t \in [T]}$ in epoch^* are distributed uniformly random over $(\{0,1\}^\kappa)^T$ and shared only between the honest users u^* and v. Due to the one-wayness of OWF, the corresponding public tokens $\left(y^{(t)}_{u^* \to v} = \mathsf{OWF}(\overline{x}^{(t)}_{u^* \to v})\right)_{t \in [T]}$ do not leak the private tokens in any meaningful way. Moreover, since each private tokens are tied to epoch^* and a counter $t \in [T]$, $\overline{x}^{(t)}_{u^* \to v}$ is only revealed when u^* is invoked on the t-th Send or $\mathsf{UpdSend}$ in epoch^*. As explained above, u^* may reveal the same private token $\overline{x}^{(t)}_{u^* \to v}$ multiple times in epoch^* on the t-th invocation of Send or $\mathsf{UpdSend}$ since a malicious user w can set $\overline{x}^{(t)}_{u^* \to v} = \overline{x}^{(t)}_{u^* \to v}$ by replaying the ciphertext ct_{u^*}. However, as long as the adversary $\mathcal{A}$ cannot reuse the private token for a different $t' \neq t$ or $\mathsf{epoch}' \neq \mathsf{epoch}^*$, this does not constitute in an attack. Indeed, since we assume an honest server, the private tokens sent by v for a particular pair (epoch, t) is processed correctly, and in particular, the adversary cannot reuse them in a different $(\mathsf{epoch}', t') \neq (\mathsf{epoch}, t)$. Thus, we conclude that $\mathcal{A}$ cannot output a private token that hasn't been used by the honest user u^*.

Let us consider the second case where the adversary is the malicious server but all users are honest. The hardness almost immediately follows from assuming $\mathsf{sEUF\text{-}CMA}$ security of the MAC as the malicious server does not know the MAC key $\mathsf{k}_{\mathsf{MAC}}$. Concretely, if $(u^*, *, \mathsf{m}) \notin L_{\mathsf{msg}}$, then user u^* will have never signed the tuple $(\overline{x}^{(t)}_{u^* \to v}, \mathsf{m})$. Moreover, since $\overline{x}^{(t)}_{u^* \to v}$ is now sampled uniformly random over $\{0,1\}^\kappa$, the probability that another user $w \neq u^*$ signed the tuple is negligible. Therefore, we conclude that if an adversary outputs a MAC tag Σ_{MAC} for such a tuple, it breaks $\mathsf{sEUF\text{-}CMA}$ security. Here, to simulate the $\mathsf{sEUF\text{-}CMA}$ security game, we rely on the pseudorandomness of the PRF to sample the MAC key $\mathsf{k}_{\mathsf{MAC}}$ directly without running PRF.

This completes the proof of the lemma. ☐

D.2.3 Proof of Lem. D.4

Lemma D.4. `QUASAR` *is anonymous assuming assuming* PRF *and* PRP *are pseudorandom and* KEM *is* $\mathsf{IND\text{-}CCA}$ *secure.*

Proof. Before we get into the proof, we make some useful observations. Due to global state-update correctness, if u_0 and u_1 can both run Send, the generated global authentication tokens Σ^0_{G} and Σ^1_{G} must contain the same epoch^*. Moreover, due to $\mathcal{A}$'s winning condition that all the users verify, every user in G must have been invoked on $\mathcal{O}_{\mathsf{UpdSend}}$ at $\mathsf{epoch}^* - 1$ as otherwise, they will not be able to verify u_0 and u_1's user authentication token at epoch^*. With this in mind, we perform several game hybrids, where anonymity trivially holds in the final game. The first two game transition is almost identical to those done in the proof of non-colluding unforgeability.

Game 1: This is the real anonymity game.

Game 2: In this game, for any $u \in \mathsf{G}$, when $\mathcal{A}$ invokes $\mathcal{O}_{\mathsf{UpdSend}}(u)$ on $\mathsf{epoch}^* - 1$, it stores a random $\overline{\mathsf{seed}}_{v \to u}$ in $\mathsf{ReceiveSEED}_u[\mathsf{epoch}^*][v]$ for all $v \in \mathsf{G}$. Note that in the previous game, it generated a user update information $(\mathsf{seed}_{v \to u}, \mathsf{ct}_v) \leftarrow\!\$ \mathsf{KEM.Enc}(\mathsf{ek}_v)$ and stored $\mathsf{seed}_{v \to u}$ in $\mathsf{ReceiveSEED}_u[\mathsf{epoch}^*][v]$. Moreover, in this game, when user v is invoked on $\mathsf{UpdReceive}$ with user update information ct_v, it simply uses $\overline{\mathsf{seed}}_{v \to u}$ rather than decrypting it by running $\mathsf{KEM.Dec}(\mathsf{dk}_v, \mathsf{ct}_v)$. Otherwise, the game proceeds identically to the previous game.

Following the same argument made to move from Game 1 to Game 2 in Lem. D.3, the two games are indistinguishable assuming the IND-CCA security of the KEM. We next invoke the pseudorandomness of the PRF.

Game 3: In this game, for any $u \in \mathsf{G}$, when $\mathcal{A}$ invokes $\mathcal{O}_{\mathsf{UpdSend}}(u)$ on $\mathsf{epoch}^* - 1$, it samples T random private tokens for every $v \in \mathsf{G}$: $\overline{x}^{(t)}_{v\to u} \leftarrow_{\$} \{0,1\}^{\kappa}$ for $t \in [T]$ (cf. line 7 of `*gen-auth-token` in Fig. 10). Note that in the previous game, it generated them by using $\overline{\mathsf{seed}}_{v\to u}$ and the PRF. Moreover, in this game, when any user v runs Send or UpdSend with $\mathsf{SendSEED}_v[\mathsf{epoch}][w] = \overline{\mathsf{seed}}_{v\to u}$, it uses $\left(\overline{x}^{(t)}_{v\to u}\right)_{t\in[T]}$ when $\mathsf{epoch} = \mathsf{epoch}^*$ rather than generating them via the PRF. Otherwise, if $\mathsf{epoch} \neq \mathsf{epoch}^*$, it samples T random fresh private tokens for this epoch and uses them. Here, note that we do not necessarily have $(w, \mathsf{epoch}) = (u, \mathsf{epoch}^*)$ since the malicious server may reuse the user update information ct_v that u sent to v by modifying user w's user update information — this is due to group update authentication tokens not being cryptographically tied to the group update information.

Finally, when `*trace-sender`$(\mathsf{st}_u, \sigma, \mathsf{m})$ with $\sigma = (\mathsf{epoch}^*, \widetilde{\mathsf{id}}_{\mathsf{send}}, x, \Sigma_{\mathsf{MAC}})$ is invoked (as part of Receive or UpdReceive) it checks if $\widetilde{\mathsf{id}}_{\mathsf{send}}$ is the permuted index of user v's ctr-th token (cf. lines 7 to 9 of `*trace-sender` in Fig. 10), where $(\mathsf{ReceiveSEED}_u[\mathsf{epoch}^*][v] = (\overline{\mathsf{seed}}_{v\to u}, \mathsf{ctr}))$. If not, it proceeds as in Game 2. Otherwise, it checks if $x = \overline{x}^{(\mathsf{ctr})}_{v\to u}$ rather than using the PRF. Otherwise, the game proceeds identically to the previous game.

Assuming the pseudorandomness of the PRF, it is clear that Game 2 and Game 3 are indistinguishable. At this point, the private tokens $\left(\overline{x}^{(t)}_{v\to u}\right)_{(t,v,u)\in[T]\times\mathsf{G}^2}$ generated via $\mathcal{O}_{\mathsf{UpdSend}}$ in epoch^* are independently distributed. We next argue that the challenge users $u_b \in \mathsf{G}$, $b \in \{0,1\}$ must also receive these private tokens $\left(\overline{x}^{(t)}_{u_b\to v}\right)_{(t,v)\in[T]\times\mathsf{G}}$. Namely, we consider the following game.

Game 4: In this game, the game aborts if there exists a user $v \in \mathsf{G}$ such that the following holds for either $b \in \{0,1\}$:

$$\mathsf{ReceiveSEED}_v[\mathsf{epoch}^*][u_b] \neq \mathsf{SendSEED}_{u_b}[\mathsf{epoch}^*][v]. \tag{2}$$

Put differently, the game aborts if the user update information ct_v from v to u_b on epoch^* was modified.

We argue that this modification has no impact on $\mathcal{A}$'s advantage. Assume if the adversary $\mathcal{A}$ won under this condition. Due to the winning condition of the anonymity game, the group authentication token generated by u_0 and u_1 must be accepted by user v. As we established in Game 3, the private tokens $\left(\overline{x}^{(t)}_{u_b\to v}\right)_{(t,b)\in[T]\times\{0,1\}}$ stored in v's state at epoch^* are distributed uniformly random over $\{0,1\}^{\kappa}$ from the view of $\mathcal{A}$. Hence, if Equation (2) holds, then the probability that v accepts u_b's private token is at most $1/2^{\kappa}$. Thus, the advantage of $\mathcal{A}$ cannot change with all but a negligible probability.

At this point, we have that the private tokens included in the group authentication tokens of u_0 and u_1 are uniformly distributed. We finally invoke the pseudorandomness of the PRF one last time along with that of the PRP.

Game 5: In this game, the permutation key $\mathsf{k}_{\mathsf{epoch}}$ at each epoch is no longer sampled. Instead, it samples a uniformly random permutation over $[NT]$ at each epoch and uses that instead. Otherwise, it is identical to the previous game.

Since the group secret key gsk is hidden to the adversary, we can first invoke the pseudorandomness of the PRF to argue that the permutation key at each epoch is uniform random and independent. Then, we invoke the pseudorandomness of the PRP to replace the PRP by a truly random permutation.

By Game 5, $\widetilde{\mathsf{id}}_{\mathsf{seed},0}$ and $\widetilde{\mathsf{id}}_{\mathsf{seed},1}$ included in the group authentication tokens Σ^0_{G} and Σ^1_{G}, respectively, no longer leak the identity of u_0 and u_1. Thus combining everything, we conclude that Σ^0_{G} and Σ^1_{G} no longer leak the identity of u_0 and u_1. This completes the proof of the lemma. ☐

D.2.4 Proof of Lem. D.5

Lemma D.5. QUASAR *is anonymous blocklistable assuming* OWF *is one-way,* PRF *is pseudorandom, and* KEM *is IND-CCA secure.*

Proof. The proof follows almost directly from the proof of non-colluding unforgeability (cf. Lem. D.3). In fact, the proof is much simpler since unlike in the non-colluding unforgeability game, we can assume without of generality that the adversary $\mathcal{A}$ never queries $\mathcal{O}_{\mathsf{Receive}}$ or $\mathcal{O}_{\mathsf{UpdReceive}}$ such that Verify or UpdVerify verify, respectively. This is because such a query can be used to directly win the anonymous blocklistable game since the private tokens can only be used once. (Concretely, we only require IND-CPA security of the KEM). Then, following an almost exact proof, we can argue that the only information leakage of the private tokens corresponding to the public tokens stored inside DB maintained by the server are only from the public parameter $\mathsf{pp} = \mathsf{DB}$. Hence, under the one-wayness of OWF, no $\mathcal{A}$ can output a group (update) authentication token for which the server verifies. ☐

D.2.5 Proof of Lem. D.6

Lemma D.6. `QUASAR` *is (unconditionally) tracing sound.*

Proof. We first focus on the case the adversary $\mathcal{A}$ outputs $(\texttt{label}, \texttt{obj}) = (\texttt{msg}, (\Sigma_{\mathsf{G}}, \mathsf{m}))$. The game then runs $(\mathsf{pp}', (\sigma_i)_{i \in [N]}) \leftarrow \mathsf{Verify}(\mathsf{pp}, \Sigma_{\mathsf{G}}, \mathsf{m})$. Since $\mathsf{pp}' \neq \perp$, $\Sigma_{\mathsf{G}} = \left(\mathsf{epoch}, \widetilde{\mathsf{id}}_{\mathsf{send}}, (x_i, \Sigma_{\mathsf{MAC},i})\right)_{i \in [N]}$ for some $\widetilde{\mathsf{id}}_{\mathsf{send}} \in [N]$, $\Sigma_{\mathsf{MAC},i}$ is a valid MAC tag under the message tuple $(\widetilde{\mathsf{id}}_{\mathsf{send}}, x_i, \mathsf{m})$, and $\mathsf{DB}[\mathsf{epoch}][i][\widetilde{\mathsf{id}}_{\mathsf{send}}] = \mathsf{OWF}(x_i)$ for all $i \in [N]$, where DB is the database stored in the public parameter pp. Then, a user authentication token for user $u \in \mathsf{G} \cap \mathcal{H}$ is set as $\sigma_u = (\mathsf{epoch}, \widetilde{\mathsf{id}}_{\mathsf{send}}, x_{\mathsf{idx}(u)}, \Sigma_{\mathsf{MAC},\mathsf{idx}(u)})$. Noticing that each honest users maintain the same group secret key gsk and σ_u contains the same $(\mathsf{epoch}, \widetilde{\mathsf{id}}_{\mathsf{send}})$, if $(\mathsf{st}'_u, b_u, v_u) \leftarrow \mathsf{Receive}(\mathsf{st}_u, \sigma_u, \mathsf{m})$ and $b_u = \top$, then v_u must be the same user for all honest user (cf. lines 7 to 9 of `*trace-sender` in Fig. 10). Finally, the case when $\mathcal{A}$ outputs $(\texttt{label}, \texttt{obj}) = (\texttt{upd}, (\widehat{\Sigma}_{\mathsf{G}}, \widehat{\mathsf{ct}}_{\mathsf{G}}))$ is proven identically. This completes the proof of the lemma. □

D.3 Alternatives to Global State Updates

Lastly, we discuss below some ideas to mitigate the shortcoming of global state updates discussed in Sec. 5.1.

Unbalancing the Number of Tokens. Depending on the group, some users may have a higher frequency of communication than others. In such scenarios, if we allocate all the users the same number of tokens T, some may take significantly longer to exhaust their budget than others. As a result, those who have already consumed their tokens may face prolonged waiting periods before everyone updates their state.

`QUASAR` can be easily modified to a protocol where each user is allocated different number of tokens $(T_u)_{u \in \mathsf{G}}$. In fact, the users can adaptively modify the number of tokens per epoch T_u^{epoch}, where one epoch corresponds to one global state updates. This is possible because, in `QUASAR`, an unlimited number of tokens can be minted from the seed using the PRF. As before, the tokens are then shuffled by using a PRP, whose keys are updated each epoch. While the server can observe the number of total tokens $\sum_{u \in \mathsf{G}} T_u^{\mathsf{epoch}}$ fluctuating between epochs, due to anonymity, it cannot deduce how many tokens are allocated to each user. From the server's perspective, it could be a group where everybody is actively talking (i.e., $T_u \approx T_v$ for any $u, v \in \mathsf{G}$) or a group where only one user is allocated most of the tokens (i.e., $T_u \gg T_v$ for any $v \in \mathsf{G} \backslash \{u\}$).

Relaxing the Global State Updates Restriction. While unbalancing the number of tokens, explained in App. D.3, mitigates one aspect of the shortcoming of global state updates, it does not solve the leading issue: once a user exhausts its tokens, it must wait till all the users update to be able to send messages again. An approach to address this is to let the users who exhausted its tokens in $\mathsf{epoch}^* - 1$ to send messages in epoch^* with the limited updates it received. More specifically, assuming a subset of the group $\overline{\mathsf{G}} \subset \mathsf{G}$ performed an update for epoch^*, any $u \in \overline{\mathsf{G}}$ can start sending a message using the user authentication tokens received from all the users in $v \in \overline{\mathsf{G}}$. Visually, the rows in Fig. 7 corresponding to users in $\overline{\mathsf{G}}$ are filled with tokens while the other rows remain empty. In the extreme case when u is the only user that performed an update for epoch^*, then $\overline{\mathsf{G}} = \{u\}$. It is worth noting that this approach is possible with `QUASAR` but not something that any GAM protocol can do.

The upside of this approach is that users can locally update their state via $\mathsf{UpdSend}$ and can start sending messages again without waiting for all the other users to perform an update. Any user $v \in \mathsf{G} \backslash \overline{\mathsf{G}}$ can at any point perform an update for epoch^* to join $\overline{\mathsf{G}}$. Moreover, users $v \in \overline{\mathsf{G}}$ can trace back any user authentication token exchanged in epoch^* to a specific user in $\overline{\mathsf{G}}$, thus achieving a limited scope of tracing soundness.

The downside of this approach is that the anonymity set of the sender is limited to the size of the subgroup $\overline{\mathsf{G}}$. This is because only the users in $\overline{\mathsf{G}}$ are capable of sending a message with a token from epoch^*. In the aforementioned extreme case, when $\overline{\mathsf{G}} = \{u\}$, then u is the only user with tokens from epoch^*. As a result, while the server does not learn who u is, it can *link* together any messages u sent in epoch^*. Another downside is that there are no tracing soundness guarantees for the users $w \in \mathsf{G}$ *outside* the subgroup $\overline{\mathsf{G}}$ as they have not minted any private tokens for epoch^*. Tracing soundness is restored only once w updates.

Combining `QUASAR` with `COSMAC`$^+$. When $|\overline{\mathsf{G}}| \ll |\mathsf{G}|$, we have seen that the anonymity for the users in $\overline{\mathsf{G}}$ and the tracing soundness of $\mathsf{G} \backslash \overline{\mathsf{G}}$ is weakened. In this case, it may be more appropriate to use `COSMAC`$^+$ instead of `QUASAR`, given that `COSMAC`$^+$ is anonymous, more efficient but lacks tracing soundness. To balance these trade-offs, we propose a hybrid protocol that primarily employs `QUASAR`, but switches to `COSMAC`$^+$ when a predetermined number of users have not yet performed an update for the next epoch^*. This approach combines the benefits of both `QUASAR` and `COSMAC`$^+$.

E More Details on `STARS`

In this section, we provide the omitted details from Sec. 5.2.

E.1 Formal Description of `STARS`

We provide the formal description of `STARS`. The main difference between `QUASAR` is that `STARS` replaces one-time tokens and MAC tags with one-time signatures (OTS). Specifically, the exchanged messages are signed by the OTS, effectively making the GAM protocol *standard* unforgeable. The difference is highlighted with a box in Fig. 11.

```
$\mathsf{Init}(1^\kappa, \mathsf{G})$
gsk $\leftarrow\$ \{0,1\}^\kappa$   / Group secret key
/ Prepare empty list and public key for users
foreach $u \in \mathsf{G}$ do
    $\mathsf{SendSEED}_u[*], \mathsf{ReceiveSEED}_u[*] := \bot$
    $(\mathsf{ek}_u, \mathsf{dk}_u) \leftarrow \mathsf{KEM.KeyGen}(1^\kappa)$
$\mathsf{EK} \leftarrow (u, \mathsf{ek}_u)_{u \in \mathsf{G}}$   / $\mathsf{EK}[u] = \mathsf{ek}_u$
$\mathsf{epoch} := 0$
$\mathsf{k_{epoch}} \leftarrow \mathsf{PRF}(\mathsf{gsk}, \mathsf{epoch})$   / Permutation keys for OTS otvks
foreach $u \in \mathsf{G}$ do
    $(\mathsf{ReceiveSEED}_u, Y_u, (\mathsf{ct}_v)_{v \in \mathsf{G}})$
        $\leftarrow$ *gen-auth-token$(\mathsf{G}, \mathsf{EK}, \mathsf{epoch}, u, \mathsf{ReceiveSEED}_u)$
    foreach $v \in \mathsf{G}$ do
        $(\mathsf{seed}_{v \to u}, 1) \leftarrow \mathsf{ReceiveSEED}_u[\mathsf{epoch}][v]$
        $\mathsf{SendSEED}_v[\mathsf{epoch}][u] \leftarrow \mathsf{seed}_{v \to u}$
foreach $u \in \mathsf{G}$ do
    $\mathsf{SEED}_u := (\mathsf{SendSEED}_u, \mathsf{ReceiveSEED}_u)$
    $\mathsf{st}_u \leftarrow (\mathsf{G}, \mathsf{gsk}, \mathsf{EK}, \mathsf{dk}_u, \mathsf{epoch}, 1, \mathsf{SEED}_u)$
$\mathsf{DB}[*] := \bot$   / Prepare empty database for Sv
for $(u, j) \in \mathsf{G} \times [NT]$ do
    $\widetilde{j} \leftarrow \mathsf{PRP}(\mathsf{k_{epoch}}, j)$   / Reorder OTS otvks and store it in DB
    $\mathsf{DB}[\mathsf{epoch}][\mathsf{idx}(u)][\widetilde{j}] \leftarrow Y_u[j]$
$\mathsf{pp} \leftarrow \mathsf{DB}$   / $\mathsf{DB}[0] \in (\{0,1\}^\kappa)^{N \times NT}$
return $\left(\mathsf{pp}, (\mathsf{st}_u)_{u \in \mathsf{G}}\right)$
```

```
$\mathsf{Send}(\mathsf{st}_u, \mathsf{m})$
parse $(\mathsf{G}, \mathsf{gsk}, \mathsf{EK}, \mathsf{dk}_u, \mathsf{epoch}, \mathsf{ctr}, \mathsf{SEED}_u) \leftarrow \mathsf{st}_u$
if $\mathsf{ctr} \geq T - 1$ then return $\bot$   / Need to update tokens
$\mathsf{ctr}' \leftarrow \mathsf{ctr} + 1$
$\Sigma_\mathsf{G} \leftarrow (\mathsf{epoch}$, *attach-auth-token$(\mathsf{st}_u$, msg :: $\mathsf{m}))$
$\mathsf{st}'_u \leftarrow (\mathsf{G}, \mathsf{gsk}, \mathsf{EK}, \mathsf{dk}_u, \mathsf{epoch}, \mathsf{ctr}', \mathsf{SEED}_u)$
return $(\mathsf{st}'_u, \Sigma_\mathsf{G})$
```

```
$\mathsf{Verify}(\mathsf{pp}, \Sigma_\mathsf{G}, \mathsf{m})$
parse $\mathsf{DB} \leftarrow \mathsf{pp}$
try $(\mathsf{pp}', (\sigma_i)_{i \in [N]})$
        $\leftarrow$ *verify-auth-token$(\mathsf{pp}, \boxed{\texttt{msg} :: (\Sigma_\mathsf{G}, \mathsf{m})})$
return $(\mathsf{pp}', (\sigma_i)_{i \in [N]})$
```

```
$\mathsf{Receive}(\mathsf{st}_u, \sigma, \mathsf{m})$
try $(\mathsf{st}'_u, b, v) \leftarrow$ *trace-sender$(\mathsf{st}_u, \sigma, \mathsf{m})$
return $(\mathsf{st}'_u, b, v)$
```

```
$\mathsf{UpdSend}(\mathsf{st}_u)$
parse $(\mathsf{G}, \mathsf{gsk}, \mathsf{EK}, \mathsf{dk}_u, \mathsf{epoch}, \mathsf{ctr}, \mathsf{SEED}_u) \leftarrow \mathsf{st}_u$
/ Cannot update again if others haven't
if $\mathsf{ctr} = T$ then return $\bot$
parse $(\mathsf{SendSEED}_u, \mathsf{ReceiveSEED}_u) \leftarrow \mathsf{SEED}_u$
$\mathsf{epoch}' \leftarrow \mathsf{epoch} + 1$
$\mathsf{k_{epoch'}} \leftarrow \mathsf{PRF}(\mathsf{gsk}, \mathsf{epoch}')$
/ Create seeds/OTS keys for new epoch
$(\mathsf{ReceiveSEED}_u, Y_u, (\mathsf{ct}_v)_{v \in \mathsf{G}})$
    $\leftarrow$ *gen-auth-token$(\mathsf{G}, \mathsf{EK}, \mathsf{epoch}', u, \mathsf{ReceiveSEED}_u)$
$\widetilde{Y}_u[*] := \bot$   / Reorder OTS otvks to be uploaded to Sv
for $j \in [NT]$ do
    $\widetilde{j} \leftarrow \mathsf{PRP}(\mathsf{k_{epoch}}, j)$
    $\widetilde{Y}_u[\widetilde{j}] \leftarrow Y_u[j]$
$\widehat{\mathsf{ct}}_\mathsf{G} \leftarrow (\mathsf{idx}(u), \widetilde{Y}_u, (\mathsf{ct}_v)_{v \in \mathsf{G}})$
$\widehat{\Sigma}_\mathsf{G} \leftarrow (\mathsf{epoch}$, *attach-auth-token$(\mathsf{st}_u$, upd :: $(\mathsf{ct}_v)_{v \in \mathsf{G}}))$
$\mathsf{SendSEED}_u[\mathsf{epoch}] \leftarrow \bot$   / Remove seeds from previous epoch
/ Update own seed/OTS otsks for next epoch
$\mathsf{SendSEED}_u[\mathsf{epoch}'][u] \leftarrow \mathsf{ReceiveSEED}_u[\mathsf{epoch}'][u]$
$\mathsf{SEED}_u \leftarrow (\mathsf{SendSEED}_u, \mathsf{ReceiveSEED}_u)$
/ Max out counter to $T$ but do not increment epoch yet
$\mathsf{st}'_u \leftarrow (\mathsf{G}, \mathsf{gsk}, \mathsf{EK}, \mathsf{dk}_u, \mathsf{epoch}, T, \mathsf{SEED}_u)$
return $(\mathsf{st}'_u, \widehat{\Sigma}_\mathsf{G}, \widehat{\mathsf{ct}}_\mathsf{G})$
```

```
$\mathsf{UpdVerify}(\mathsf{pp}, \widehat{\Sigma}_\mathsf{G}, \widehat{\mathsf{ct}}_\mathsf{G})$
parse $\mathsf{DB} \leftarrow \mathsf{pp}$
try $(\mathsf{pp}', (\widehat{\sigma}_i)_{i \in [N]}) \leftarrow$ *verify-auth-token$(\mathsf{pp}, \boxed{\texttt{upd} :: (\widehat{\Sigma}_\mathsf{G}, \widehat{\mathsf{ct}}_\mathsf{G})})$
parse $\left(\mathsf{epoch}, \widetilde{\mathsf{id}}_\mathsf{send}, (x_1, \ldots, x_N)\right) \leftarrow \widehat{\Sigma}_\mathsf{G}$
parse $(\mathsf{id}_\mathsf{receive}, \widetilde{Y}, (\mathsf{ct}_i)_{i \in [N]}) \leftarrow \widehat{\mathsf{ct}}_\mathsf{G}$
req $\mathsf{DB}[\mathsf{epoch} + 1][\mathsf{id}_\mathsf{receive}] = \bot$   / No OTS otvks for $\mathsf{id}_\mathsf{receive}$ in epoch $+ 1$
$\mathsf{DB}[\mathsf{epoch} + 1][\mathsf{id}_\mathsf{receive}] \leftarrow \widetilde{Y}$   / Update DB for next epoch
foreach $i \in [N]$ do
    $\widehat{\mathsf{ct}}_i \leftarrow \mathsf{ct}_i$
return $(\mathsf{pp}', (\widehat{\sigma}_i, \widehat{\mathsf{ct}}_i)_{i \in [N]})$
```

```
$\mathsf{UpdReceive}(\mathsf{st}_u, \widehat{\sigma}, \widehat{\mathsf{ct}})$
parse $(\mathsf{G}, \mathsf{gsk}, \mathsf{EK}, \mathsf{dk}_u, \mathsf{epoch}, \mathsf{ctr}, \mathsf{SEED}_u) \leftarrow \mathsf{st}_u$
parse $(\mathsf{SendSEED}_u, \mathsf{ReceiveSEED}_u) \leftarrow \mathsf{SEED}_u$
try $(\mathsf{st}_u, b, v) \leftarrow$ *trace-sender$(\mathsf{st}_u, \widehat{\sigma}, \widehat{\mathsf{ct}})$
if $v = u$ then return $(\mathsf{st}_u, \top, u)$   / $\mathsf{SendSEED}_u$ is already updated with $\mathsf{seed}_{u \to u}$
req $\mathsf{SendSEED}_u[\mathsf{epoch} + 1][v] = \bot$   / $v$ hasn't updated yet in epoch $+ 1$
$\mathsf{seed}_{u \to v} \leftarrow \mathsf{KEM.Dec}(\mathsf{dk}_u, \widehat{\mathsf{ct}})$   / Seed used by $u$ to send message to $v$ in epoch $+ 1$
$\mathsf{SendSEED}_u[\mathsf{epoch} + 1][v] \leftarrow \mathsf{seed}_{u \to v}$
$\mathsf{SEED}_u \leftarrow (\mathsf{SendSEED}_u, \mathsf{ReceiveSEED}_u)$
if $\forall w \in \mathsf{G}, \mathsf{SendSEED}_u[\mathsf{epoch} + 1][w] \neq \bot$ then
    / Increment epoch and refresh counter to 1 if everybody updated
    $\mathsf{st}'_u \leftarrow (\mathsf{G}, \mathsf{gsk}, \mathsf{EK}, \mathsf{dk}_u, \mathsf{epoch} + 1, 1, \mathsf{SEED}_u)$
else $\mathsf{st}'_u \leftarrow (\mathsf{G}, \mathsf{gsk}, \mathsf{EK}, \mathsf{dk}_u, \mathsf{epoch}, \mathsf{ctr}, \mathsf{SEED}_u)$
return $(\mathsf{st}'_u, \top, v)$
```

Figure 11: `STARS`: An anonymous group authenticated messaging protocol with standard unforgeability. The main differences between `QUASAR` is highlighted by a box. `label` :: `obj` denotes that `obj` has a type `label`, where `label` is a special string used nowhere else. The helper algorithms used above are detailed in Fig. 12.

```
Func $\texttt{*gen-auth-token}(\mathsf{G},\mathsf{EK},\mathsf{epoch},u,\mathsf{ReceiveSEED})$
req $\mathsf{ReceiveSEED}[\mathsf{epoch}] = \bot$
foreach $v \in \mathsf{G}$ do
    $(\mathsf{seed}_{v\to u},\mathsf{ct}_v) \leftarrow\$\, \mathsf{KEM.Enc}(\mathsf{EK}[v])$
    $\mathsf{ReceiveSEED}[\mathsf{epoch}][v] \leftarrow (\mathsf{seed}_{v\to u},1)$
    foreach $t \in [T]$ do
        / $t$-th OTS key $v$ uses to send a message to $u$ in epoch
        $\mathsf{r}^{(t)}_{v\to u} \leftarrow \mathsf{PRF}(\mathsf{seed}_{v\to u},\mathsf{epoch}\|t\|u)$
        $\boxed{(\mathsf{otvk},\mathsf{otsk}) \leftarrow \mathsf{OTS.KeyGen}(1^\kappa;\mathsf{r}^{(t)}_{v\to u})}$
        $\boxed{y^{(t)}_{v\to u} := \mathsf{otvk}}$
/ Arrange all OTS otvks in order of users $\mathsf{G} = (u_1,\cdots,u_N)$
$Y \leftarrow \Big(y^{(1)}_{u_1\to u},\ldots,y^{(T)}_{u_1\to u},$
            $\ldots,y^{(1)}_{u_N\to u},\ldots,y^{(T)}_{u_N\to u}\Big) \in (\{0,1\}^\kappa)^{NT}$
return $(\mathsf{ReceiveSEED},Y,(\mathsf{ct}_v)_{v\in\mathsf{G}})$
```

```
Func $\texttt{*attach-auth-token}(\mathsf{st}_u,\texttt{label} :: \texttt{obj})$
parse $(\mathsf{G},\mathsf{gsk},\mathsf{EK},\mathsf{dk}_u,\mathsf{epoch},\mathsf{ctr},\mathsf{SEED}_u) \leftarrow \mathsf{st}_u$
parse $(\mathsf{SendSEED}_u,\mathsf{ReceiveSEED}_u) \leftarrow \mathsf{SEED}_u$
foreach $v \in \mathsf{G}$ do
    $\mathsf{seed}_{u\to v} \leftarrow \mathsf{SendSEED}_u[\mathsf{epoch}][v]$
    / Derive ctr-th OTS key $u$ uses to send a message to $v$ in epoch
    $\mathsf{r}^{(\mathsf{ctr})}_{u\to v} \leftarrow \mathsf{PRF}(\mathsf{seed}_{u\to v},\mathsf{epoch}\|\mathsf{ctr}\|v)$
    $\boxed{(\mathsf{otvk},\mathsf{otsk}) \leftarrow \mathsf{OTS.KeyGen}(1^\kappa;\mathsf{r}^{(\mathsf{ctr})}_{u\to v})}$
    if $\texttt{label} = \texttt{msg}$ then
        parse $\mathsf{m} \leftarrow \texttt{obj}$
        $\boxed{x^{(\mathsf{ctr})}_{u\to v} := \mathsf{sig} \leftarrow \mathsf{OTS.Sign}(\mathsf{otsk},\mathsf{m})}$
    elseif $\texttt{label} = \texttt{obj}$ then
        parse $(\mathsf{ct}_v)_{v\in\mathsf{G}} \leftarrow \texttt{obj}$
        / Sign $v$'s ciphertext using otsk from $v$
        $\boxed{x^{(\mathsf{ctr})}_{u\to v} := \mathsf{sig} \leftarrow \mathsf{OTS.Sign}(\mathsf{otsk},\mathsf{ct}_v)}$
$\mathsf{k}_{\mathsf{epoch}} \leftarrow \mathsf{PRF}(\mathsf{gsk},\mathsf{epoch})$
/ Compute permuted index of the corresponding OTS otvk in DB
$\mathsf{id}_{\mathsf{send}} \leftarrow \mathsf{ctr} + (\mathsf{idx}(u)-1)\cdot T$
$\widetilde{\mathsf{id}}_{\mathsf{send}} \leftarrow \mathsf{PRP}(\mathsf{k}_{\mathsf{epoch}},\mathsf{id}_{\mathsf{send}})$   / $\mathsf{id}_{\mathsf{send}},\widetilde{\mathsf{id}}_{\mathsf{send}} \in [NT]$
return $\Big(\widetilde{\mathsf{id}}_{\mathsf{send}},\Big(x^{(\mathsf{ctr})}_{u\to v}\Big)_{v\in\mathsf{G}}\Big)$
```

```
Func $\texttt{*verify-auth-token}(\mathsf{pp},\boxed{\texttt{label} :: \texttt{obj}})$
$\mathsf{DB} \leftarrow \mathsf{pp}$
$\mathbf{Y} \leftarrow \mathsf{DB}[\mathsf{epoch}]$   / $\mathbf{Y} \in (\{0,1\}^\kappa)^{N\times NT}$
if $\texttt{label} = \texttt{msg}$ then
    parse $\Big(\Big(\mathsf{epoch},\widetilde{\mathsf{id}}_{\mathsf{send}},(x_1,\ldots,x_N)\Big),\mathsf{m}\Big) \leftarrow \texttt{obj}$
    foreach $i \in [N]$ do
        $\boxed{\text{if } \mathsf{OTS.Verify}(\mathbf{Y}[i][\widetilde{\mathsf{id}}_{\mathsf{send}}],\mathsf{m},x_i) = \bot \text{ then}}$
            return $\bot$
elseif $\texttt{label} = \texttt{upd}$ then
    parse $(\widehat{\Sigma}_{\mathsf{G}},\widehat{\mathsf{ct}}_{\mathsf{G}}) \leftarrow \texttt{obj}$
    parse $\Big(\mathsf{epoch},\widetilde{\mathsf{id}}_{\mathsf{send}},(x_1,\ldots,x_N)\Big) \leftarrow \widehat{\Sigma}_{\mathsf{G}}$
    parse $\Big(\mathsf{id}_{\mathsf{receive}},\widetilde{Y},(\mathsf{ct}_i)_{i\in\mathsf{G}}\Big) \leftarrow \widehat{\mathsf{ct}}_{\mathsf{G}}$
    foreach $i \in [N]$ do
        $\boxed{\text{if } \mathsf{OTS.Verify}(\mathbf{Y}[i][\widetilde{\mathsf{id}}_{\mathsf{send}}],\mathsf{ct}_i,x_i) = \bot \text{ then}}$
            return $\bot$
/ If check passes, set user authentication tokens and
/ delete column from $\mathbf{Y}$
foreach $i \in [N]$ do
    $\sigma_i \leftarrow (\mathsf{epoch},\widetilde{\mathsf{id}}_{\mathsf{send}},x_i)$
    $\mathbf{Y}[i][\widetilde{\mathsf{id}}_{\mathsf{send}}] \leftarrow \bot$
$\mathsf{DB}[\mathsf{epoch}] \leftarrow \mathbf{Y}$   / Update DB
$\mathsf{pp}' \leftarrow \mathsf{DB}$
return $(\mathsf{pp}',(\sigma_i)_{i\in[N]})$
```

```
Func $\texttt{*trace-sender}(\mathsf{st}_u,\sigma,\mathsf{m})$
parse $(\mathsf{G},\mathsf{gsk},\mathsf{EK},\mathsf{dk}_u,\mathsf{epoch},\mathsf{ctr},\mathsf{SEED}_u) \leftarrow \mathsf{st}_u$
parse $(\mathsf{SendSEED}_u,\mathsf{ReceiveSEED}_u) \leftarrow \mathsf{SEED}_u$
parse $(\mathsf{epoch}',\widetilde{\mathsf{id}}_{\mathsf{send}},x) \leftarrow \sigma$
req $\mathsf{epoch} = \mathsf{epoch}'$
$\mathsf{k}_{\mathsf{epoch}} \leftarrow \mathsf{PRF}(\mathsf{gsk},\mathsf{epoch})$
$\mathsf{id}_{\mathsf{send}} \leftarrow \mathsf{PRP}^{-1}(\mathsf{k}_{\mathsf{epoch}},\widetilde{\mathsf{id}}_{\mathsf{send}})$
$i \leftarrow \lfloor \mathsf{id}_{\mathsf{send}}/T \rfloor + 1$   / Index of the user who sent the message
$t \leftarrow \mathsf{id}_{\mathsf{send}} - \lfloor \mathsf{id}_{\mathsf{send}}/T \rfloor \cdot T$   / $t$-th OTS otvk from user in epoch
$v \leftarrow i\text{-th user in } \mathsf{G}$   / $v \in \mathsf{G}$ and $\mathsf{id}_{\mathsf{send}} = t + (\mathsf{idx}(v)-1)\cdot T$
$(\mathsf{seed}_{v\to u},\mathsf{ctr}) \leftarrow \mathsf{ReceiveSEED}_u[\mathsf{epoch}][v]$
req $\mathsf{ctr} = t$   / Require that token wasn't used yet
$\mathsf{r}^{(\mathsf{ctr})}_{v\to u} \leftarrow \mathsf{PRF}(\mathsf{seed}_{u\to v},\mathsf{epoch}\|\mathsf{ctr}\|u)$
$\boxed{(\mathsf{otvk},\mathsf{otsk}) \leftarrow \mathsf{OTS.KeyGen}(1^\kappa;\mathsf{r}^{(\mathsf{ctr})}_{v\to u})}$
$\boxed{\text{if } \mathsf{OTS.Verify}(\mathsf{otvk},\mathsf{m},x) = \bot \text{ then return } (\mathsf{st}_u,\bot,\bot)}$
/ Mark $t$-th OTS otsk to be used
$\mathsf{ReceiveSEED}_u[\mathsf{epoch}][v] \leftarrow (\mathsf{seed}_{v\to u},\mathsf{ctr}+1)$
$\mathsf{SEED}_u \leftarrow (\mathsf{SendSEED}_u,\mathsf{ReceiveSEED}_u)$
$\mathsf{st}'_u \leftarrow (\mathsf{G},\mathsf{gsk},\mathsf{EK},\mathsf{dk}_u,\mathsf{epoch},\mathsf{ctr},\mathsf{SEED}_u)$
return $(\mathsf{st}'_u,\top,v)$
```

Figure 12: Helper functions used by `STARS`. The main differences between `QUASAR` is highlighted by a box.

E.2 Security Proof of `STARS`

We prove that `STARS` is correct and secure. The proof follows almost identically to the proof given for `QUASAR`.

Theorem E.1. *The* GAM *protocol* `STARS` *in Fig. 11 is signing correct, global state-update correct, standard unforgeable, anonymous, anonymous blocklistable, and tracing sound assuming* PRF *and* PRP *are pseudorandom,* OTS *is* $\mathsf{EUF\text{-}CMA}$, KEM *is* $\mathsf{IND\text{-}CCA}$ *secure.*

Proof. Correctness of signing follows from construction. Moreover, we omit the proof of global state-update correctness and tracing soundness as they follow almost identically to the proof given for `QUASAR` (cf. Apps. D.2.1 and D.2.5). In the following, we prove `STARS` is unforgeable, anonymous, and anonymous blocklistable in Lems. E.2 to E.4, respectively. Proving these lemmas completes the proof of the theorem. $\square$

E.2.1 Proof of Lem. E.2

Lemma E.2. `STARS` *is unforgeable assuming* PRF *is pseudorandom,* OTS *is* $\mathsf{EUF\text{-}CMA}$*, and* KEM *is IND-CCA secure.*

Proof. The proof is identical up to Game 2 of the non-colluding unforgeability proof given for `QUASAR` (cf. App. D.2.2). We pick up the proof from Game 3. Below, recall we focus on two honest users u and v (that may possibly be $u = v$) and assume the adversary $\mathcal{A}^*$ forges a group (update) authentication on epoch^*.

Game 3: In this game, when $\mathcal{A}$ invokes $O_{\mathsf{UpdSend}}(u)$ on $\mathsf{epoch}^* - 1$, it samples T fresh randomness for each $w \in \mathsf{G}$ used to run algorithm $\mathsf{OTS.KeyGen}$: $\overline{r}^{(t)}_{w \to u} \leftarrow_{\$} \{0,1\}^*$ for $t \in [T]$ (cf. line 7 of `*gen-auth-token` in Fig. 10). Note that in the previous game, it generated them by running $\mathsf{PRF}(\overline{\mathsf{seed}}_{w \to u}, \mathsf{epoch}^* \| t \| u)$. Moreover, in this game, when user v runs Send or $\mathsf{UpdSend}$ with $\mathsf{SendSEED}_v[\mathsf{epoch}][w] = \overline{\mathsf{seed}}_{v \to u}$, it uses $\left(\overline{r}^{(t)}_{w \to u}\right)_{t \in [T]}$ when $\mathsf{epoch} = \mathsf{epoch}^*$ rather than generating them via the PRF. Otherwise, if $\mathsf{epoch} \neq \mathsf{epoch}^*$, it samples T random fresh randomness for the pair (w, epoch) and uses them. Here, note that we do not necessarily have $(w, \mathsf{epoch}) = (u, \mathsf{epoch}^*)$ since a malicious user w can reuse the user update information ct_v that u sent to v at some later epoch.

Finally, when `*trace-sender`(st_u, σ) with $\sigma = (\mathsf{epoch}^*, \widetilde{\mathsf{id}}_{\mathsf{send}}, x, \mathsf{m})$ is invoked (as part of $\mathsf{Receive}$ or $\mathsf{UpdReceive}$) it checks if $\widetilde{\mathsf{id}}_{\mathsf{send}}$ is the permuted index of user v's ctr-th token (cf. lines 7 to 9 of `*trace-sender` in Fig. 10), where $(\mathsf{ReceiveSEED}_u[\mathsf{epoch}^*][v] = (\overline{\mathsf{seed}}_{v \to u}, \mathsf{ctr}))$. If not, it proceeds as in Game 2. Otherwise, it generates the OTS keys $(\mathsf{otvk}, \mathsf{otsk})$ using randomness $\overline{r}^{(\mathsf{ctr})}_{v \to u}$ and check if the signature message pair $(x = \mathsf{sig}, \mathsf{m})$ verifies under otvk. Otherwise, the game proceeds identically to the previous game.

Assuming the pseudoranomness of the PRF, it is clear that $\mathsf{Game\ 2}$ and $\mathsf{Game\ 3}$ are indistinguishable.

We are now ready to invoke the $\mathsf{EUF\text{-}CMA}$ security of OTS to show standard unforgeability. In $\mathsf{Game\ 3}$, notice the OTS signing keys $\left(\mathsf{otsk}^{(t)}_{v \to u} := \overline{x}^{(t)}_{v \to u}\right)_{t \in [T]}$ in epoch^* are distributed uniformly random over the signing key space and shared only between the honest users u and v. The only information on these signing keys are provided to the adversary $\mathcal{A}$ through the corresponding public OTS verification keys $\left(\mathsf{otvk}^{(t)}_{v \to u} := y^{(t)}_{v \to u}\right)_{t \in [T]}$. Moreover, since each OTS signing keys are generated by fresh randomness $\overline{\mathsf{r}}^{(t)}_{v \to u}$, $\mathsf{otsk}^{(t)}_{v \to u}$ is only used when v is invoked on the t-th Send or $\mathsf{UpdSend}$ in epoch^* to prepare a signature for u. Note that this is why we modified `QUASAR` to included the sender of $\mathsf{seed}_{v \to u}$ to derive the randomness $\overline{\mathsf{r}}^{(t)}_{v \to u}$ (cf. line 7 of `*gen-auth-token` in Fig. 10). Without this modification, a malicious user w that sets $\mathsf{SendSEED}_v[\mathsf{epoch}^*][w] = \overline{\mathsf{seed}}_{v \to u}$ can trick v to sign multiple times using otsk since $\overline{\mathsf{r}}^{(t)}_{v \to u}$ will only depend on epoch^* and counter t.

Due to the winning condition, if $\mathcal{A}$ outputs a valid forgery that traces to v in epoch^*, it must be on a message that v hasn't signed yet. This is exactly the winning condition of the $\mathsf{EUF\text{-}CMA}$ security game of OTS. Moreover, an adversary against the $\mathsf{EUF\text{-}CMA}$ security can simulate $\mathsf{Game\ 3}$ to $\mathcal{A}$ by embedding its challenge into one of its $\mathsf{otsk}^{(t)}_{v \to u}$ since each OTS signing key is only used once. Finally, the case when $\mathcal{A}$ outputs $(\mathtt{label}, \mathtt{obj}) = (\mathtt{upd}, (\widehat{\Sigma}_{\mathsf{G}}, \widehat{\mathsf{ct}}_{\mathsf{G}}))$ is proven identically. This completes the proof of the lemma. $\square$

E.2.2 Proof of Lem. E.3

Lemma E.3. `STARS` *is anonymous assuming assuming* PRF *and* PRP *are pseudorandom,* OTS *is* $\mathsf{EUF\text{-}CMA}$*, and* KEM *is* $\mathsf{IND\text{-}CCA}$ *secure.*

Proof. As the proof is a straightforward modification of the anonymity proof of `QUASAR` (cf. App. D.2.3), we only sketch the proof. The only modification is how we argue indistinguishability of $\mathsf{Game\ 3}$ and $\mathsf{Game\ 4}$ in Lem. D.4. In `QUASAR`, we used the entropy of the private token to argue that these two games are statistically indistinguishable. In `STARS`, we use the $\mathsf{EUF\text{-}CMA}$ security of the OTS. In particular, if the challenge users u_0 or u_1 generate a user authentication token for which user v accepts under condition Equation (2), then that token can be used to win the $\mathsf{EUF\text{-}CMA}$ security game. This completes the proof of the lemma. $\square$

E.2.3 Proof of Lem. E.4

Lemma E.4. `STARS` *is anonymous blocklistable assuming* PRF *is pseudorandom,* OTS *is* EUF-CMA*, and* KEM *is IND-CCA secure.*

Proof. The proof follows almost directly from the proof of unforgeability (cf. Lem. E.2), combined with the discussion provided in the proof of anonymous blocklistability of `QUASAR` (cf. App. D.2.4). ☐

Contents

1 Introduction **1**
1.1 Background . . . 1
1.2 Our Contributions . . . 2

2 Setting: Authentication in SGM **3**
2.1 Secure Group Messaging and Our Goal . . . 3
2.2 Environment . . . 4
2.3 Threat Model for Authentication . . . 4
2.4 Modeling Choices and Simplifications . . . 5

3 Group Authenticated Messaging Protocol **6**
3.1 Definition . . . 6
3.2 Correctness . . . 7
3.3 Security . . . 7

4 `COSMOS`: Authentication with One-Time Tokens **10**
4.1 Construction of `COSMOS` . . . 10
4.2 `COSMAC`: An Anonymous `COSMOS` with Anonymous Blocklisting . . . 10
4.3 Optimizations of `COSMOS` and `COSMAC` . . . 13

5 Anonymous and Tracing Sound GAMs **13**
5.1 `QUASAR`: Anonymous Authentication with Tokens . . . 13
5.2 `STARS` and `GEMSTARS` . . . 16

6 Running GAM Protocols on MLS **17**
6.1 Authentication in a Static Group . . . 17
6.2 Authentication in a Dynamic Group . . . 18

7 Bandwidth Efficiency Analysis **18**
7.1 Instantiation . . . 18
7.2 Efficiency . . . 19

8 Open Problems and Future Work **21**

A Other Related Work **26**

B Preliminary: Cryptographic Tools **27**

C Security Proofs for `COSMOS` and `COSMAC` **29**
C.1 Security Proof of `COSMOS` . . . 29
C.2 Security Proof of `COSMAC` . . . 30

D More Details on `QUASAR` **31**
D.1 Formal Description of `QUASAR` . . . 31
D.2 Security Proof of `QUASAR` . . . 31
D.3 Alternatives to Global State Updates . . . 36

E More Details on `STARS` **36**
E.1 Formal Description of `STARS` . . . 36
E.2 Security Proof of `STARS` . . . 39