A Formal Treatment of End-to-End Encrypted Cloud Storage

Matilda Backendal [1], Hannah Davis[2], Felix Günther [3], Miro Haller [4], and Kenneth G. Paterson [1]

[1]Department of Computer Science, ETH Zurich, Zurich, Switzerland
[2]Seagate Technology, Minneapolis, USA
[3]IBM Research Europe – Zurich, Zurich, Switzerland
[4]Department of Computer Science & Engineering, University of California San Diego, USA

Abstract

Users increasingly store their data in the cloud, thereby benefiting from easy access, sharing, and redundancy. To additionally guarantee security of the outsourced data even against a server compromise, some service providers have started to offer end-to-end encrypted (E2EE) cloud storage. With this cryptographic protection, only legitimate owners can read or modify the data. However, recent attacks on the largest E2EE providers have highlighted the lack of solid foundations for this emerging type of service.

In this paper, we address this shortcoming by initiating the formal study of E2EE cloud storage. We give a formal syntax to capture the core functionality of a cloud storage system, capturing the real-world complexity of such a system's constituent interactive protocols. We then define game-based security notions for confidentiality and integrity of a cloud storage system against a fully malicious server. We treat both selective and fully adaptive client compromises. Our notions are informed by recent attacks on E2EE cloud storage providers. In particular we show that our syntax is rich enough to capture the core functionality of MEGA and that recent attacks on it arise as violations of our security notions. Finally, we present an E2EE cloud storage system that provides all core functionalities and that is both efficient and provably secure with respect to our selective security notions. Along the way, we discuss challenges on the path towards bringing the security of cloud storage up to par with other end-to-end primitives, such as secure messaging and TLS.

Keywords: end-to-end encryption · cloud storage · provable security

Contents

1 Introduction

The *cloud*, which outsources data storage and computation to third party services, is one of the prevalent computing paradigms today. Our focus in this paper is on *cloud storage.* Aimed at consumers and enterprises alike, cloud storage relieves users of worrying about backups and data availability, and allows the data to be accessed from anywhere. But what about other security properties, beyond availability? Users should be rightly concerned about the confidentiality and integrity of their data too. Today the largest service providers such as Google and Microsoft expect users to trust them to look after these aspects on their behalf; data can be cryptographically protected, but using keys under the control of the service provider. If the providers are ever compromised (or compelled to provide access by law enforcement agencies) then all security guarantees may be voided.

However, there is an alternative approach for cloud storage, often referred to as End-to-End Encryption (E2EE). Here, encryption is carried out by the users themselves – under keys that they control – before uploading files to the cloud. If properly executed, this results in cryptographic confidentiality and authenticity guarantees: The service provider may learn some metadata about files, but nothing about the data content, nor can they modify individual files in an undetectable manner, thanks to the integrity protection provided by the encryption. As a consequence, security of users' data is guaranteed even in the event of a compromise of the service. Ideally, these security guarantees should continue to hold even against a fully malicious service provider, removing the need for users to trust the providers with anything but the availability of their data.

A particular challenge for E2EE cloud storage is that key management is delegated from the service provider to the individual user. But users, as humans, are not particularly good at memorizing the cryptographic key material necessary to access their data from anywhere. For this reason, security in such services is typically bootstrapped from a user-memorable secret such as a password. The reliance on passwords brings fundamental security limitations to E2EE cloud storage, since a malicious server can always carry out variants of dictionary attacks. This risk can be partially obviated with strong password policies, trading convenience for security.

The lack of trust in the service provider also brings other challenges for E2EE cloud storage services. A primary feature target is *file sharing*: the capacity for a file's owner to share access to the file with selected other users. Services offering file sharing face another difficulty related to key management. For the recipient to access the shared file, they need the relevant decryption keys. But these keys cannot be sent unprotected via the service provider if the provider is not trusted. Consequently, file sharing is typically done by encrypting the file key under the authenticated public key of the recipient, or via some out-of-band mechanism.

Despite the challenges associated to combining strong security with usability, E2EE encrypted cloud storage is on the rise. Prominent providers include MEGA [40], Nextcloud [48] and recently also Apple, who added optional support for E2EE to iCloud in 2023 [4]. To illustrate their scale, MEGA claims to have 300 million users and store more than 150 billion files, corresponding to 1000 petabytes of data [40]. Smaller E2EE providers (such as icedrive [42], pCloud [1], Seafile [43] and Tresorit [52]) also exist in abundance, with tens of millions of users collectively.

This development should delight privacy-minded users. However, recent analyses have thrown doubt on the security claims of MEGA and Nextcloud [6, 30, 3, 2] – the two largest providers of E2EE cloud storage by default – finding multiple attacks on their E2EE security. While both providers patched their systems quickly against the specific attacks, they still rely on complex, *ad hoc* cryptographic designs that lack any formal security guarantees. The MEGA case is particularly interesting – with so much data encrypted under keys known only to the users, migrating to more secure cryptographic mechanisms proved to be infeasible, since re-encryption would require user participation. Hence, MEGA initially adopted a minimal approach to patching after the initial attacks in [6], which was quickly found to be insufficient [3], leading to a second round of more complex patching. The consequences for Nextcloud were even more dire: they were forced to turn off their E2EE file sharing service altogether until a full redesign could be carried out, constituting a significant feature regression [2].

These examples highlight the value of starting with a formally analyzed cryptographic design. The security of the other above-mentioned providers is unknown, but all rely on proprietary designs that are yet to be analyzed. Overall, our confidence in the security of E2EE cloud storage lags far behind that of other E2EE applications such as messaging and browsing, which have seen significantly more design input and analysis from the cryptographic research community.

1.1 Our Contributions

In this paper, we initiate the formal study of E2EE cloud storage. Our goal is to construct a solid foundation on which we can build our understanding of its security, as a first step towards bringing it up to par with other E2EE applications. We are driven by practicality – we want E2EE cloud storage systems that are as efficient as those deployed today and with the same functionality. We hence accept passwords as the basis of security. At the same time, we want to be able to formally analyze the systems – we want security models that are rich enough to capture all the key features of E2EE cloud storage, whilst still being tractable enough to use for proofs. We offer four main contributions towards this goal:

1. We give a formal syntax capturing the core functionality of E2EE cloud storage. Our syntax encompasses user registration and authentication, as well as upload, download and sharing of files. Each of these operations is modeled as a stateful, potentially multi-round interactive protocol executed between a client and a server. For file sharing, we modularize the out-of-band component, allowing it to be instantiated in practice in a variety of different ways (e.g., by link sharing or the use of a PKI, if available).

2. We define game-based security notions for E2EE cloud storage, which we refer to as Client-to-Client (C2C) security to emphasize that the end points are (user) clients. Our notions hence focus on the setting where the adversary is a malicious service provider aiming to subvert data confidentiality or file integrity. Our C2C models allow the adversary to interact in a fine-grained manner with the clients of honest users. For example, it can execute a single step of a protocol at a time (without running it to completion), arbitrarily interleave protocol runs across users and functions, as well as deviate in any way from the honest behavior of the party it emulates. The model additionally supports compromise of honest users. In tackling this complexity, we took inspiration from the key exchange literature. We consider both selective and adaptive versions of our C2C security notions; in the former, the set of compromised users is specified up front by the adversary, while in the latter, the adversary can compromise users in a fully adaptive manner.

3. We put our model to the test and use it to formally capture some of the recent attacks on MEGA [6, 3, 30]. We thereby show that our framework is rich enough to model the complexity of a deployed system, and that our notions capture practical attacks.

4. We present a provably secure E2EE cloud storage system called CSS. Our construction makes use of standard components: an oblivious pseudorandom function (OPRF) called $\mathsf{2HashDH}$ [32, 34] is used to convert a user's password into a user-specific root key, a pseudorandom function is then used to derive separate keys for (key) encryption and authentication

from the root key. Finally, a nonce-based AEAD scheme is used to build a key hierarchy (which allows password-rotation without file re-encryption): the key encryption key is used to encrypt a user-specific master key, which in turn encrypts randomly sampled file keys, each of which protect a single file. We prove the confidentiality and integrity of CSS in our selective C2C model, achieving bounds that reflect the unavoidable brute-force password attacks by a malicious server. In principle, our CSS system, if implemented, would provide a fully functional E2EE cloud storage system with proven security guarantees.

Finally, we point to many open problems and topics for future work.

1.2 On Modeling Choices

Our model is a formal answer to the question "what security should an end-to-end encrypted cloud storage system provide?" At first glance, this seems to have a relatively simple solution, one that is outright stated in the name of the system. The term "end-to-end encryption" denotes confidentiality against a malicious service provider, and "cloud storage" suggests that users can access their files with a high degree of reliability, tamper-resistance, and portability.

However, the intuitive notion that first emerges from these constraints might look very different from our final definitions. In the abstract, a perfectly secure cloud storage system might register clients by their passwords, then take in client files and release them unchanged only to authorized clients, while revealing to the server only the number and length of files. We would like this perfect security definition to include strong guarantees, including a treatment of adaptive corruption and some protection against offline password-guessing attacks from external adversaries in the presence of an honest service provider. Without concern for undue complexity, we might even go further and also address advanced features like metadata hiding, key rotation, and forward security.

However, a good security definition also possesses two other properties: first, it should be achievable by at least one practical system, and second, it should accommodate convincing and legible proofs of security. It is in accomplishing these two goals that the "intuitive" notion described above breaks down irretrievably. Our security definitions are the result of many carefully weighed decisions and compromises that maximize feasibility, generality, and comprehensibility, while still preserving the spirit of that first intuition.

The first significant choice is our use of a game-based model. Many password-based cryptographic schemes prove security in simulation-based settings, most notably the universal composability (UC) framework [15]. Secure UC-realizable protocols emulate an "ideal functionality" that captures the intended behavior of a protocol, with only unavoidable deviations from this ideal. Because UC provides very strong security, not all ideal functionalities are realizable [18]. In particular, fully secure channels cannot be constructed without non-standard encryption primitives that are not likely to see practical adoption [17]. Cloud storage systems experience the same obstacle to UC realization as secure channels, and the UC technique used to build a "weakly-secure" channel from standard primitives does not apply due to the use of passwords as the root of trust.

A similar problem arises in the treatment of adaptive security. Because we empower our adversaries with decryption oracles, cloud storage, like encryption, requires attacks to identify their targeted ciphertexts in advance. In our CSS system, some of these ciphertexts appear in the user registration and authentication process to enable a practical key hierarchy, leading to so-called commitment issues when simulating these operations for honest users in the proof. Hence, in order for the reductions to hold, we need attackers to identify not only their target files but also the users that they will compromise in advance. Due to file sharing, the loss in security from the standard technique for avoiding this (guessing the choices of the adversary, also known as complexity leveraging) would be exponential. The consequence is that we prove selective rather than adaptive security for CSS; efficiently achieving the latter remains an open challenge.

To minimize complexity in our game-based model, we restrict ourselves to the core set of features and security properties common to existing E2EE services. Even in the simplified setting, complications arise when we consider that real-world attacks may interleave the various multi-step subprotocols of a cloud storage system and attempt to exploit state confusion between simultaneous operations of many users on many files. This type of non-atomic interchange also appears to a lesser degree in many key exchange security models. A secure cloud storage system could in theory be used to instantiate a key exchange protocol (simply put the key in a shared file!), so it is not surprising that the complexity we encounter is (at least) as high as that of key exchange.

File sharing also adds significant complexity, as it allows interaction between honest and malicious users, allowing malicious users to learn some plaintext without using decryption oracles. Tracking this in the security model requires care. Furthermore, users may expect to be able to share files without relying on a trusted service provider to authenticate the recipient. We idealize this part of the system via an "out of band" (OOB) mechanism. This makes our model general enough to capture many different choices of implementing such a channel, such as link sharing (as in MEGA), using public key encryption (as in Nextcloud), or even sending keys over a secure messenger.

We do make one choice that increases complexity, in service of future results. In our C2C setting, a malicious server can eschew any steps a protocol takes to disambiguate clients. As a consequence of this, we could obtain equivalent levels of security for CSS by using a salted hash to convert user passwords into keys as we do by using a stronger primitive like an OPRF. But the OPRF and its security properties will come to the fore in the *external* adversary setting; there we expect CSS to limit password guesses to one per client login attempt (comparable to what one hopes to achieve in a PAKE protocol).

1.3 Related Work

The reader may wonder: is E2EE cloud storage not already a solved problem? The short answer is that while many schemes for cloud storage have been proposed in the literature, only some of them address the E2EE problem, and few of them come with formal security analysis.

Table 1 lists properties covered by representative work from different areas. All of these works propose cloud storage systems, in some cases with more advanced security features such as key rotation, forward security, master key recovery, or metadata hiding. However, none of them offer a formal security proof that models all core components of cloud storage. Many consider the server to be honest (but curious), do not model file sharing, or ignore the security impact of outsourcing key material to the server. One feature in particular emerges as unique to our work: the modeling of non-atomic, multi-step protocols, with adversary-controlled arbitrary interleaving of protocol steps in the security games. This adds significant complexity to our model, but in return captures the interactive nature of cloud storage operations in practice.

Companies like Apple and Meta have recently launched systems providing encrypted backups for user data with user-controlled encryption [4, 54]. There is no public specification or analysis of E2EE in iCloud, but the WhatsApp backup system was formally analyzed in [24] using a UC approach. The latter analysis does not model a fully-malicious server but relies on a (trusted) hardware security module on the WhatsApp server instead. Like our CSS scheme, the WhatsApp system uses an OPRF to derive keys from passwords. This usage prevents pre-computation attacks on user passwords by snapshot adversaries, as well as offline attacks by an external adversary, explaining its common appearance in their setting and ours. Backups by nature only pertain to a

	formal	**share**	**pw**	**mal**	**m-sess**	**non-atomic**	**single**	**advanced**
SiRiUs [27]	○	●	○	○	○	○	●	●
DepSky [13]	○	○	○	◐	○	○	○	●
Mylar [50]	◐	●	○	●	◐	○	●	●
Burnbox [53]	●	○	○	●	●	○	●	●
OKMS [36]	●	○	●	◐	○	○	○	●
PFS [5]	●	○	○	◐	○	○	●	●
Metal [22]	●	●	○	○	○	○	○	●
Titanium [21]	●	●	○	◐	○	○	○	●
DPaSE [23]	●	○	●	●	●	○	○	○
PBCS [20]	●	○	●	●	●	○	○	○
This work	●	●	●	●	●	●	●	○

Table 1: Representative work from different areas and the features they consider: formal security analysis (*formal*), file sharing (*share*), modeling password-based security (*pw*), considering malicious servers and users (*mal*), key management across sessions on multiple devices (*m-sess*), non-atomic operations and arbitrary interleavings (*non-atomic*), single server instead of requiring multiple servers, some of which are honest (*single*), and advanced security guarantees (*advanced*) such as hiding access patterns and metadata, forward security, or temporary access revocation. We denote missing, partially covered, and fully covered properties with ○, ◐, and ●, respectively.

single user, and hence do not need to address the complexity of file sharing.

The core idea of using an OPRF in these settings seems to date back to [25], which introduced the concept of a "PRF as a service", and proposed various applications using Partially Oblivious PRFs (PO-PRFs), including enterprise password storage. Follow-up works propose password-hardened encryption and key management services [39, 38, 36, 20, 23]. These are related to our notion of E2EE cloud storage, but do not include the file sharing capability or key hierarchies to support efficient password changes. While some of them suggest cloud storage as an application, most do not specify in detail how files are encrypted. Moreover, none of these works seem to consider the malicious server setting. See also [32, 37] for the use of OPRFs in password-based authentication protocols. An excellent overview of OPRFs and their applications can be found in [19].

Closely related in (proof) techniques is the prior work on password-based encryption [9, 12]. While focused on simpler primitives, their techniques for capturing passwords in game-based security models assisted us in establishing the security achievable in password-based settings and in developing the proofs of our proposed system.

2 Preliminaries

<u>Notation.</u> If $\mathbf{w}$ is a vector then $|\mathbf{w}|$ is its length (the number of its coordinates) and $\mathbf{w}[i]$ is its i-th coordinate. By ε we denote the empty string or vector. By $x \,\|\, y$ we denote the concatenation of (bit-)strings x, y. Using a tuple (x, y) where a string is required indicates an unambiguous encoding of (x, y) into a string. If S is a finite set, then $|\mathrm{S}|$ denotes it size. We use $x \leftarrow a$ for assigning the variable x to a, and let $x \leftarrow_{\$} \mathrm{S}$ denote picking an element of S uniformly at random and assigning it to x. For conciseness, we use the bulk assignment $x, y, z \leftarrow a$ as abbreviation for $x \leftarrow a$, $y \leftarrow a$, and $z \leftarrow a$. Furthermore, we use pattern matching to assign variables to elements of tuples, e.g., $(x, (y, z)) \leftarrow (a, (b, c))$ assigns x, y, and z to a, b, and c respectively. For sets S_1 and S_2, the

shorthand $\mathrm{S}_1 \xleftarrow{\cup} \mathrm{S}_2$ denotes $\mathrm{S}_1 \leftarrow \mathrm{S}_1 \cup \mathrm{S}_2$.

If A is an algorithm, we let $y \leftarrow A^{\mathrm{O}_1,\cdots}(x_1, \ldots)$ denote running A on inputs $x_1, \ldots$ with oracle access to $\mathrm{O}_1, \ldots$, and assigning the output to y. The output assignment of randomized algorithms is denoted with $y \leftarrow\$ A^{\mathrm{O}_1,\cdots}(x_1, \ldots)$. Running time is worst case, which for an algorithm with access to oracles means across all possible replies from the oracles. We use $\top$ (top) and $\bot$ (bot) as special symbols, not in $\{0,1\}^*$, to denote acceptance and rejection, respectively.

We use the notation $\mathrm{T}[k] \leftarrow v$ to denote storing the key-value pair (k, v) in table T. In case $k = \bot$, the lookup aborts. The keys are unique and reusing an existing key k overwrites the previous value stored in $\mathrm{T}[k]$. Table entries are assumed to be implicitly initialized to $\bot$, such that the comparison $\mathrm{T}[k] = \bot$ is well-defined (and evaluates to true when no entry for k was stored). Furthermore, every table has the entry $\mathrm{T}[\varepsilon] = \varepsilon$ to map the empty index to an empty entry. We use $k \in \mathrm{T}$ as shorthand for checking whether $\mathrm{T}[k] \neq \bot$. Let $|\mathrm{T}|$ denote the number of key-value pairs in the table. We use a dot (e.g., $\mathrm{T}[(k, \cdot)]$) to denote a wildcard matching any element (here, any tuple where element one is k).

We use the dot notation from object-oriented programming to access specific elements of a tuple when the context is clear. For instance, to store a password pw and an account ID aid in a state st, we write $st \leftarrow (pw, aid)$ and access these values directly using $st.pw$ and $st.aid$. Furthermore, we use this notation to set values, e.g., $st.sid \leftarrow sid$ appends a session ID sid to st. such that it now stores (pw, aid, sid).

Games. We use the code-based game-playing framework of [11]. By $\mathbf{G}_{\mathsf{S}}^{\text{Sec-}b}(\mathcal{A})$, we denote running the game for security experiment Sec, parameterized by scheme S and (optionally) bit $b \in \{0,1\}$, with adversary $\mathcal{A}$. We denote the probability that the execution of game $\mathbf{G}$ results in the output y as $\Pr[\mathbf{G}(\mathcal{A}) \Rightarrow y]$, and write just $\Pr[\mathbf{G}]$ for $\Pr[\mathbf{G}(\mathcal{A}) \Rightarrow 1]$ and $\Pr[\mathbf{G}(\mathcal{A}) \Rightarrow \mathsf{true}]$. Our proofs are in the random oracle model [10]. In games, this is modeled via an oracle $\mathrm{RO}_i(v)$, which implements lazy sampling for some random oracle H_i on input v.

In games, integer variables, set variables boolean variables and string variables are assumed initialized, respectively, to 0, the empty set $\emptyset$, the boolean false and ε. We use **global** to mark variables with global scope, i.e., accessible from all parts of the game, including oracles. The symbol ι_{mal} is reserved for tracking actions by the adversary and is assumed not to be in $\mathbb{N}$.

3 A Framework for Cloud Storage

Our framework for cloud storage systems identifies seven core operations:

Account registration (areg). Register an account with the cloud storage server.

Session authentication (auth). Authenticate registered user and create a new (client) session to interact with the server over the following operations.

File upload (put). Upload a file to the server.

File update (upd). Replace an existing file on the server.

File download (get). Retrieve a file from the server.

File share (share). Prepare sharing an existing file with another user.

Accept share (accept). Accept the sharing of a file by another user.

Each of these operations generally involves an interaction between a client (session) and the cloud storage server, which we will capture as interactive, message-based protocols. In our framework, we explicitly treat the operations described above (and their expected security properties), but the notation is generic and can be extended to cover further operations one might be interested in when capturing systems that allow for, e.g., account deregistration, session deauthentication, or password recovery mechanisms. In the following, we will first introduce a generic syntax that allows us to capture all of the above protocols in a unified manner, then discuss the specific behavior and inputs of each protocol.

3.1 Syntax

Syntactically, we consider a cloud storage scheme $\mathsf{CS} = (\Pi_{\mathsf{areg}}, \Pi_{\mathsf{auth}}, \Pi_{\mathsf{put}}, \Pi_{\mathsf{upd}}, \Pi_{\mathsf{get}}, \Pi_{\mathsf{share}}, \Pi_{\mathsf{accept}})$ to consist of the seven core protocols outlined above and described in detail in Section 3.2; see also Table 2 for their minimal input-output behavior. Each of these is a two-party interactive protocol executed between a client and a server, the formalities of which we define next.

<u>PROTOCOL STEPS.</u> The execution of protocol Π involves $\Pi.\mathsf{SN}$ protocol steps overall, alternating between client and server. We denote each round of the interaction as $\Pi^{(C:i)}$ and $\Pi^{(S:i)}$, for step i of client and server, respectively. Syntactically, we write the i-th protocol step of a party $P \in \{C, S\}$ as

$$(st_P, st_P^{\mathrm{tmp}}, out_P, m_{out}) \leftarrow\!\$\ \Pi^{(P:i)}(st_P, st_P^{\mathrm{tmp}}, [in_P,]\, m_{in}),$$

where st_P and st_P^{tmp} is the local (persistent resp. temporary/protocol) state of party P, in_P is the local input [present only in the first step, $i = 1$], m_{in} and m_{out} is the incoming resp. outgoing message of the protocol,[1] and out_P is the local output of party P. As a convention, we set empty protocol inputs to ε.

For a first, rough example, a file download protocol Π_{get} might, in step $\Pi^{(C:1)}$, take a file identifier *fid* as client-side input in_C and send that identifier to the server as part its output message m_{out}. The server step $\Pi^{(S:1)}$ then receives *fid* as message input m_{in}, might use it to look up the corresponding file f in its persistent state st_S, and send f back within its output message. In step $\Pi^{(C:2)}$, the client would then receive the file f from the server in the input message m_{in} and output that file to the user within out_C.

We use the convention that the client sends the first message and the server sends the last message. This restricts the syntax to client-initiated protocols, but we argue that it suffices to model all operations relevant for the cryptographic core of a cloud storage protocol. For example, our syntax cannot capture "pushing notifications" from the server to clients, say to inform the laptop of a user that their phone just uploaded a new file. However, in most cases, such functionality can be handled by the surrounding, non-cryptographic protocol: the cryptographic core functionality can simply assume that it is known which files can be requested. Alternatively, if necessary for security, a server-initiated operation can be modeled in our syntax by setting the first client message to the empty message (effectively turning a "push" mechanism into "polling").

<u>FULL EXECUTION.</u> We write the full execution of the protocol Π on initial states st_C, st_S and inputs in_C, in_S, of client and server respectively,[2] as

$$(\mathbf{m}, st'_C, st'_S, out_C, out_S) \leftarrow\!\$\ \mathbf{Exec}_{\Pi}(st_C, st_S, in_C, in_S).$$

[1]The input message m_{in} is empty, i.e., $m_{in} = \varepsilon$, for the initiating party in step $i = 1$.

[2]Note that the inputs in_C and in_S are only arguments to the first step of their respective protocols. We discuss the reasons behind this choice and its implications in the general discussion of our cloud storage framework, Section 4.4.

```
$\mathbf{Exec}_{\Pi}(st_C, st_S, in_C, in_S)$:
$(st_C, st_C^{\mathrm{tmp}}, out_C, \mathbf{m}[1]) \leftarrow\!\!\$\ \Pi^{(C:1)}(st_C, \varepsilon, in_C, \varepsilon)$
$(st_S, st_S^{\mathrm{tmp}}, out_S, \mathbf{m}[2]) \leftarrow\!\!\$\ \Pi^{(S:1)}(st_S, \varepsilon, in_S, \mathbf{m}[1])$
for $i = 3, \ldots, \Pi.\mathsf{SN}$:
  $r \leftarrow \lceil i/2 \rceil$ ; if $i$ odd: $P \leftarrow C$ else $P \leftarrow S$
  if $out_P.dec = \bot$: $(st_P, st_P^{\mathrm{tmp}}, out_P, \mathbf{m}[i]) \leftarrow\!\!\$\ \Pi^{(P:r)}(st_P, st_P^{\mathrm{tmp}}, \mathbf{m}[i-1])$
return $(\mathbf{m}, st_C, st_S, out_C, out_S)$
```

Figure 1: Execution $\mathbf{Exec}_{\Pi}$ of a protocol Π between client C and server S.

Protocol		**Inputs** in_C	**Outputs** out_C
Account registration	Π_{areg}	*aid*, *pw*	—
Session authentication	Π_{auth}	*aid*, *pw*	— (st_C created)
File upload	Π_{put}	*fid*, *f*	—
File update	Π_{upd}	*fid*, *f*	—
File download	Π_{get}	*fid*	*f*
File share	Π_{share}	*raid*, *fid*	*oob*
Accept share	Π_{accept}	*fid*, *oob*	—

Table 2: Core protocols making up a cloud storage scheme CS, along with their mandatory *client-side* inputs and outputs; see the detailed explanation in Section 3.2.

The output of the execution is a transcript $\mathbf{m}$ of the interaction, which is the sequence of messages exchanged, the new persistent states st'_C, st'_S of the parties, and their respective outputs out_C, out_S. As part of the output, the decision of party $P \in \{C, S\}$ to accept or reject is encoded as $out_P.dec \in \{\mathsf{true}, \mathsf{false}\}$.

We specify the details of $\mathbf{Exec}_{\Pi}$ in Figure 1: it calls the protocol for each party and relays messages as long as parties have not finalized their decisions $out_P.dec$. We will use this full (or "atomic") execution of a protocol Π to define correctness, where we assume that operations are executed honestly and in sequential order (per client session). Later, when defining security, we allow the adversary to arbitrarily and maliciously interleave protocol steps.

Termination. For more concise return statements in standard success or failure cases, we introduce shorthands $\mathbf{success}_P$ and $\mathbf{fail}_P$ for $P \in \{C, S\}$ defined as

$$\mathbf{success}_P(m_P)\colon\ out_P.dec \leftarrow \mathsf{true};\ \ \mathbf{return}(st_P, st_P^{\mathrm{tmp}}, out_P, m_P),$$

$$\mathbf{fail}_P\colon\ out_P.dec \leftarrow \mathsf{false};\ \ \mathbf{return}(st_P, st_P^{\mathrm{tmp}}, out_P, \bot),$$

where by convention $\mathbf{success}_P := \mathbf{success}_P(\top)$ returns the success symbol $\top$.

Failing procedures and operations – including $\mathbf{fail}_P$, calls returning the error symbol $\bot$, operations on $\bot$, and using index $\bot$ for tables – propagate this failure through calling procedures (comparable to program exceptions), causing the caller of the failing procedure to return $\mathbf{fail}_P$.

3.2 The Protocols

Let us now describe the formalized core protocols of a cloud storage scheme CS in more detail. Table 2 lists them along with their mandatory inputs and outputs.

Registration and authentication. The account registration protocol Π_{areg} creates a new user in the system, represented by an account ID (e.g., an email address) provided by the client. It expects that account ID *aid* as well as the user's password *pw* as (minimal) client-side input within in_C and must be run before any other protocols can be run on behalf of that user.[3] The account registration protocol does not involve any persistent client session state yet (i.e., $st_C = st'_C = \varepsilon$), as such state pertains only to active sessions.

A registered user can then run the authentication protocol Π_{auth} to authenticate to the server and initiate a new client session, again using account ID *aid* and password *pw* as client-side input, and upon empty client state $st_C = \varepsilon$. This creates a persistent client session state st'_C, which is shared among all protocols run within that session. A user can initiate multiple client sessions in parallel (each holding their own state st_C), which can concurrently access the user's files in the cloud storage.

All remaining protocols operating on files can only be called with non-empty client state st_C (i.e., we implicitly require them to fail otherwise). Overall, this enforces that account registration must be run before authentication, and authentication must be run before any file operation protocol.

File up- and download. Within an active session, a client can upload a new file using Π_{put} on input a (globally unique) file identifier *fid* and file content f. File retrieval is correspondingly done via Π_{get} on (client-side) input the file's identifier *fid*.[4] We handle file updates and deletions through the Π_{upd} protocol, taking as client-side input an existing identifier *fid* and the updated file content f. File deletion can be interpreted as setting the file f in the client input to $\perp$.

File sharing. We capture file sharing through two operations, Π_{share} and Π_{accept}. The sharing protocol Π_{share} allows a client to prepare sharing a file identified by *fid*,[5] and notify the server about its intent to share said file with another user, identified by the receiver account ID *raid*. To capture that means of access (e.g., key material) might be exchanged between sender and receiver of a shared file through some out-of-band channel (e.g., messaging a sharing link), we allow the sharing protocol to output some dedicated out-of-band information $out_C.oob$. Apart from the OOB channel, there is no other client-to-client communication.

To accept the sharing of a file, the receiver client session runs Π_{accept} with the server. On input the file identifier *fid* of the shared file and the obtained out-of-band information *oob*, the Π_{accept} operation integrates the key material for the shared file into this client's cloud storage for future access (through Π_{get}, Π_{upd}, or Π_{share}). In our notion, sharer and sharees all have full access to the *same* file with no distinction between the original file owner and share recipients.

While, in syntax, we do not put constraints on the out-of-band information, in practice, such out-of-band channels are highly restricted and hence realistic schemes will have to restrict their use of it.[6] In our proposed cloud storage scheme in Section 6, we use it only to exchange keys that are a few bytes long (embedded, e.g., in a URL or QR code) for providing access to shared files.

[3] We list here only the *minimal* required client-side inputs to protocols, needed to define basic functionality as well as security. A concrete cloud storage scheme might take further (client- or server-side) inputs.

[4] Clients can learn *fid* from communication outside of our modeled core protocols, e.g., by fetching a mapping from file names to file identifiers from the server.

[5] Some cloud storage protocols may use account-dependent globally unique file identifiers (e.g., (aid, n) where n is a monotonic counter). In such a protocol, sharing files with user *raid* may require changing the file identifier (in the previous example, to $(raid, n')$). This can still be modeled in with our framework, but the protocol description needs to introduce a level of indirection and map account-dependent identifiers to account-independent ones that can be used by multiple users simultaneously, and then translate them depending on context.

[6] There may be trivially secure schemes that make heavy use of the out-of-band channel, e.g., by using the channel to entirely bypass the service provider. While such protocols might match our definitions, they are not interesting in practice as they do not adhere to said real-world constraints.

There is no dedicated mechanism to revoke access to a shared file from a user. However, since all users share access to the same file, any one of them can delete and re-upload the file to create a version of it which is not shared. It is possible that by explicitly modeling revocation, one could capture further security guarantees associated to access rights management. For example, in the honest-server setting, it might be possible to ensure that a user who gets their access to a file revoked cannot download that file in the future, nor see updates or new versions of the file. We discuss extensions such as these to the functionality and security of our model in Section 4.4.

3.3 Correctness

We now define the expected functional behavior, i.e., correctness, of a cloud storage scheme CS in executions of honestly behaving clients and server. Due to the potential interleavings between protocol executions of different client sessions, we do this via the game $\mathbf{G}^{\mathrm{corr}}_{\mathsf{CS}}$ given in Figure 2, with oracles defined in Figure 3.

The intuition of the correctness game is that if a user successfully uploaded a file to the server, or if a file was shared with them, then fetching that file should succeed and return the expected file content. Put differently, if the server behaves honestly (and in particular does not perform any availability attacks), then users can successfully recover their cloud storage files. Formally, this is captured by letting the adversary win the game (by setting a win flag) if a Π_{get} operation does not return the expected file content (see Figure 3, line 20) or if a Π_{accept} operation fails even though the file has been shared with the receiving user (Figure 3, line 35).

Definition 3.1 (Cloud storage correctness) *Let CS be a cloud storage scheme and $\mathbf{G}^{\mathrm{corr}}_{\mathsf{CS}}$ be the correctness game for CS defined in Figures 2 and 3. We define the advantage of an adversary $\mathcal{A}$ playing this game as*

$$\mathbf{Adv}^{\mathrm{corr}}_{\mathsf{CS}}(\mathcal{A}) := \Pr\left[\,\mathbf{G}^{\mathrm{corr}}_{\mathsf{CS}}(\mathcal{A})\,\right],$$

and say that CS is correct if $\mathbf{Adv}^{\mathrm{corr}}_{\mathsf{CS}}(\mathcal{A}) = 0$ for all (even unbounded) $\mathcal{A}$.

```
Game $\mathbf{G}^{\mathrm{corr}}_{\mathsf{CS}}(\mathcal{A})$:
1  global $\mathrm{S}_A, n_c, st_S, \mathrm{T}_{\mathrm{stC}}, \mathrm{T}_{\mathrm{f}}, \mathrm{T}_{\mathrm{oob}}, \mathrm{S}_{\mathrm{shr}}, \mathsf{win}$
2  $\mathsf{win} \leftarrow \mathsf{false}$ ; $n_c \leftarrow 0$
3  $\mathcal{A}^{\text{AReg},\text{Auth},\text{Put},\text{Upd},\text{Get},\text{Share},\text{Acpt}}()$
4  return $\mathsf{win}$

$\mathsf{RunProt}(\Pi, i_c, in_C, in_S)$:
5  pre: $(\mathrm{T}_{\mathrm{stC}}[i_c] \neq \bot)$ or $in_C.\{aid\}$
6  if $\mathrm{T}_{\mathrm{stC}}[i_c] \neq \bot$: $(aid, st_C) \leftarrow \mathrm{T}_{\mathrm{stC}}[i_c]$   // Load client session state for Put, Get, Share, Acpt
7  else: $st_C \leftarrow \varepsilon$   // No client session state for AReg, Auth
8  $(\mathbf{m}, st_C, st_S, out_C, out_S) \leftarrow\$ \mathbf{Exec}_{\Pi}(st_C, st_S, in_C, in_S)$
9  if $(out_C.\mathrm{dec} \wedge out_S.\mathrm{dec})$:
10     if $\Pi = \Pi_{\mathsf{areg}}$: $\mathrm{S}_A \xleftarrow{\cup} \{aid\}$   // Register account upon successful AReg
11     if $\Pi = \Pi_{\mathsf{auth}}$: $\mathrm{T}_{\mathrm{stC}}[i_c] \leftarrow (aid, st_C)$   // Register client session state upon successful Auth
12 if $\Pi \notin \{\Pi_{\mathsf{areg}}, \Pi_{\mathsf{auth}}\}$: $\mathrm{T}_{\mathrm{stC}}[i_c] \leftarrow (aid, st_C)$
13 return $(\mathbf{m}, st_C, st_S, out_C, out_S)$
```

Figure 2: Correctness game $\mathbf{G}^{\mathrm{corr}}_{\mathsf{CS}}$ for cloud storage scheme CS, with oracles given in Figure 3, and helper function $\mathsf{RunProt}$ used by oracles.

```
$\underline{\text{AREG}(in_C, in_S)}$:
    pre: $in_C.aid \notin \mathrm{S}_A$, $in_C.\{aid, pw\}$
    return $\mathsf{RunProt}(\Pi_{\mathsf{areg}}, \perp, in_C, in_S)$

$\underline{\text{AUTH}(in_C, in_S)}$:
    pre: $in_C.aid \in \mathrm{S}_A$, $in_C.\{aid, pw\}$
    $n_c \leftarrow n_c + 1$   // increment client session counter
    return $\mathsf{RunProt}(\Pi_{\mathsf{auth}}, n_c, in_C, in_S)$

$\underline{\text{PUT}(i_c, in_C, in_S)}$:
    pre: $i_c \in \mathrm{T}_{\mathrm{stC}}$, $in_C.\{fid, f\}$
    if $fid \in \mathrm{T}_{\mathrm{f}}$: return $\perp$   // not allowed to upload same file id twice
    $(\mathbf{m}, st_C, st_S, out_C, out_S) \leftarrow \mathsf{RunProt}(\Pi_{\mathsf{put}}, i_c, in_C, in_S)$
    if $out_S.dec$:
        $ow \leftarrow \{\mathrm{T}_{\mathrm{stC}}[i_c].aid\}$; $\mathrm{T}_{\mathrm{f}}[fid] \leftarrow (ow, f)$   // store owner and file in file table
    return $(\mathbf{m}, st_C, st_S, out_C, out_S)$

$\underline{\text{UPD}(i_c, in_C, in_S)}$:
    pre: $i_c \in \mathrm{T}_{\mathrm{stC}}$, $in_C.\{fid, f\}$
    if $\mathrm{T}_{\mathrm{stC}}[i_c].aid \notin \mathrm{T}_{\mathrm{f}}[fid].ow$: return $\perp$   // only allow updated by file owners
    $(\mathbf{m}, st_C, st_S, out_C, out_S) \leftarrow \mathsf{RunProt}(\Pi_{\mathsf{upd}}, i_c, in_C, in_S)$
    if $out_S.dec$: $\mathrm{T}_{\mathrm{f}}[fid].f \leftarrow f$   // update file in file table
    return $(\mathbf{m}, st_C, st_S, out_C, out_S)$

$\underline{\text{GET}(i_c, in_C, in_S)}$:
    pre: $i_c \in \mathrm{T}_{\mathrm{stC}}$, $in_C.\{fid\}$
    if $\mathrm{T}_{\mathrm{stC}}[i_c].aid \notin \mathrm{T}_{\mathrm{f}}[fid].ow$: return $\perp$   // only allow downloads by file owners
    $(\mathbf{m}, st_C, st_S, out_C, out_S) \leftarrow \mathsf{RunProt}(\Pi_{\mathsf{get}}, i_c, in_C, in_S)$
    if $(out_C.dec$ and $out_C.f \neq \mathrm{T}_{\mathrm{f}}[fid].f)$: $\mathsf{win} \leftarrow \mathsf{true}$   // retrieved unexpected file
    if $(\neg out_C.dec$ and $fid \in \mathrm{T}_{\mathrm{f}}[fid])$: $\mathsf{win} \leftarrow \mathsf{true}$   // fail to retrieve uploaded file
    return $(\mathbf{m}, st_C, st_S, out_C, out_S)$

$\underline{\text{SHARE}(i_c, in_C, in_S)}$:
    pre: $i_c \in \mathrm{T}_{\mathrm{stC}}$, $in_C.\{raid, fid\}$
    if $\mathrm{T}_{\mathrm{stC}}[i_c].aid \notin \mathrm{T}_{\mathrm{f}}[fid].ow$: return $\perp$   // only allow sharing by file owners
    $(\mathbf{m}, st_C, st_S, out_C, out_S) \leftarrow \mathsf{RunProt}(\Pi_{\mathsf{share}}, i_c, in_C, in_S)$
    if $out_C.dec$:
        $\mathrm{S}_{\mathrm{shr}} \overset{\cup}{\leftarrow} \{(raid, fid)\}$; $\mathrm{T}_{\mathrm{oob}}[(raid, fid)] \leftarrow out_C.oob$   // store OOB info for receiver
    return $(\mathbf{m}, st_C, st_S, out_C, out_S)$

$\underline{\text{ACPT}(i_c, in_C, in_S)}$:
    pre: $i_c \in \mathrm{T}_{\mathrm{stC}}$, $in_C.\{fid\}$
    if $(\mathrm{T}_{\mathrm{oob}}[(\mathrm{T}_{\mathrm{stC}}[i_c].aid, fid)] = \perp)$: return $\perp$   // only proceed if out-of-band info available
    $(in_C.oob) \leftarrow \mathrm{T}_{\mathrm{oob}}[(\mathrm{T}_{\mathrm{stC}}[i_c].aid, fid)]$   // retrieve out-of-band info for receiver
    $(\mathbf{m}, st_C, st_S, out_C, out_S) \leftarrow \mathsf{RunProt}(\Pi_{\mathsf{accept}}, i_c, in_C, in_S)$
    if $out_S.dec$: $\mathrm{T}_{\mathrm{f}}[fid].ow \overset{\cup}{\leftarrow} \{\mathrm{T}_{\mathrm{stC}}[i_c].aid\}$   // mark receiver as file owner
    if $out_C.dec$: $\mathrm{T}_{\mathrm{oob}}[(\mathrm{T}_{\mathrm{stC}}[i_c].aid, fid)] \leftarrow \perp$   // remove out-of-band info for receiver
    if $\neg out_C.dec \wedge (\mathrm{T}_{\mathrm{stC}}[i_c].aid, fid) \in \mathrm{S}_{\mathrm{shr}}$: $\mathsf{win} \leftarrow \mathsf{true}$   // fail to accept shared file
    return $(\mathbf{m}, st_C, st_S, out_C, out_S)$
```

Figure 3: Oracles for the correctness game $\mathbf{G}^{\mathrm{corr}}_{\mathsf{CS}}$ for cloud storage scheme CS. The helper function $\mathsf{RunProt}$, given in Figure 2, captures core execution steps the protocols.

Let us explain some further technical details of the correctness game.

Atomic execution. Recall that, for simplicity, correctness is defined w.r.t. "atomic" protocol executions where individual protocol steps are executed sequentially and cannot be interleaved. This simplifies tracking of variables and avoids handling race conditions (e.g., which file version should be expected to be downloaded after two concurrent file uploads using the same file id?). We emphasize that when defining security later, we capture arbitrary and malicious interleaving of individual protocol steps.

Focus on core functionality. Our correctness definition only captures the core functionality of a cloud storage scheme: a successfully uploaded or shared file should be retrievable by authorized clients. In particular, correctness does *not* demand that uploading or sharing operations have to be successful. This is intentional: it allows for different instantiations of these protocols. For example, a scheme might reject files that are too large or whose file identifiers are malformatted – but we do not want to prescribe such rules for a generic scheme. As another example, one scheme may allow sharing a file only with existing accounts while another might allow sharing with non-existing accounts (e.g., to invite a new user to the system via their not-yet-registered email address) – hence, we do not prescribe when sharing ought to be successful. We note that this opens our correctness notion up to "correct" dummy schemes (e.g., where uploading or sharing is never successful); in practice such schemes are trivially uninteresting, and we accept this trade-off in favor of being able to express a rich variety of cloud storage schemes in our framework.

Preconditions and other notation. We annotate game oracles with preconditions (written **pre**: ...) which specify the minimal set of input values required by the game for the oracle call of some protocol to be meaningful. An oracle may for example specify **pre**: $in_C.\{fid, f\}$ to indicate that it requires the values $in_C.fid$ and $in_C.f$ to be set. For brevity, we refer to the values required in a precondition without the prefix in_C inside the code of the oracle (in the above example: fid instead of $in_C.fid$). Unsatisfied preconditions cause the oracle to abort.

Accounts and clients. Users are identified via their account ID aid, set as client input by the adversary in the oracles AReg for account registration and Auth for session authentication. The game keeps a count of all authenticated client sessions; whenever authentication succeeds in Auth on input account ID aid, it associates a new client session counter value i_c with aid and the session's state st_C (see Figure 2, line 11).

Global game variables. The correctness game tracks several global variables across oracle calls. This includes the set S_A of all IDs for registered accounts, the counter n_c for client sessions, the server state st_S, and the table T_{stC} which tracks client session states. Uploaded files and their owners are tracked using table T_{f}, which is indexed via the (globally unique) file identifiers fid specified by the adversary. The game enforces globally unique file identifiers that are never repurposed by never resetting the owner set of a file (even when a file is "deleted" by overwriting it with $\perp$). Out-of-band information exchanged between the sender and receiver of file shares is tracked using table T_{oob}.

The game oracles update most of these global variables only if operations succeed (indicated by $out_C.dec$ and $out_S.dec$ being true), encoding that most correctness guarantees only apply after a successful protocol run. For instance, the oracle Π_{put} updates the file table T_{f} after the protocol run succeeds on the server (see Figure 3, line 9). However, we do not require that both parties always succeed. Indeed, for all file-related protocols, we update the client session states in T_{stC} on line 12 of Figure 2 irrespective of the output decisions. This captures that even a correct scheme might be expected to fail, for instance, when trying to upload a file with a malformed file ID, but that in case of such a protocol failure, the scheme should still continue to function. (That is, the failed operation should not lead to a corrupted state on either client or server.) In contrast, for registration and authentication, line 9 of Figure 2 ensures that $\mathrm{T_{stC}}$ is only updated when both parties succeed, capturing that these protocols can only be correct if both participants agree on their outcome. This is necessary, as later operations for the same user or in the same session rely on registration and authentication, respectively, to have been successful.

4 Client-to-Client Security of Cloud Storage

In our cloud storage framework, we are interested in guaranteeing end-to-end encryption security for the data of clients. We call this *client-to-client* (C2C) security. The aim is to protect honest clients against a fully malicious, adversarially-controlled cloud provider, giving both *confidentiality* (C2CConf) and *integrity* (C2CInt) guarantees. Our security model focuses on the operations of the cloud storage system itself. We hence assume clients communicate with the storage server over a server-authenticated and confidential channel (e.g., a TLS connection) and do not concern ourselves with capturing the setup or concrete security of that channel. We point out subtle reasons for further modeling choices where they arise, and expand on them in the discussion in Section 4.4. There, we also discuss potential extensions to our security model, including additional protection against weaker adversaries that do not control the cloud server, as well as the modeling of metadata. More future work is mentioned in Section 7.

<u>PASSWORD MODELING.</u> Users in our cloud storage framework are authenticated through passwords (see Section 3). All of our security games are parameterized by $\mathcal{PW}_n$, a distribution of password vectors of cardinality n (the maximal number of user accounts), and pre-sample a password vector $\mathbf{pw} \overset{\vec{r}}{\leftarrow} \mathcal{PW}_n$ from this distribution. Security is then defined with respect to the distribution $\mathcal{PW}_n$. This approach is inspired by prior game-based treatments of password-based encryption [9, 12], modeling non-uniform password distributions and correlated or even identical passwords in game-based proofs – i.e., the choices users make when picking passwords.

Previous work [9, 12] assumed that $\mathbf{pw}$ consists of unique passwords to “avoid trivial attacks.” We do not make this assumption, as we deem it impossible to enforce in practice even for an honest service provider. Instead, we assume that the *account IDs aid* are globally unique for honest users. This can be enforced in practice, e.g. by taking email addresses as account IDs, possibly verified via a token exchange. We hence demand that the cloud storage scheme achieves security even when multiple users have the same password. Note that we do assume that the password distribution is independent from the hash functions used to hash passwords in our proposed cloud storage scheme. This is required for our cloud storage proofs, as the existence of pathological distributions otherwise make the derivation of pseudorandom keys from passwords impossible.

Security obviously depends on the strength of the passwords chosen by users (as modeled by the distribution $\mathcal{PW}_n$). More specifically, it depends on the chances of an adversary outright *guessing* a user's password. Following [9, 12], we are then interested in the security of a cloud storage scheme beyond the “guessing probability” of passwords from the given distribution. Naturally, a heavily skewed password distribution where guessing user passwords is easy leads to poor security guarantees for such users; real-world systems (e.g., Nextcloud [48] and OnePassword [49]) employ according mitigations to assist users in picking secure passwords.

We capture the password-guessing probability through the game in Figure 4, where the adversary, given selective access to the passwords of compromised users, wins by guessing the password of an uncorrupted user.

Definition 4.1 (Password guessing) *Let* $\mathbf{G}^{\mathrm{PG}}_{n,\mathcal{PW}_n}$ *be the password-guessing game defined in Fig-*

```
Game $\mathbf{G}^{\mathrm{PG}}_{n,\mathcal{PW}_n}(\mathcal{A})$:
1 global $\mathrm{S}_{\mathsf{comp}}$, $\mathbf{pw}$, win
2 $\mathbf{pw} \xleftarrow{\nabla} \mathcal{PW}_n$
3 $\mathrm{S}_{\mathsf{comp}} \leftarrow\!\!\$\ \mathcal{A}()$
4 for $i_u \in \mathrm{S}_{\mathsf{comp}}$: $\mathbf{pw}_{\mathsf{comp}}[i_u] \leftarrow \mathbf{pw}[i_u]$
5 $\mathcal{A}^{\textsc{Test}}(\mathbf{pw}_{\mathsf{comp}})$
6 return win

$\textsc{Test}(i_u, pw)$:
7 if $(pw = \mathbf{pw}[i_u]) \wedge (i_u \notin \mathrm{S}_{\mathsf{comp}})$:
8     win $\leftarrow$ true
```

Figure 4: Password guessing game with selective compromises against a password distribution $\mathcal{PW}_n$ and at most n users.

---

ure 4, for at most n *users and with respect to a password distribution* $\mathcal{PW}_n$*. We define the password-guessing advantage of an adversary* $\mathcal{A}$ *playing this game as*

$$\mathbf{Adv}^{\mathrm{PG}}_{n,\mathcal{PW}_n}(\mathcal{A}) := \Pr\left[\mathbf{G}^{\mathrm{PG}}_{n,\mathcal{PW}_n}\right].$$

<u>Adversarial interaction.</u> Before introducing our confidentiality and integrity notions, we explain some common features of our games with respect to the interaction with the adversary. The first surrounds the fine-grained level at which we capture the operation of the cloud storage scheme: As mentioned in earlier sections, the protocols comprising a cloud storage scheme may consist of several, interactive steps. For instance, the file update protocol Π_{upd} might first fetch some information about the file to be updated from the storage server before uploading the updated file content. In practice, a dishonest server might maliciously interleave the executions of these protocol steps to its benefit. Our cloud storage security games hence allow step-by-step execution of the protocols, with arbitrary, adversary-controlled interleaving across protocols.

In the games (see Figures 5 and 6), this multi-step nature of protocols is captured by providing the adversary with oracles to initiate each protocol run (AReg1, Auth1, etc.), which check potential preconditions, and a "stepping" oracle Step through which the adversary can trigger the next client-side step of a protocol (identified by some process ID i_p). The games use the table $\mathrm{T_p}$ to keep track of the state of each ongoing protocol execution, including the participants' state.

On the client side, the adversary gets to pick relevant inputs (e.g., account and file identifiers, file contents) for honest clients. The server instead is embodied by the adversary, capturing the C2C guarantees against malicious cloud providers. In particular, this means that the adversary receives the honest clients' messages and controls the server's responses in each protocol step.

4.1 Client-to-Client Confidentiality

We first define client-to-client confidentiality, via the game $\mathbf{G}^{\mathrm{C2CConf}\boxed{\mathrm{S}}}_{\mathsf{CS},n,\mathcal{PW}_n}$ given in the top panel of Figure 5, together with oracles and helper functions in Figures 5 and 6. (Our integrity game is defined through the same figures, as it heavily overlaps in the basic structure; it contains the code lines in $\boxed{\text{dashed}}$ boxes, which are not present in the confidentiality game. These can hence safely be ignored for now.) The game is parameterized by the maximum number of user accounts n and a distribution $\mathcal{PW}_n$ of password vectors of the same cardinality. We further distinguish two versions of the game, depending on whether the adversary is allowed to adaptively (C2CConf) or selectively (C2CConfS) compromise clients; the latter version includes the code in $\boxed{\text{dotted}}$ boxes.

In brief, confidentiality is captured as left-or-right indistinguishability[7] (akin to classical IND-CCA security for encryption): The game samples a random challenge bit b and provides the adversary $\mathcal{A}$ with a special challenge oracle CHALL which uploads f_b out of a pair of files (f_0, f_1), via a put or upd operation of an adversary-selected client. The adversary's task is to determine b, while acting as the malicious server and interacting with honest clients through the game oracles. To prevent trivial attacks, the adversary cannot compromise any user that owns a challenge file. We first state the formal security definition before we explain the game in more detail.

Definition 4.2 (Client-to-client confidentiality) *Let* CS *be a cloud storage scheme and let* $\mathbf{G}^{\text{C2CConf}\boxed{S}}_{\mathsf{CS},n,\mathcal{PW}_n}$ *be the adaptive* ($\boxed{\textit{selective}}$) client-to-client confidentiality (C2CConf$\boxed{S}$) *game for* CS *defined in Figures 5 and 6, for at most n honest users and with respect to a password distribution $\mathcal{PW}_n$. We define the advantage of an adversary $\mathcal{A}$ playing this game as*

$$\mathbf{Adv}^{\text{C2CConf}\boxed{S}}_{\mathsf{CS},n,\mathcal{PW}_n}(\mathcal{A}) := 2 \cdot \Pr\left[\mathbf{G}^{\text{C2CConf}\boxed{S}}_{\mathsf{CS},n,\mathcal{PW}_n}\right] - 1.$$

<u>ORACLES.</u> In game $\mathbf{G}^{\text{C2CConf}\boxed{S}}_{\mathsf{CS},n,\mathcal{PW}_n}$, the adversary $\mathcal{A}$ has access to the following oracles: one oracle of the form ORC1 for each CS protocol (used to initiate a protocol run), the oracle STEP to perform the next client step in a protocol, the challenge oracle CHALL for uploading a challenge file, and the oracle COMP to compromise (the entire knowledge) of a user. As in the correctness game (see Section 3.3), we annotate oracles with preconditions where, e.g., **pre**: $in_C.\{fid\}$ indicates that $in_C.fid$ must be set.

Oracles of the form ORC1 capture basic protocol operations. Each of them simulates an honest client (session) initiating the corresponding CS protocol. The game tracks the honest clients' accounts, passwords, as well as persistent session state (st_C) and temporary, protocol-internal state (st_C^{tmp}). Since the adversary plays the role of the (malicious) server, the game does not keep any server state, but only tracks client-related information relevant for checking trivial winning conditions, e.g., which files an honest user has uploaded or shared.

In more detail, the basic oracles work as follows:

- AREG1 initiates registering a new, honest account for a user with index i_u and account ID *aid*, setting the user password to $\mathbf{pw}[i_u]$ (hidden from $\mathcal{A}$). The game allows repeated account registration attempts of honest clients, but enforces that the user index to account ID mapping is bijective.
- AUTH1 initiates a new session i_c of an honest user with given user index i_u, tracking the new session's user index in $\mathrm{T_u}[i_c]$ and persistent session state in $\mathrm{T_{stC}}[i_c]$.
- PUT1 and UPD1 let session i_c initiate an upload resp. update of a file f under ID *fid*, both adversary-chosen. The precondition of PUT1 enforces unique file IDs *fid*, which can be achieved in practice by prefixing file IDs by *aid*. For a newly put file, the user ID of the calling client is tracked as an owner in the set $\mathrm{T_f}[fid].\mathrm{U}$.
- GET1 lets session i_c initiate fetching a file with given ID *fid*. Successfully downloaded files are handed to $\mathcal{A}$ in the helper function Check (serving as a decryption oracle), except for challenge files (to prevent trivial wins).
- SHARE1 and ACPT1 let session i_c initiate sharing a file resp. accepting a shared file. The game tracks the owners of shared files in $\mathrm{T_f}[fid].\mathrm{U}$, and of challenge files in $\mathrm{S_{challOwners}}$, and keeps track of the out-of-band information in $\mathrm{T_{oob}}$. Note that to prevent trivial attacks, the

[7] Our framework does not single out "ciphertext" outputs, leaving the cloud storage scheme maximal freedom in how it implements E2EE. Consequently left-or-right indistinguishability is the appropriate confidentiality notion, as an IND\$-style notion would require such a distinguished ciphertext (in order to replace it by random).

```
Game $\boxed{\mathbf{G}^{\text{C2CConf}\boxed{\text{S}}}_{\mathsf{CS},n,\mathcal{PW}_n}(\mathcal{A})}$:
$\mathbf{global}\ \mathrm{T_{uaid}},\ \mathrm{T_u},\ \mathrm{T_{stC}},\ \mathrm{T_p},\ \mathrm{T_{oob}},\ \mathrm{T_f},\ \mathrm{S_{reg}},\ \mathrm{S_{chall}},\ \mathrm{S_{hon}},\ \mathrm{S_{comp}},\ \mathrm{S_{challOwners}},\ \mathbf{pw},\ b$
$\mathbf{pw} \xleftarrow{\vec{\tau}} \mathcal{PW}_n\ ;\ n_c \leftarrow 0\ ;\ n_p \leftarrow 0\ ;\ b \leftarrow_\$ \{0,1\}$
$\boxed{\mathrm{S_{comp}} \leftarrow_\$ \mathcal{A}()}$
$b' \leftarrow_\$ \mathcal{A}^{\textsc{Step},\textsc{AReg1},\textsc{Auth1},\textsc{Put1},\textsc{Upd1},\textsc{Get1},\textsc{Share1},\textsc{Acpt1},\textsc{Chall},\textsc{Comp}}()$
$\mathsf{trivial} \leftarrow \exists i_u \in \mathrm{S_{challOwners}} : i_u \in \mathrm{S_{comp}} \lor i_u \notin \mathrm{S_{hon}}$   // $\exists$ adversary-owned challenge file
$\mathbf{return}\ \neg\mathsf{trivial} \land (b' = b)$
```

```
Game $\boxed{\mathbf{G}^{\text{C2CInt}\boxed{\text{S}}}_{\mathsf{CS},n,\mathcal{PW}_n}(\mathcal{A})}$:
$\mathbf{global}\ \mathrm{T_{uaid}},\ \mathrm{T_u},\ \mathrm{T_{stC}},\ \mathrm{T_p},\ \mathrm{T_{oob}},\ \mathrm{T_f},\ \mathrm{S_{reg}},\ \mathrm{S_{hon}},\ \mathrm{S_{comp}},\ \mathbf{pw},\ \mathsf{win}$
$\mathbf{pw} \xleftarrow{\vec{\tau}} \mathcal{PW}_n\ ;\ n_p \leftarrow 0\ ;\ n_c \leftarrow 0\ ;\ \mathrm{T_f}[\cdot].\mathrm{F} \leftarrow \emptyset\ ;\ \mathrm{T_f}[\cdot].\mathrm{U} \leftarrow \emptyset\ ;\ \mathsf{win} \leftarrow \mathsf{false}$
$\boxed{\mathrm{S_{comp}} \leftarrow_\$ \mathcal{A}()}$
$\mathcal{A}^{\textsc{Step},\textsc{AReg1},\textsc{Auth1},\textsc{Put1},\textsc{Upd1},\textsc{Get1},\textsc{Share1},\textsc{Acpt1},\textsc{Comp}}()$
$\mathbf{return}\ \mathsf{win}$
```

```
$\underline{\mathsf{Step1}(\Pi, i_c, i_r, in_C)}$:
$(\mathrm{T_{stC}}[i_c], st_C^{\mathrm{tmp}}, out_C, m') \leftarrow_\$ \Pi^{(C:1)}(\mathrm{T_{stC}}[i_c], \varepsilon, in_C, \varepsilon)$
$i_p \leftarrow n_p{+}{+}\ ;\ \mathrm{T_p}[(\Pi, i_p)] \leftarrow (i_c, i_r, st_C^{\mathrm{tmp}}, in_C, 2)$
$\mathbf{return}\ \mathsf{Check}(\Pi, i_p, out_C, m')$

$\underline{\mathsf{Check}(\Pi, i_p, out_C, m')}$:
$(i_c, i_r, st_C^{\mathrm{tmp}}, in_C, r) \leftarrow \mathrm{T_p}[(\Pi, i_p)]\ ;\ fid \leftarrow in_C.fid$
$\mathbf{if}\ \Pi = \Pi_{\mathsf{areg}} \land out_C.dec$: $\mathrm{S_{reg}} \xleftarrow{\cup} \{\mathrm{T_u}[i_c]\}$
$\mathbf{if}\ \Pi = \Pi_{\mathsf{get}}$:
    $\boxed{\mathbf{if}\ \mathrm{T_u}[i_c] \in \mathrm{S_{hon}} \setminus \mathrm{S_{comp}}:}$
    $\boxed{\quad \mathsf{win} \leftarrow out_C.dec \land out_C.f \notin \mathrm{T_f}[fid].\mathrm{F} \land \mathrm{T_f}[fid].\mathrm{U} \subseteq \mathrm{S_{hon}} \setminus \mathrm{S_{comp}}}$
    $\boxed{\mathbf{if}\ fid \in \mathrm{S_{chall}}:\ out_C.f \leftarrow \bot}$
    $\mathbf{return}\ (out_C.f, m')$
$\mathbf{if}\ (\Pi = \Pi_{\mathsf{share}}\ \mathbf{and}\ out_C.dec)$: $\mathrm{T_{oob}}[(i_r, fid)] \leftarrow out_C.oob$
$\mathbf{if}\ (\Pi = \Pi_{\mathsf{accept}}\ \mathbf{and}\ out_C.dec)$: $\mathrm{T_{oob}}[(\mathrm{T_u}[i_c], fid)] \leftarrow \bot$
$\mathbf{return}\ m'$

$\underline{\textsc{Step}(\Pi, i_p, m)}$:
$(i_c, st_C^{\mathrm{tmp}}, in_C, r) \leftarrow \mathrm{T_p}[(\Pi, i_p)]$
$\mathbf{if}\ r > \Pi.\mathsf{SN}\ \lor (\Pi = \Pi_{\mathsf{areg}} \land \mathrm{T_u}[i_c] \in \mathrm{S_{reg}})$: $\mathbf{return}\ \bot$
$(\mathrm{T_{stC}}[i_c], st_C^{\mathrm{tmp}}, out_C, m') \leftarrow_\$ \Pi^{(C:r)}(\mathrm{T_{stC}}[i_c], st_C^{\mathrm{tmp}}, m)$
$\mathrm{T_p}[(\Pi, i_p)] \leftarrow (i_c, st_C^{\mathrm{tmp}}, in_C, r+1)$
$\mathbf{return}\ \mathsf{Check}(\Pi, i_p, out_C, m')$
```

Figure 5: Client-to-client confidentiality C2CConf$\boxed{\text{S}}$ (top) and integrity C2CInt$\boxed{\text{S}}$ (middle) games for security against a fully malicious cloud storage server with fine-grained control, as well as helper functions $\mathsf{Step1}$ and Check and oracle $\textsc{Step}$ (bottom). Code only included in the C2CConf$\boxed{\text{S}}$ or C2CInt$\boxed{\text{S}}$ game is marked with $\boxed{\text{boxed}}$ and $\boxed{\text{dashed}}$ boxes, respectively. Moreover, $\boxed{\text{dotted}}$ code is only included in the *selective security* version of the games; the adversary is implicitly assumed to be stateful in that case.

```
$\underline{\text{AReg1}(i_u, in_C)\text{:}}$
$\textbf{pre}\text{: } in_C.\{aid\}, (\nexists i'_u \in \text{T}_{\text{uaid}} \setminus \{i_u\})\text{: } aid = \text{T}_{\text{uaid}}[i'_u], i_u \notin \text{S}_{\text{reg}}$
$in_C.pw \leftarrow \textbf{pw}[i_u]$ ; $\text{S}_{\text{hon}} \xleftarrow{\cup} \{i_u\}$
$\textbf{if } i_u \in \text{T}_{\text{uaid}}\text{: } in_C.aid \leftarrow \text{T}_{\text{uaid}}[i_u] \textbf{ else}\text{: } \text{T}_{\text{uaid}}[i_u] \leftarrow aid$
$\textbf{return } \mathsf{Step1}(\Pi_{\mathsf{areg}}, \varepsilon, \varepsilon, in_C)$

$\underline{\text{Auth1}(i_u, in_C)\text{:}}$
$in_C.aid \leftarrow \text{T}_{\text{uaid}}[i_u]$ ; $in_C.pw \leftarrow \textbf{pw}[i_u]$ ; $i_c \leftarrow n_c{+}{+}$ ; $\text{T}_{\text{u}}[i_c] \leftarrow i_u$ ; $\text{T}_{\text{stC}}[i_c] \leftarrow \varepsilon$
$\textbf{return } \mathsf{Step1}(\Pi_{\mathsf{auth}}, i_c, \varepsilon, in_C)$

$\underline{\text{Put1}(i_c, in_C)\text{:}}$
$\textbf{pre}\text{: } in_C.\{fid, f\}, fid \notin \text{T}_{\text{f}}$
$\text{U} \leftarrow \{\text{T}_{\text{u}}[i_c]\}$ ; $\boxed{\text{F} \leftarrow \{f\}}$
$\text{T}_{\text{f}}[fid] \leftarrow (\text{U}, \text{F})$
$\textbf{return } \mathsf{Step1}(\Pi_{\mathsf{put}}, i_c, \varepsilon, in_C)$

$\underline{\text{Upd1}(i_c, in_C)\text{:}}$
$\textbf{pre}\text{: } in_C.\{fid, f\}$
$\boxed{\textbf{if } \text{T}_{\text{u}}[i_c] \in \text{T}_{\text{f}}[fid].\text{U}\text{: } \text{T}_{\text{f}}[fid].\text{F} \leftarrow \text{T}_{\text{f}}[fid].\text{F} \cup \{f\}}$
$\textbf{return } \mathsf{Step1}(\Pi_{\mathsf{upd}}, i_c, \varepsilon, in_C)$

$\underline{\text{Get1}(i_c, in_C)\text{:}}$
$\textbf{pre}\text{: } in_C.\{fid\}$
$\textbf{return } \mathsf{Step1}(\Pi_{\mathsf{get}}, i_c, \varepsilon, in_C)$

$\underline{\text{Share1}(i_c, i_r, in_C)\text{:}}$
$\textbf{pre}\text{: } in_C.\{fid\}$
$\text{T}_{\text{f}}[fid].\text{U} \xleftarrow{\cup} \{\text{T}_{\text{u}}[i_c]\}$
$\textbf{if } i_r \in \text{T}_{\text{uaid}}\text{: } in_C.raid \leftarrow \text{T}_{\text{uaid}}[i_r]$
$\boxed{\textbf{if } fid \in \text{S}_{\text{chall}}\text{: } \text{S}_{\text{challOwners}} \xleftarrow{\cup} \{i_r, \text{T}_{\text{u}}[i_c]\}}$
$\textbf{return } \mathsf{Step1}(\Pi_{\mathsf{share}}, i_c, i_r, in_C)$

$\underline{\text{Acpt1}(i_c, in_C)\text{:}}$
$\textbf{pre}\text{: } in_C.\{fid\}, in_C.oob \neq \bot \vee \text{T}_{\text{oob}}[(i_r, fid)] \neq \bot$
$i_r \leftarrow \text{T}_{\text{u}}[i_c]$
$\text{T}_{\text{f}}[fid].\text{U} \xleftarrow{\cup} \{i_r\}$
$\textbf{if } in_C.oob = \bot\text{: } in_C.oob \leftarrow \text{T}_{\text{oob}}[(i_r, fid)]$
$\textbf{else}\text{: } \text{T}_{\text{f}}[fid].\text{U} \xleftarrow{\cup} \{\iota_{\text{mal}}\};$ $\boxed{\textbf{if } fid \in \text{S}_{\text{chall}}\text{: } \text{S}_{\text{challOwners}} \xleftarrow{\cup} \{\iota_{\text{mal}}\}}$
$\textbf{return } \mathsf{Step1}(\Pi_{\mathsf{accept}}, i_c, \varepsilon, in_C)$

$\underline{\text{Chall}(i_c, in_C, f_0, f_1, \text{Orc})\text{:}}$
$\textbf{pre}\text{: } in_C.\{fid\}, |f_0| = |f_1|, \text{Orc} \in \{\text{Put1}, \text{Upd1}\}$
$\boxed{\textbf{if } (\text{Orc} = \text{Upd1} \wedge fid \notin \text{S}_{\text{chall}})\text{: } \textbf{return } \bot}$
$in_C.f \leftarrow f_b$ ; $\text{S}_{\text{chall}} \xleftarrow{\cup} \{fid\};$ $\text{S}_{\text{challOwners}} \xleftarrow{\cup} \text{T}_{\text{u}}[i_c] \cup \text{T}_{\text{f}}[fid].\text{U}$
$\textbf{return } \text{Orc}(i_c, in_C)$

$\underline{\text{Comp}(i_u)\text{:}}$
$\boxed{\textbf{if } i_u \notin \text{S}_{\text{comp}} \textbf{ return } \bot}$
$\text{S}_{\text{comp}} \leftarrow \text{S}_{\text{comp}} \cup \{i_u\}$
$\textbf{return } (\textbf{pw}[i_u], \{\text{T}_{\text{stC}}[i_c] \mid \exists i_c : \text{T}_{\text{u}}[i_c] = i_u\}, \{\text{T}_{\text{oob}}[(i_u, \cdot)]\})$
```

Figure 6: Oracles for the client-to-client games (see Figure 5). Code only included in the C2CConf$\boxed{\text{S}}$ or C2CInt$\boxed{\text{S}}$ game is marked with $\boxed{\text{boxed}}$ and $\boxed{\text{dashed}}$ boxes, respectively. Moreover, $\boxed{\text{dotted}}$ code is only included in the *selective security* version of the games.

sharer is added as an owner on lines 17 and 19 in oracle $\textsc{Share1}$. This avoids attacks from compromised users sharing files which they do not own with honest users, causing the receiver to use malicious file key material for future file updates. The game allows the adversary itself to share files with and accept files from honest users by specifying the out-of-band message $in_C.oob$ in $\textsc{Acpt1}$ or using the special index ι_{mal} as recipient in $\textsc{Share1}$, respectively.

All basic oracles use the common subroutine $\mathsf{Step1}$ (in Figure 5) to initialize a protocol run and game-internal process tracking. The adversary can then use $\textsc{Step}$ to execute the next client step in an already running protocol Π, identified by process index i_p, on input the next (adversary-chosen) server message m.

Finally, security is captured via the challenge ($\textsc{Chall}$) and compromise ($\textsc{Comp}$) oracles:

- $\textsc{Chall}$ takes two files (f_0, f_1) of equal size and an oracle name $\{\Pi_{\mathsf{put}}, \Pi_{\mathsf{upd}}\}$. If $\textsc{Orc} = \Pi_{\mathsf{put}}$, the oracle uploads f_b, depending on the game bit b. The adversary can subsequently update the challenge file by running $\textsc{Chall}$ again on the same file ID with $\textsc{Orc} = \Pi_{\mathsf{upd}}$. The game allows for multiple challenge files, the IDs of which are tracked in the set $\mathrm{S}_{\mathsf{chall}}$.
- $\textsc{Comp}$ compromises a given honest user i_u, yielding their password and entire (current) state to the adversary. It may be called at any time, and repeatedly, to capture a persistent user compromise. Compromised users are tracked via $\mathrm{S}_{\mathsf{comp}}$. We do not model partial user compromise. That is, the adversary can only compromise the full state of a user, including all of its active sessions.

<u>Malicious vs compromised clients.</u> Beyond orchestrating actions of honest clients, the adversary can directly simulate any number of *malicious* clients locally, without the need for oracles, since it fully controls the server state. Furthermore, after compromising a client via the oracle $\textsc{Comp}$, the server can continue using the protocol oracles on that client, but can also use the compromised client state to compute operations outside the honest behavior of clients.

Client compromise in particular models that an adversary may learn information about the password distribution. For instance, a scheme that leaks which users have the same password is vulnerable to attacks compromising passwords.

<u>Selective security.</u> In the selective security version of the game, C2CConfS, the adversary has to commit to the set $\mathrm{S}_{\mathsf{comp}}$ of users it is allowed to compromise at the outset of the game (see Figure 5, line 3). It can then still call $\textsc{Comp}$ adaptively at any point and repeatedly in the game, but only on these selected users, to learn their current state.

<u>Winning condition.</u> The adversary wins the game if it correctly guesses the challenge bit b in a non-trivial way. Line 5 of Figure 5 encodes this trivial win check: files that the adversary called the challenge oracle on may only be owned by or shared with honest and non-compromised users $i_u \in \mathrm{S}_{\mathsf{hon}} \setminus \mathrm{S}_{\mathsf{comp}}$. (Note that this in particular encodes that challenge files may not be shared with fully malicious, adversary-controlled clients.)

4.2 Client-to-Client Integrity

We now turn to the second security goal, client-to-client integrity, captured by the game $\mathbf{G}^{\mathrm{C2CInt}}_{\mathsf{CS},n,\mathcal{PW}_n}$ given in the middle panel of Figure 5, again with oracles and helper functions in Figures 5 and 6. The integrity game includes the dashed code and excludes that in $\boxed{\text{solid}}$ boxes; in particular, the challenge oracle $\textsc{Chall}$ is not present in the integrity game.

The task of the adversary in the integrity game is to forge the content of a file, i.e., to make an honest user successfully retrieve a file with a content that differs from what they uploaded or received as a share. This is captured by setting a win flag in line 19 of Figure 5, conditioned on there being no trivial attacks. We define the resulting client-to-client integrity as follows.

Definition 4.3 (Client-to-Client Integrity) *Let* CS *be a cloud storage scheme and* $\mathbf{G}^{\text{C2CInt}\boxed{S}}_{\mathsf{CS},n,\mathcal{PW}_n}$ *be the adaptive* ($\boxed{\textit{selective}}$) *client-to-client integrity* (C2CInt$\boxed{S}$) *game for* CS *defined in Figures 5 and 6, for at most* n *honest users and with respect to a password distribution* $\mathcal{PW}_\mathrm{n}$*. We define the advantage of an adversary* $\mathcal{A}$ *playing this game as*

$$\mathbf{Adv}^{\text{C2CInt}\boxed{S}}_{\mathsf{CS},n,\mathcal{PW}_n}(\mathcal{A}) := \Pr\left[\mathbf{G}^{\text{C2CInt}\boxed{S}}_{\mathsf{CS},n,\mathcal{PW}_n}\right].$$

<u>WHAT COUNTS AS A FORGERY.</u> To evaluate the winning condition (in Figure 5, line 19), we track the content of honestly uploaded files in set $\mathrm{T_f.F}$ and their owners in $\mathrm{T_f.U}$, both of which are implicitly initialized to $\emptyset$.[8] As a valid forgery we count any successful get operation (i.e., for which $out_C.dec = \mathsf{true}$) for which either (1) the file ID does not exist, meaning that the file is not tracked in the game, or (2) the retrieved file content does not match any version of the file stored in $\mathrm{T_f.F}$. For case (2), we also need to ensure the file is not owned by a compromised or malicious user; i.e., the file owners $\mathrm{T_f.U}$ must be a subset of $\mathsf{S_{hon}} \setminus \mathsf{S_{comp}}$.

Note that a file can be either entirely or partially untracked, but line 19 covers both cases. That is, if $\mathrm{T_f}[\mathit{fid}].\mathrm{F} = \emptyset$, then $out_C.f \notin \mathrm{T_f}[\mathit{fid}].\mathrm{F}$ evaluates to true. The file then counts as forgery if $\mathrm{T_f}[\mathit{fid}].\mathrm{U} \subseteq \mathsf{S_{hon}} \setminus \mathsf{S_{comp}}$ is true too, either because the file is entirely untracked and $\mathrm{T_f}[\mathit{fid}].\mathrm{U} = \emptyset$, or because the owner set only contains honest users who shared or accepted a file ID that was never uploaded through oracle PUT1.

To prevent trivial attacks, the win flag is only set to true if the get operation is performed for an honest user (line 18, Figure 5). That is, the user ID $\mathrm{T_u}[i_c]$ of the calling client must be in $\mathsf{S_{hon}} \setminus \mathsf{S_{comp}}$. Otherwise, the adversary could easily forge a file, since it knows the retrieving user's key material.

The tracking of file content and owners in table $\mathrm{T_f}$ is conservative, in three aspects.

- In PUT1 and UPD1, we set resp. add the to-be-uploaded file content immediately in the table, without waiting for successful termination of the protocol. The reasoning here is that we cannot know at which point in the execution of a generic upload protocol a file should be considered successfully stored on the server. In particular, the adversary acting as the malicious server might claim that an upload failed, when it actually did store the corresponding file information which would make a later download successful. However, we only update the tracked file content in oracle UPD1 if the calling user is already an owner of the file. This reflects that the "expected content" (for the purpose of what constitutes a forgery) is only modified if an owner performs the update.
- We never remove file contents from the set of uploaded files under some file ID. This captures that we do not aim to protect against rollback attacks, i.e., a server serving an old copy of a file to a client does not count as an integrity attack.
- As in the confidentiality game, we add sharers as owners of the files they share. This prevents a trivial attack where the adversary uses a compromised user to share a file that is not yet tracked by the game with an honest user. Since the adversary knows the key material of the compromised user, it can create a file header containing a malicious file key which the sharer will decrypt and put in the OOB channel. Upon accepting the shared file, the honest user would then be the (only) owner of a file for which the adversary knows the file key, making a forgery trivial. Adding the sharer as an owner ensures that the file is correctly tracked as owned by a compromised user.

[8] We combine our convention for tables and sets and consider the table entry uninitialized while the sets are implicitly initialized to empty. That is, we let $\mathit{fid} \notin \mathrm{T_f}$ evaluate to true while $\mathrm{T_f}[\mathit{fid}] = (\emptyset, \emptyset)$.

4.3 Designing a Security Model for Cloud Storage: UC vs. Game-Based

Having introduced our security definitions in detail, let us take a moment to explain our decision to propose *game-based* definitions for cloud storage instead of simulation-based security notions in the universal composability (UC) framework [15].

Many protocols with similar aims, particularly those in the multi-server password-protected secret sharing line of research ([7], [35], [14] [33]), analyze their security in the UC framework. This approach has obvious advantages: only one proof is needed to capture all security properties, the results can build upon the framework's powerful composition theorem, and a proof in this model can argue about the security of users' files relative to their passwords without attempting to quantify password security directly.

When we attempt to capture single-server cloud storage security, however, a commitment problem arises. In the UC framework, all honest clients are controlled by the environment, which chooses their inputs to the protocol. If the environment instructs an honest client C to store a file f with a malicious server, the simulator must produce, without knowledge of f, a ciphertext indistinguishable from the one C would send in an honest protocol interaction. If C should later become compromised, the simulator will learn f and must reveal some key under which the prior ciphertext decrypts to f. This is only possible if the cloud storage scheme employs some form of non-committing encryption.

A similar problem appears in the UC treatment of secure channels, where a client can send a message that must be decryptable after compromise [17]. Canetti et al. proposed as a solution an auxiliary "non-information oracle", which takes in client messages (or files, in our setting) and outputs information to the simulator which is computationally independent from the input messages. Effectively, the non-information oracle produces committing ciphertexts for the simulator, and the computational independence of the ciphertexts from their plaintexts demonstrates that no leakage occurs. Upon client compromise, the non-information oracle can forward its internal keys and assist in decryption. This technique achieves UC channel security from standard primitives, and similar methods are used to capture secure messaging in the UC model [16].

Unfortunately, the non-information oracle strategy breaks down when applied to cloud storage due to our reliance on passwords as a basis for security rather than strong cryptographic keys. No matter how many layers of keys a protocol introduces between clients' passwords and the ciphertexts they upload, a client must always be able to recover files using only its password as input by running the `auth` and `get` protocols. Consequently the ciphertexts cannot possibly be computationally independent from their underlying files. To achieve a UC security proof, we would instead require some computational secrecy property that relates the degree of leakage from the non-information oracle to the probability of a successful password-guessing attack. By moving the bulk of the simulation into an oracle with knowledge of client files and then making a separate computational argument about the secrecy of this oracle's outputs, we effectively introduce a game-based confidentiality model. The advantages of the UC framework if we must also have games are significantly less. Thus, we focus solely on game-based notions in the remainder of this work.

4.4 Extensions, Limitations and Rationale of the Model

In the process of defining our cloud storage framework and security notions, there were many choices to be made for how to abstract and formalize real-world cloud storage schemes. Here, we provide rationale for certain aspects of our framework and why we did or did not include certain real-world facets in the model, and discuss the limitations that these choices imply. We also list possible extensions to the security model that could be interesting for future work.

PROTOCOL INPUTS. In our syntax, the execution of a cloud storage protocol only takes external input from the client and server in the first step of the respective party. (See Section 3.1.)

This is a design decision that we made for two reasons. First, it reduces the complexity of games. If we were to allow inputs at all protocol steps, it would be substantially more complex to track the security critical variables. For instance, our correctness and security games need to track the file identifier *fid* in file operations to check (trivial) win conditions. A game that supports taking *fid* as input at any protocol step would require a precondition to check that *fid* is not set multiple times (otherwise the adversary could interfere with the game-internal tracking). Additionally, the tracking and checks could only be performed once *fid* has been provided, and this time-point would now be protocol specific, making generic tracking more difficult.

Second, all protocols that we are aware of can be expressed with a single, initial input, even if it may seem like they require communication at first sight. For instance, consider a protocol to get a file that first fetches a file tree, presents it to the user to let them pick a file name, and then downloads the file with the ID corresponding to the name specified by the user. It may seem like the file ID is needed as a separate input in the second client-side step. However, we can easily model this protocol in our single input syntax by encoding the user action of translating a file name to a file ID into the client-side protocol. A user calls the file retrieval algorithm with the name of the file it wants to download. The client downloads the file tree, resolves the file name to a file ID, and fetches the file.

It is more challenging to accurately model a protocol that uses an out-of-band channel between server and client. (Such channels are not captured by our model, which only supports the client-to-client OOB channel used for file sharing.) An example of such a server-to-client channel could be email or SMS, as used e.g. for two factor authentication. If values exchanged through this channel appear as inputs in intermediate protocol steps, then these inputs cannot be captured by omitting the interaction or reordering (as in the file fetching example above), since they might be influenced by earlier protocol steps. We can either get around this limitation by only modeling the authentication after out-of-band material was already received (see our modeling of email tokens for MEGA in Appendix A.1), or by including the OOB message exchange in the protocol messages over the normal client-server channel. Both approaches do not affect C2C security as the adversary playing the server anyway sees all exchanged messages. However, more accurate modeling would be required for security against external adversaries, as discussed later in this section.

CLIENT-SIDE STATE. We use a single persistent client-side session state st_C (beyond the temporary per-protocol execution state st_C^{tmp}). Therefore, when modeling real-world schemes, our state handling might sometimes need to differ from their actual implementation. For instance, consider a cloud storage scheme that stores the user password on the device after the first login. This increases usability at the cost of security as the user does not need to enter their credentials for every login. However, if the device is corrupted, then the password is compromised, even if there is currently no active session on the device. In our model, we would either model such a protocol by storing the password in the session state, or by requiring the user to enter it for each operation. Both approaches lead to the same adversarial strength in our model with respect to compromise, as the adversary can only compromise users completely, learning both their password and any session state. In other words, we cannot capture that not storing the password offers better security against device compromise.

A more fine-grained model could distinguish between user, device, and session states, capturing settings where one device runs multiple sessions in parallel, keeping some information in device memory and other information in per-session memory. This could allow to capture more fine-grained compromise notions, e.g., the security effects of a compromise of an old, revoked user device on later-uploaded files, or that of a compromised session on one device on another, uncompromised session on the same device. We leave capturing such aspects to future work.

Out-of-band channel. Our model modularizes the out-of-band channel used for file sharing. This allows it to capture a wide range of real-world instantiations, such as sharing via a PKI or through links with encoded passwords. However, this modularization also means that within our confidentiality and integrity notions, the security of the out-of-band channel is idealized; it is assumed to be perfectly secure and incorruptible. A full security analysis of a concrete cloud storage system would hence need to separately prove the security of the actual channel used, and additionally analyze the security of its composition with the cloud storage scheme.

Security against external adversaries. An alternative security notion to the client-to-client security given above would be to consider security against an *external* adversary that does not control the server but only plays the role of malicious clients. While this at first seems strictly weaker, in this model one could capture security properties which are trivially broken by a malicious cloud provider in the C2C setting, e.g.:

- *Limited password guessing:* an external adversary must interact with the server to verify password guesses online. In particular, the protocol does not leak any information to unauthenticated users that would allow offline password guesses. (In the C2C setting, the malicious server can always perform offline password guesses against the stored files of honest users.)
- *No rollback attacks:* an external adversary should not be able to serve old versions of a file to honest users. (In the C2C setting, rollback attacks cannot be prevented without additional assumptions, such as MPC, third parties, or direct device-to-device communication: a malicious server can always roll back the full server state and serve a previous version to a user who is logging in from a fresh device. We are not aware of any currently deployed E2EE cloud storage system that attempts to protect against such rollback attacks.)
- *Availability:* an external adversary should not be able to prevent honest users from fetching files which are not owned or shared with compromised or malicious users. (In the C2C setting, the malicious server can always reject to serve, or garble, the data of honest users.)
- *OOB channels:* real systems often use multiple channels between the server and the client, beyond the main TLS connection. For instance, using email or SMS as an out-of-band channel for multi-factor authentication is common practice. While the malicious server trivially knows the messages exchanged in these channels, the benefit of multiple channels becomes visible against an external adversary who may compromise some, but not all, channels.

We leave formalizing security against external adversaries as future work, but note that we kept the above properties in mind when designing the cloud storage protocol in Section 6.

Additional functionality. The core functionality considered in this paper could be extended – at the cost of higher complexity – with extra features such as file deletion, revocation of shared files, password recovery, session deauthentication (and ensuing security notions for adversaries with access to expired sessions), account deregistration, and distinguishing between read and write access to files. Out of these, more advanced access management (e.g., supporting revocation of access to shared files, or providing distinct access classes such as "owner", "editor" or "read-only") is especially interesting in the external adversary setting, where the server can be (at least partially) trusted to help enforce access restrictions.

Metadata. Protection of metadata, such as file names, types and paths, can be crucial for the overall security of a cloud storage system. For instance, insufficient encryption of file names or paths would leak information about the uploaded file to the server. Moreover, not protecting the integrity of metadata may directly affect the file integrity (e.g., when a file length field is modified).

In our security notions, we implicitly model strong security guarantees for metadata by treating it as part of the "file" plaintext input to oracle PUT1. This implies that the metadata must be confidential and integrity-protected even from a malicious server. Note that protocols which encrypt metadata separately from files for efficiency can still be captured in this model (e.g., by treating the metadata as its own "file").

For the sake of complexity, we leave the modeling of weaker security guarantees of metadata – such as "authentication-only" – and the impact thereof on file security to future work. Moreover, our CSS protocol does not aim to protect against usage metadata, such as file access patterns and share relationships. Constructing a protocol that achieves the metadata-hiding properties of Metal [22] and Titanium [21] while having the provable guarantees of our detailed security models – including password-based security, a single fully malicious server, non-atomic operations, and considering multiple devices – is an open problem.

CHANNEL ASSUMPTIONS. Additional work could explore different channel assumptions. For instance, only assuming an authenticated but not confidential communication channel would give the adversary access to the exchanged messages. This may be interesting as a standalone solution, or as defense-in-depth measure to protect against implementation failures that occasionally undermine confidentiality guarantees in practice. For instance, implementation failures of TLS session tickets lead to the loss of confidentiality of early data for some implementations [29].

Moreover, specific instantiations of the OOB channel, especially those operated by the malicious server, could be integrated into CSS and analyzed.

ADVANCED SECURITY GUARANTEES. It is an open question how post-compromise and forward security apply to the cloud storage setting, and what is achievable; key rotation has been considered in prior works [36], as has fine-grained forward security via puncturing techniques [5, 53], but not in combination with the full feature set we target. Furthermore, CSS does not attempt to protect against partial rollback attacks. For instance, future work could attempt to protect the integrity of the user's cloud storage state to prevent a malicious server from selectively serving some old and some new files.

5 Formally Breaking MEGA

To put our new cloud storage framework to the test, we will give both a negative analysis, capturing some of the recent attacks against the MEGA cloud storage system from [6, 3, 30] in our framework (in this section), and a positive one, presenting a provably secure cloud storage scheme (in the Section 6).

MEGA provides end-to-end encrypted cloud storage since 2013 and is currently the largest provider with E2EE by default, with roughly 300 million users [40]. In addition to its popularity, MEGA is an interesting system to study because recent cryptanalysis [6, 3, 30] showed that clients released prior to June 2022 [44] were vulnerable to a range of attacks by a malicious server. In the following, we show that MEGA can be cast in our syntax, demonstrating the expressiveness of our framework, and that recent attacks [6, 3, 30] arise in our framework as formal violations of the C2C security notions (already for selective security, as the attacks do not rely on client compromise). While the attacks discussed here target the unpatched version of MEGA, the patched protocol still cannot be proven secure in our model due to various design decisions, including key reuse and the lack of binding between file IDs and files.

In this section, we only highlight the core, vulnerable protocol components and how the main attack ideas can be cast in our security framework. We defer the full details of formalizing the MEGA protocols, attacks, and barriers for proving the patched protocol secure to Appendix A.

```
⋮
client: $\mathsf{M}_{\mathsf{auth}}^{(C:3)}(st_C, st_C^{\mathrm{tmp}}, m_S)$:
 1  $(c_m, (pk, c_{rsa}), c_{sid}) \leftarrow m_S$
 2  $k_m \leftarrow$ AES-ECB.Dec$(k_e, c_m)$
 3  $sk \leftarrow$ AES-ECB.Dec$(k_m, c_{rsa})$
 4  $st_C.sid \leftarrow$ RSA.Dec$(sk, c_{sid})$ ; $st_C.k_m \leftarrow k_m$
 5  $st_C.pk \leftarrow pk$ ; $st_C.sk \leftarrow sk$
 6  return $(st_C, st_C^{\mathrm{tmp}}, out_C, st_C.sid)$
$\mathsf{M}_{\mathsf{auth}}^{(C:4)}(st_C, st_C^{\mathrm{tmp}}, m_S)$:
 7  if $m_S = \bot$: fail$_C$ else: success$_C$

⋮
server: $\mathsf{M}_{\mathsf{auth}}^{(S:2)}(st_S, st_S^{\mathrm{tmp}}, m_C)$:
 8  $aid \leftarrow st_S^{\mathrm{tmp}}.aid$
 9  $(n_C, c_m, h_a, pk, c_{rsa}) \leftarrow st_S.acc[aid]$
10  . . . // omitted
11  $c_{sid} \leftarrow$ RSA.Enc$(pk, sid)$
12  $m_S \leftarrow (c_m, pk, c_{rsa}, c_{sid})$
13  return $(st_S, st_S^{\mathrm{tmp}}, \varepsilon, m_S)$
$\mathsf{M}_{\mathsf{auth}}^{(S:3)}(st_S, st_S^{\mathrm{tmp}}, m_C)$:
14  if $m_C \neq sid$: fail$_S$
15  $st_S.sess[sid] \leftarrow st_S^{\mathrm{tmp}}.aid$
16  success$_S$
```

Figure 7: The part of MEGA's authentication protocol $\mathsf{M}_{\mathsf{auth}}$ hosting the core vulnerabilities; see Figure 12 in Appendix A.1 for the full protocol.

The vulnerable core of MEGA, formalized. Highly simplified, in the account registration protocol $\mathsf{M}_{\mathsf{areg}}$, a client generates a uniform, symmetric master key k_m and an RSA key pair (pk, sk). It then encrypts sk under k_m, and k_m in turn under a password-derived key k_e, using AES-ECB encryption for both, and uploads the resulting ciphertexts c_{rsa} and c_m, respectively, to the server.

During authentication ($\mathsf{M}_{\mathsf{auth}}$, see Figure 7), the server sends c_{rsa} and c_m back to the client, along with another ciphertext c_{sid} which encrypts a server-chosen session ID sid under the client's RSA public key. The client decrypts, in order, c_m, c_{rsa}, and c_{sid}, and sends the obtained session ID value back to the server to complete authentication.

To upload a new file ($\mathsf{M}_{\mathsf{put}}$), the MEGA client generates a new file key k_f and encrypts the file using a custom-made authenticated encryption algorithm. For the attack, the key thing to note is that the file key is encrypted (alongside a nonce and an authentication tag) under the master key k_m using AES-ECB and stored on the server, just like the user's RSA private key.

The authentication protocol hosts the core vulnerability; we hence show its relevant steps in Figure 7. The full details of all protocols are in Appendix A.1.

Breaking confidentiality. The attacks from [3, 6, 30] now translate into formal violations of our security games for the MEGA cloud storage scheme. We summarize the attacks and their formalization in our framework here and provide the full pseudocode of the adversaries for attacks 1–3 from [6] (discussing aspects from [3, 30]) in Appendix A.2.

We begin by showing how the RSA key and plaintext recovery attacks by Backendal et al. [6] break C2CConfS confidentiality, by combining them into an adversary $\mathcal{A}_{conf}$ that wins the $\mathbf{G}_{\mathsf{M},n,\mathcal{PW}_n}^{\mathrm{C2CConfS}}$ game (see Definition 4.2) with high probability. The adversary starts by registering an honest user and lets them upload two distinct files as challenge. It then runs a simpler version of the RSA key recovery attack [6, Section III] by letting the user repeatedly authenticate, tampering with the AES-ECB ciphertext c_{rsa} and exploiting the returned session ID sid as a partial decryption oracle. For the latter, observe that $\mathcal{A}_{conf}$, acting as the malicious server in the $\mathsf{M}_{\mathsf{auth}}$ protocol, can pick an arbitrary ciphertext (on line 12 of Figure 7) which the honest client will decrypt using the maliciously modified RSA secret key sk, and send back in the client message m_C of $\mathsf{M}_{\mathsf{auth}}^{(S:3)}$. This allows $\mathcal{A}_{conf}$ to recover the user's sk through a binary search.

Knowing the user's RSA secret key, the adversary can then mount the AES-ECB decryption attack from [6, Section IV] to decrypt the challenge file. On a high level, $\mathcal{A}_{conf}$ injects the encrypted challenge file key into the RSA ciphertext c_{rsa} sent in the authentication protocol. This is possible because both the RSA key and file keys are encrypted with the master key k_m using AES-ECB, enabling the replacement of individual ciphertext blocks in c_{rsa}. This second attack step is probabilistic and fails with probability at most 2^{-30} (see [6]). Adversary $\mathcal{A}_{conf}$ wins with advantage $\mathbf{Adv}^{\text{C2CConfS}}_{\mathsf{M},n,\mathcal{PW}_n}(\mathcal{A}_{conf}) = 1 - 2^{-30}$, making 1023 calls to $\mathsf{M}_{\mathsf{auth}}$. (The number of oracle calls to $\mathsf{M}_{\mathsf{auth}}$ needed to mount the attack was since reduced to 6 by Heninger and Ryan [30].)

Breaking integrity. To violate C2CIntS integrity, we formalize the integrity attack from [6, Section V.B.2]. Instead of assuming the knowledge of a plaintext-ciphertext pair (which is required for their attack) we integrate the AES-ECB encryption oracle identified by Albrecht et al. [3, Section 2.2] to exercise the full scope of our framework. This allows the adversary to obtain the ciphertext for a chosen plaintext via the sharing protocol.

The C2CIntS adversary $\mathcal{A}_{int}$ first honestly registers and authenticates a user, and then makes them accept a shared file chosen by $\mathcal{A}_{int}$ (which, of course, does not count as win yet). The client will re-encrypt the shared file key under its master key using AES-ECB, at which point the adversary knows both the plaintext and ciphertext of the file key. Adversary $\mathcal{A}_{int}$ can exploit that MEGA's custom decryption algorithm causes the client to derive an all-zero file key during file downloads when an AES-ECB block is duplicated in the file key ciphertext. Knowing the decryption of one ciphertext block and carefully crafting an encrypted file (see Appendix A.2 and [6, Section V.B.3]), $\mathcal{A}_{int}$ succeeds in making the user decrypt a forged file and thus wins the integrity game. This adversary $\mathcal{A}_{int}$ has advantage $\mathbf{Adv}^{\text{C2CIntS}}_{\mathsf{M},n,\mathcal{PW}_n}(\mathcal{A}_{int}) = 1$ with a total of only four oracle calls.

In conclusion, we observe that problems with the E2EE cloud storage protocol of MEGA show up naturally when it is written down in our syntax and tested against our security games.

6 A Provably Secure Cloud Storage Scheme

In this section, we showcase the instantiability of our framework by presenting a concrete cloud storage scheme CSS which enjoys provable client-to-client security. The scheme provides all functionality from the core protocols, including file sharing, and supports password rotation without file re-encryption. We first introduce the building blocks used in our construction, before explaining the scheme itself and then analyzing its security. Appendix 6.5 presents limitations and extensions of the scheme, including a discussion of selective vs. adaptive security and considerations in practice.

6.1 Building Blocks

PRF, AEAD and MAC. We use the following basic building blocks, further specified in Appendix B.1, in our cloud storage scheme:

- a PRF $\mathsf{F}\colon \{0,1\}^{kl} \times \{0,1\}^* \to \{0,1\}^{kl}$ taking a key $k \in \{0,1\}^{kl}$ and a variable-length label $x \in \{0,1\}^*$ as inputs and returning a string of length kl.
- a message authentication code $\mathsf{MAC} = (\mathsf{Tag}, \mathsf{Vrfy})$ with key space $\{0,1\}^{kl}$.
- a nonce-based authenticated encryption scheme with associated data $\mathsf{AEAD} = (\mathsf{Enc}, \mathsf{Dec})$, also with key space $\{0,1\}^{kl}$. We write $c \leftarrow \mathsf{Enc}(k, n, m, ad)$ to encrypt message $m \in \{0,1\}^*$ using key $k \in \{0,1\}^{kl}$, nonce $n \in \{0,1\}^{nl}$, and associated data $ad \in \{0,1\}^*$, and $m \leftarrow \mathsf{Dec}(k, n, c, ad)$ for the corresponding decryption of the ciphertext $c \in \{0,1\}^*$.

Note that, for ease of presentation, we assume that the output length kl of the PRF is the key length of the MAC and AEAD scheme.

2HashDH OPRF. Our construction further uses $\mathsf{2HashDH}$ [32, 34], an oblivious pseudorandom function (OPRF) [47, 26], to derive a hardened secret from the user password. An OPRF is an interactive protocol between two parties (henceforth called client and server), which allows the client, holding a secret input x, to compute a PRF evaluation $y = \mathsf{F}(k, x)$ with the help of the server, which holds the key k. In our use, we let the client sample the OPRF key k during account registration and subsequently send it to the server; this ensures that the initial client output is computed with an honestly generated key.

We briefly recap the $\mathsf{2HashDH}$ OPRF here. Let G be a group of prime order q, and let $\mathrm{H}_1 \colon \{0,1\}^* \to G$ and $\mathrm{H}_2 \colon \{0,1\}^* \times G \to \{0,1\}^{kl}$ be two hash functions. The $\mathsf{2HashDH}$ OPRF interactively computes $\mathrm{H}_2(x, \mathrm{H}_1(x)^k))$ for the client secret input $x \in \{0,1\}^*$ and server-side key $k \in \mathbb{Z}_q$. In the interactive protocol, the input x is hidden from the server with a random blinding value $r \leftarrow\!\!\$\, \mathbb{Z}_q$. The client sends $\alpha \leftarrow \mathrm{H}_1(x)^r$ to the server, which computes the blind evaluation $\beta \leftarrow \alpha^k$. The result is recovered on the client by unblinding β using r, computing $y \leftarrow \mathrm{H}_2(x, \beta^{1/r}))$. We give the full specification in our syntax in Appendix B.2.

The goal of using an OPRF in our cloud storage protocol is to provide a means for the client to derive a key from the user password, which is secure against a malicious server. This is a somewhat non-standard setting for an OPRF, prompting us to adapt $\mathsf{2HashDH}$ in the following ways (further discussed in Section 6.3):

- The client secret x consists of (aid, pw), i.e., the combination of the *public* account ID and the user password. Thanks to the assumption that account IDs are unique, this implies that client secrets are unique.
- The OPRF key is sampled by the client during registration, and subsequently sent to the server; this ensures that the initial client output is computed with an honestly generated key.
- H_1 and H_2 should ideally be memory-hard and slow, to increase the work required for a password-guessing attack from a malicious server.

6.2 Scheme Overview

We specify the protocols that make up our cloud storage scheme $\mathsf{CSS} = (\mathsf{CSS}_{\mathsf{areg}}, \mathsf{CSS}_{\mathsf{auth}}, \mathsf{CSS}_{\mathsf{put}}, \mathsf{CSS}_{\mathsf{upd}}, \mathsf{CSS}_{\mathsf{get}}, \mathsf{CSS}_{\mathsf{share}}, \mathsf{CSS}_{\mathsf{accept}})$, in pseudocode in Figures 8–10. Let us first explain the core design rationale behind our cloud storage scheme, before describing the algorithms in more detail.

REGISTRATION AND AUTHENTICATION. To register an account, a user first derives a hardened secret rw from their account ID and password using the $\mathsf{2HashDH}$ OPRF. They then AEAD-encrypt a randomly generated master key under a key derived from rw. To complete registration, the client uploads the encrypted master key, the OPRF server key and a MAC key derived from rw to the server.

To authenticate, a user evaluates the $\mathsf{2HashDH}$ OPRF on their account ID and password with the help of the server(-side key) to recover rw, and from it the master key and MAC key. The server assigns a fresh session ID (sid) to the client and uses the MAC key to verify that the client can produce a valid tag for the sid (and interaction transcript) as a means of authentication. Subsequent requests in the same session are authenticated by sending along the session ID.

FILE HANDLING. Our scheme follows the established approach of using a key hierarchy, employing the master key as key encryption key to wrap file keys, which in turn encrypt single files. Both file keys and files are AEAD-encrypted, bound to the owner and file ID, respectively. Each encrypted file is stored along with a header containing the wrapped file key. Importantly, we use the independently sampled master key as the root of the key-wrapping hierarchy in our cloud storage scheme instead of directly using the raw secret rw derived from the password; this way, users can change their password without having to re-encrypt all their data. The server stores the encrypted files and headers and maintains a record of owners to perform access control.

Sharing and updates. To share access to a file with another registered user, an owner shares the file key with the recipient out-of-band and informs the server about the added owner. The newly added user can then wrap the file key under its own master key and add it to the file header, allowing regular future access to the file.

The separation between headers and encrypted file content allows lightweight updates of shared files: any owner can update the encrypted file content while preserving the file key, thereby retaining access for all other owners.

6.3 Detailed Description

Notational conventions. In the pseudocode description of the CSS scheme in Figures 8–10, we use the convention that subroutines returning $\perp$ cause their callers to abort by running the appropriate choice of $\mathbf{fail}_C$ or $\mathbf{fail}_S$.

Account registration. In the account registration protocol $\mathsf{CSS}_{\mathsf{areg}}$, shown at the top of Figure 8, the client samples a 2HashDH OPRF key k and locally computes the OPRF value rw on its account ID aid and password pw. Involving the unique account ID here ensures uniqueness of the OPRF input (even if passwords are reused across users), preventing a malicious server from serving encrypted files across users that it suspects to share the same password. In a sense, we use the pair (aid, pw) as what was the password input in prior usage of 2HashDH [32, 34]; this allows us to avoid assuming that passwords are unique – an assumption that was necessary in other password-based encryption work [12], but is problematic in practice – and only make the milder assumption that (honest) users use unique account IDs, such as email addresses.

The client then derives a key encryption key k_{kek} and MAC key k_{mac} from rw using a PRF F, samples a symmetric master key k_{mk} uniformly at random, and AEAD-encrypts k_{mk} under k_{kek} (bound to the account ID via the associated data field). It sends its aid, the OPRF key k, the MAC key k_{mac}, and the computed ciphertext and used nonce to the server for storage in its account registry acc.

Authentication. In the authentication protocol $\mathsf{CSS}_{\mathsf{auth}}$ (Figure 8 bottom), the client sends the blinded hash $\alpha = \mathrm{H}_1(aid, pw)^r$ to initiate a 2HashDH OPRF run. The server computes the 2HashDH evaluation $\beta = \alpha^k$ and samples a random, unused session ID sid (of bit-length $sidl$), sending both to the client. The client unblinds the OPRF value to obtain its raw secret rw and derives k_{kek} and k_{mac} from it. It then authenticates to the server by sending a MAC value over the first two protocol messages (in particular containing sid as random challenge), using key k_{mac}. Upon successful verification of the MAC, the server adds the client to its session registry $sess$ and sends back the client's encrypted master key. The client AEAD-decrypts its master key k_{mk} using k_{kek} (aborting implicitly upon failure) and saves the session state st_C containing aid, sid, and k_{mk}.

Core file operations. The core operations for handling files in the protocols $\mathsf{CSS}_{\mathsf{put}}$, $\mathsf{CSS}_{\mathsf{upd}}$, $\mathsf{CSS}_{\mathsf{get}}$, $\mathsf{CSS}_{\mathsf{share}}$, and $\mathsf{CSS}_{\mathsf{accept}}$ (all in Figure 9) is captured through subroutines for getting/downloading and putting/uploading headers and contents of files, shown in Figure 10.

putFile, putHeader. In the core operation for putting a file or header, the client AEAD-encrypts the given file resp. file key under the file resp. master key, to be sent to the server. The server ensures that the client user is an owner of the file ($aid \in ows[fid]$) and then stores the encryption in its file or file-key registry (fs and fks), respectively.

getFile, getHeader. Conversely, for fetching a file or header, the client first sends the corresponding file ID fid. The server ensures that the client user owns the requested file and then returns the encrypted header and (for $\mathsf{getFile}$) file content from its registries. The client AEAD-decrypts the header ciphertext c_{fk} using its master key, and, for a file download, the file ciphertext c_f using the resulting file key.

In each of these subroutines, the client sends its session ID *sid* as authentication token along with its message. The server uses *sid* to associate the request to an authenticated user account $\mathit{aid} \leftarrow \mathit{sess}[\mathit{sid}]$, implicitly failing if $\mathit{sid} \notin \mathit{sess}$.

<u>Up- and download.</u> The complete upload of a file ($\mathsf{CSS}_{\mathsf{put}}$, first box in Figure 9) consists of the client sampling a fresh AEAD file key $k_f \leftarrow_{\$} \{0,1\}^{kl}$ and uploading that key as header ($\mathsf{putHeader}$) along with the file content ($\mathsf{putFile}$); the server in turn ensures that this file ID is unused ($\mathit{fid} \notin \mathit{ows}$) and then registers the client user as owner and stores the encrypted header and file content. To update a file ($\mathsf{CSS}_{\mathsf{upd}}$, second box in Figure 9), the client first fetches the file key k_f via $\mathsf{getHeader}$ and then uses it to upload the updated file content ($\mathsf{putFile}$). Downloading a file ($\mathsf{CSS}_{\mathsf{get}}$, third box in Figure 9) uses $\mathsf{getFile}$ to retrieve the file content.

<u>Sharing.</u> Our CSS scheme allows sharing of files with other, registered users. To this end, a client invokes the $\mathsf{CSS}_{\mathsf{share}}$ protocol (fourth box in Figure 9) on input a file ID *fid* and the account ID *raid* of the receiving user with whom that file shall be shared. The sharing client fetches the file key k_f via $\mathsf{getHeader}$, upon whose success the server adds *raid* to the set of owners for file *fid*. The client can then share k_f out-of-band with the receiving user by setting $out_C.oob \leftarrow k_f$. We intentionally leave the details of the out-of-band channel open; we only assume that it is inaccessible to the malicious cloud adversary (except for compromised clients), and aim to minimize its usage. Possible instantiations include using a secure messaging app or an external PKI infrastructure.

To accept the sharing of a file, a client in the $\mathsf{CSS}_{\mathsf{accept}}$ protocol (last box in Figure 9) uses the $\mathsf{putHeader}$ subroutine to upload the shared file key obtained through the out-of-band channel ($in_C.oob$), encrypted under the user's master key, as a new header for the shared file ID *fid*. That is, each file owner has its own file header (which all contain the same file key, wrapped under their individual master keys), but the actual file ciphertext is shared among all owners. Hence, $\mathsf{CSS}_{\mathsf{upd}}$ reuses the file key to ensure that access to shared files is maintained across updates. We do not model an "unshare" operation; however, any owner can delete a file by overwriting it with $\perp$.

6.4 Proving CSS Secure

We are now ready to present our fully quantified bounds on the selective confidentiality and integrity of CSS. They are dominated by the probability of a successful offline dictionary attack on user passwords, which is inherent in the malicious-server setting due to unavoidable guessing attempts. That is, since the server possesses ciphertexts created by the user under a key derivable from their password, a malicious server can attempt to brute-force the password and use the ciphertexts to verify guesses without any client interaction. The only protection against this type of attack is hence to choose a strong password; this is reflected in our theorem bounds by the term corresponding to the password-guessing advantage against the password distribution from which users sample their passwords in our model.

As we outlined in the introduction, our proofs pertain to our selective security notions rather than the adaptive notions because of the commitment issues that arise with adaptive compromises. For CSS, in particular, the most challenging issues arise because the challenge files can be shared with multiple users. In the proof, this means that the reduction needs to replace the master keys of all owners of challenge files with random in order for us to be able to reduce to the security of the AEAD scheme used for file key encryption in the next step. This replacement needs to happen during registration, since the master key is generated in that step and then stored in encrypted form

```
$\underline{\mathsf{CSS}^{(C:1)}_{\mathsf{areg}}(\varepsilon, \varepsilon, in_C, \varepsilon)~:}$
$\textbf{pre}: in_C.\{aid, pw\}$
$k \leftarrow\!\!\$\ \mathbb{Z}_q$
$rw \leftarrow \mathrm{H}_2(aid, pw, \mathrm{H}_1(aid, pw)^k)$
$k_{kek} \leftarrow \mathsf{F}(rw, \text{“kek”})$
$k_{mk} \leftarrow\!\!\$\ \{0,1\}^{kl}$ ; $n_{mk} \leftarrow\!\!\$\ \{0,1\}^{nl}$
$c_{mk} \leftarrow \mathsf{Enc}(k_{kek}, n_{mk}, k_{mk}, (aid, \text{“mk”}))$
$k_{mac} \leftarrow \mathsf{F}(rw, \text{“mac”})$
$m_C \leftarrow (aid, k, n_{mk}, c_{mk}, k_{mac})$
$\textbf{success}_C(m_C)$
```

```
$\underline{\mathsf{CSS}^{(S:1)}_{\mathsf{areg}}(st_S, \varepsilon, in_S, m_C):}$
$\textbf{pre}: st_S.\{acc\}$
$(aid, k, n_{mk}, c_{mk}, k_{mac}) \leftarrow m_C$
$acc[aid] \leftarrow (k, n_{mk}, c_{mk}, k_{mac})$
$\textbf{success}_S$
```

```
$\underline{\mathsf{CSS}^{(C:1)}_{\mathsf{auth}}(st_C, \varepsilon, in_C, \varepsilon):}$
$\textbf{pre}: in_C.\{aid, pw\}$
$r \leftarrow\!\!\$\ \mathbb{Z}_q$ ; $\alpha \leftarrow \mathrm{H}_1(aid, pw)^r$
$m_C \leftarrow (aid, \alpha)$
$st_C^{\mathrm{tmp}} \leftarrow (aid, pw, r, m_C)$
$\textbf{return}\ (st_C, st_C^{\mathrm{tmp}}, \varepsilon, m_C)$

$\underline{\mathsf{CSS}^{(C:2)}_{\mathsf{auth}}(st_C, st_C^{\mathrm{tmp}}, m_S):}$
$(st_C^{\mathrm{tmp}}.sid, \beta) \leftarrow m_S$
$(aid, pw, r, m'_C) \leftarrow st_C^{\mathrm{tmp}}$
$\textbf{if}\ \beta \notin G$: $\textbf{fail}_C$
$\gamma \leftarrow \beta^{1/r}$; $rw \leftarrow \mathrm{H}_2(aid, pw, \gamma)$
$st_C^{\mathrm{tmp}}.k_{kek} \leftarrow \mathsf{F}(rw, \text{“kek”})$
$k_{mac} \leftarrow \mathsf{F}(rw, \text{“mac”})$
$m_C \leftarrow \mathsf{Tag}(k_{mac}, (m'_C, m_S))$
$\textbf{return}\ (st_C, st_C^{\mathrm{tmp}}, \varepsilon, m_C)$

$\underline{\mathsf{CSS}^{(C:3)}_{\mathsf{auth}}(st_C, st_C^{\mathrm{tmp}}, m_S):}$
$\textbf{pre}: st_C^{\mathrm{tmp}}.\{k_{kek}, aid, sid\}$
$(n_{mk}, c_{mk}) \leftarrow m_S$
$k_{mk} \leftarrow \mathsf{Dec}(k_{kek}, n_{mk}, c_{mk}, (aid, \text{“mk”}))$
$st_C \leftarrow (aid, sid, k_{mk})$
$\textbf{success}_C$
```

```
$\underline{\mathsf{CSS}^{(S:1)}_{\mathsf{auth}}(st_S, \varepsilon, in_S, m_C):}$
$\textbf{pre}: st_S.\{acc, sess\}$
$(aid, \alpha) \leftarrow m_C$
$\textbf{if}\ (aid \notin acc) \vee (\alpha \notin G)$: $\textbf{fail}_S$
$(k, n_{mk}, c_{mk}, k_{mac}) \leftarrow acc[aid]$
$\beta \leftarrow \alpha^k$
$sid \leftarrow\!\!\$\ \{0,1\}^{sidl}$ s.t. $sid \notin sess$
$m_S \leftarrow (sid, \beta)$
$st_S^{\mathrm{tmp}} \leftarrow (aid, k_{mac}, m_C, m_S)$
$\textbf{return}\ (st_S, st_S^{\mathrm{tmp}}, \varepsilon, m_S)$

$\underline{\mathsf{CSS}^{(S:2)}_{\mathsf{auth}}(st_S, st_S^{\mathrm{tmp}}, m_C):}$
$\textbf{pre}: st_S.\{sess\}$
$(aid, k_{mac}, m'_C, m'_S) \leftarrow st_S^{\mathrm{tmp}}$
$\textbf{if}\ \neg\mathsf{Vrfy}(k_{mac}, (m'_C, m'_S), m_C)$: $\textbf{fail}_S$
$sess[sid] \leftarrow aid$
$m_S \leftarrow (n_{mk}, c_{mk})$
$\textbf{success}_S(m_S)$
```

Figure 8: Account registration and authentication protocols of CSS.

on the server. But in the adaptive game, the reduction does not know at the time of registration whether a user will become the owner of a challenge file later, or be compromised. Hence we run into a commitment issue: the reduction needs to commit to a ciphertext corresponding to the user master key during registration, but it does not yet know whether it will later have to open this commitment (if the user is compromised) or if this ciphertext should be independent of the actual master key (if the user is made the owner of a challenge file).

We are not aware of any attacks on CSS in the adaptive setting. We hence hypothesize that this is a proof technique issue, rather than an issue with the protocol. Indeed, one could possibly

```
$\mathsf{CSS}^{(C:1)}_{\mathsf{put}}(st_C, \varepsilon, in_C, \varepsilon)$:
    pre: $in_C.\{fid, f\}$ $st_C.\{sid\}$
    $k_f \leftarrow\$ \{0,1\}^{kl}$
    $m_{fk} \leftarrow\$ \mathsf{putHeader}^{(C:1)}(st_C, fid, k_f)$
    $m_f \leftarrow\$ \mathsf{putFile}^{(C:1)}(st_C, fid, k_f, f)$
    $m_C \leftarrow (fid, m_f, m_{fk})$
    success$_C(m_C)$
```

```
$\mathsf{CSS}^{(S:1)}_{\mathsf{put}}(st_S, \varepsilon, in_S, m_C)$:
    pre: $st_S.\{ows, sess\}$
    $(fid, m_f, m_{fk}) \leftarrow m_C$
    if $fid \in ows$: fail$_S$
    $ows[fid] \leftarrow \{sess[sid]\}$
    $\mathsf{putHeader}^{(S:1)}(st_S, m_{fk})$
    $\mathsf{putFile}^{(S:1)}(st_S, m_f)$
    success$_S$
```

```
$\mathsf{CSS}^{(C:1)}_{\mathsf{upd}}(st_C, \varepsilon, in_C, \varepsilon)$:
    pre: $in_C.\{fid, f\}$, $st_C.\{sid\}$
    $m_C \leftarrow \mathsf{getHeader}^{(C:1)}(st_C, fid)$
    $st_C^{\mathrm{tmp}}.\{fid, f\} \leftarrow \{fid, f\}$
    return $(st_C, st_C^{\mathrm{tmp}}, \varepsilon, m_C)$

$\mathsf{CSS}^{(C:2)}_{\mathsf{upd}}(st_C, st_C^{\mathrm{tmp}}, m_S)$:
    pre: $st_C^{\mathrm{tmp}}.\{fid, f\}$
    $k_f \leftarrow \mathsf{getHeader}^{(C:2)}(st_C, fid, m_S)$
    $m_C \leftarrow \mathsf{putFile}^{(C:1)}(st_C, fid, k_f, f)$
    success$_C(m_C)$
```

```
$\mathsf{CSS}^{(S:1)}_{\mathsf{upd}}(st_S, \varepsilon, in_S, m_C)$:
    $m_S \leftarrow \mathsf{getHeader}^{(S:1)}(st_S, m_C)$
    return $(st_S, \varepsilon, \varepsilon, m_S)$

$\mathsf{CSS}^{(S:2)}_{\mathsf{upd}}(st_S, st_S^{\mathrm{tmp}}, m_C)$:
    $\mathsf{putFile}^{(S:1)}(st_S, m_C)$
    success$_S$
```

```
$\mathsf{CSS}^{(C:1)}_{\mathsf{get}}(st_C, \varepsilon, in_C, \varepsilon)$:
    pre: $in_C.\{fid\}$, $st_C.\{sid\}$
    $m_C \leftarrow \mathsf{getFile}^{(C:1)}(st_C, fid)$
    $st_C^{\mathrm{tmp}}.fid \leftarrow fid$
    return $(st_C, \varepsilon, \varepsilon, m_C)$

$\mathsf{CSS}^{(C:2)}_{\mathsf{get}}(st_C, st_C^{\mathrm{tmp}}, m_S)$:
    pre: $st_C.\{aid\}$, $st_C^{\mathrm{tmp}}.\{fid\}$
    $f \leftarrow \mathsf{getFile}^{(C:2)}(st_C, fid, m_S)$
    $out_C.f \leftarrow f$
    success$_C$
```

```
$\mathsf{CSS}^{(S:1)}_{\mathsf{get}}(st_S, \varepsilon, in_S, m_C)$:
    $m_S \leftarrow \mathsf{getFile}^{(S:1)}(st_S, m_C)$
    success$_S(m_S)$
```

```
$\mathsf{CSS}^{(C:1)}_{\mathsf{share}}(st_C, \varepsilon, in_C, \varepsilon)$:
    pre: $in_C.\{raid, fid\}$ $st_C.\{sid\}$
    $m'_C \leftarrow \mathsf{getHeader}^{(C:1)}(st_C, fid)$
    $st_C^{\mathrm{tmp}}.fid \leftarrow fid$
    return $(st_C, st_C^{\mathrm{tmp}}, \varepsilon, (m'_C, raid, fid))$

$\mathsf{CSS}^{(C:2)}_{\mathsf{share}}(st_C, st_C^{\mathrm{tmp}}, m_S)$:
    pre: $st_C^{\mathrm{tmp}}.\{fid\}$
    $k_f \leftarrow \mathsf{getHeader}^{(C:2)}(st_C, fid, m_S)$
    $out_C.oob \leftarrow k_f$   // share file key out-of-band
    success$_C$
```

```
$\mathsf{CSS}^{(S:1)}_{\mathsf{share}}(st_S, \varepsilon, in_S, m_C)$:
    pre: $st_S.\{ows\}$
    $(m'_C, raid, fid) \leftarrow m_C$
    $m_S \leftarrow \mathsf{getHeader}^{(S:1)}(st_S, m'_C)$
    $ows[fid] \overset{\cup}{\leftarrow} \{raid\}$   // authorize receiver
    success$_S(m_S)$
```

```
$\mathsf{CSS}^{(C:1)}_{\mathsf{accept}}(st_C, \varepsilon, in_C, \varepsilon)$:
    pre: $in_C.\{fid, oob\}$ $st_C.\{sid\}$
        // $k_f$ received in $oob$
    $m_C \leftarrow \mathsf{putHeader}^{(C:1)}(st_C, fid, oob)$
    success$_C(m_C)$
```

```
$\mathsf{CSS}^{(S:1)}_{\mathsf{accept}}(st_S, \varepsilon, in_S, m_C)$:
    $\mathsf{putHeader}^{(S:1)}(st_S, m_C)$
    success$_S$
```

Figure 9: File download, upload, update, share, and accept protocols for CSS.

```
$\boxed{\mathsf{putFile}}\,\dbox{\mathsf{putHeader}}^{(C:1)}(st_C, fid,\ k_f\,\boxed{, f})$:
    $n \leftarrow\!\$\ \{0,1\}^{nl}$
    $\dbox{md \leftarrow (st_C.aid, fid, \text{“fk”})}$
    $\dbox{c_{fk} \leftarrow \mathsf{Enc}(st_C.k_{mk}, n, k_f, md)}$
    $\boxed{c_f \leftarrow \mathsf{Enc}(k_f, n, f, (fid, \text{“file”}))}$
    $m_C \leftarrow (st_C.sid, fid, n\,\dbox{, c_{fk}}\,\boxed{, c_f})$
    return $m_C$

$\boxed{\mathsf{putFile}}\,\dbox{\mathsf{putHeader}}^{(S:1)}(st_S, m_C)$:
    pre: $st_S.\{sess, ows, fks, fs\}$
    $(sid, fid, n\,\dbox{, c_{fk}}\,\boxed{, c_f}) \leftarrow m_C$
    $aid \leftarrow sess[sid]$
    if $aid \notin ows[fid]$: return $\bot$
    $\dbox{fks[(aid, fid)] \leftarrow (n, c_{fk})}$
    $\boxed{fs[fid] \leftarrow (n, c_f)}$

$\boxed{\mathsf{getFile}}\,\dbox{\mathsf{getHeader}}^{(C:1)}(st_C, fid)$:
    $m_C \leftarrow (st_C.sid, fid)$
    return $m_C$

$\boxed{\mathsf{getFile}}\,\dbox{\mathsf{getHeader}}^{(C:2)}(st_C, fid, m_S)$:
    $((n_{fk}, c_{fk})\,\boxed{, (n_f, c_f)}) \leftarrow m_S$
    $md \leftarrow (st_C.aid, fid, \text{“fk”})$
    $k_f \leftarrow \mathsf{Dec}(st_C.k_{mk}, n_{fk}, c_{fk}, md)$
    $\dbox{\textbf{return } k_f}$
    $\boxed{f \leftarrow \mathsf{Dec}(k_f, n_f, c_f, (fid, \text{“file”}))}$
    $\boxed{\textbf{return } f}$

$\boxed{\mathsf{getFile}}\,\dbox{\mathsf{getHeader}}^{(S:1)}(st_S, m_C)$:
    pre: $st_S.\{sess, ows, fks, fs\}$
    $(sid, fid) \leftarrow m_C$
    $aid \leftarrow sess[sid]$
    if $aid \notin ows[fid]$: return $\bot$
    $m_S \leftarrow (fks[(aid, fid)]\,\boxed{, fs[fid]})$
    return $m_S$
```

Figure 10: Subroutines for putting and getting header and/or files for CSS. $\boxed{\text{Solid-boxed}}$, $\dbox{\text{dashed-boxed}}$ code is included in the correspondingly marked algorithms.

prove CSS adaptively secure by guessing the subset of users which will become compromised and/or the challenge file owners. However, the loss from this naïve “complexity leveraging” technique would be expontential, because the reduction needs to guess one subset in the full power set of the set of users. Developing other proof techniques to show that CSS is adaptively secure with less loss is an interesting question for future work.

An interesting proof artifact in the confidentiality bound is our reductions to two different AEAD security notions of indistinguishability. We could equally have reduced twice to AE security, which implies both IND-CCA and IND$; we find that the extra granularity adds clarity to the proof. One AEAD security property we do not require is key robustness, which protects password-based cryptography against partitioning oracle attacks allowing an adversary to test several passwords in one interaction. In our setting, key derivation employs a random oracle and becomes interactive, so partitioning oracle attacks do not decrease adversarial interaction costs.

In the following, we let $\mathcal{Q}_{\text{ORC}}(\mathcal{A})$ denote the number of queries to oracle ORC by adversary $\mathcal{A}$. Similarly, we let $q_{\text{ORC}}(\mathcal{A})$ be the maximum number of queries *per user* made to oracle ORC by adversary $\mathcal{A}$. We use the convention that $\mathcal{Q}_{\text{PUT1}}(\mathcal{A})$ and $\mathcal{Q}_{\text{UPD1}}(\mathcal{A})$ include indirect queries made via oracle CHALL. To count only queries via oracle CHALL, we let CPUT1 and CUPD1 denote oracle CHALL with input ORC set to PUT1 and UPD1, respectively.

Theorem 6.1 (C2CConfS **of** CSS) *Let* CSS *be the cloud storage scheme defined in Figures 8–10, with building blocks specified in Section 6.1 and hash functions* H_1, H_2 *modeled as random oracles. Let* $\mathcal{A}$ *be an adversary against the* C2CConfS *security of* CSS. *Then we can construct adversaries*

$\mathcal{B}_{\mathrm{pg}}$, $\mathcal{B}_{\mathsf{prf}}$, $\mathcal{B}^{12}_{\mathsf{aead}}$ and $\mathcal{B}^{3}_{\mathsf{aead}}$ such that

$$\mathbf{Adv}^{\mathrm{C2CConfS}}_{\mathsf{CSS},n,\mathcal{PW}_n}(\mathcal{A}) \leq 2 \cdot \Big(\mathbf{Adv}^{\mathrm{PG}}_{n,\mathcal{PW}_n}(\mathcal{B}_{\mathrm{pg}}) + \mathbf{Adv}^{\mathrm{PRF}}_{\mathsf{F}}(\mathcal{B}_{\mathsf{prf}}) + 2\mathbf{Adv}^{\mathrm{IND\$}}_{\mathsf{AEAD}}(\mathcal{B}^{12}_{\mathsf{aead}})$$
$$+ \frac{(\mathcal{Q}_{\mathrm{UPD1}}(\mathcal{A})+1)^2 + (q_{\mathrm{PUT1}}(\mathcal{A}) + q_{\mathrm{ACPT1}}(\mathcal{A}))^2}{2^{nl+1}}\Big) + \mathbf{Adv}^{\mathrm{IND\text{-}CCA}}_{\mathsf{AEAD}}(\mathcal{B}^{3}_{\mathsf{aead}}).$$

The query counts of adversaries the adversaries $\mathcal{B}_{\mathrm{pg}}$, $\mathcal{B}_{\mathsf{prf}}$, $\mathcal{B}^{12}_{\mathsf{aead}}$, and $\mathcal{B}^{3}_{\mathsf{aead}}$ are bounded by the total query counts of adversary $\mathcal{A}$, as detailed in Appendix C.1. The running times of all adversaries are approximately that of $\mathcal{A}$.

Proof sketch. We give an overview over the main proof steps here; the full proof is in Appendix C.1. The proof proceeds through a sequence of games G_0–G_7, which step by step replaces the keys in the key hierarchy of honest users ($i_u \in (\mathrm{S}_{\mathsf{hon}} \setminus \mathrm{S}_{\mathsf{comp}})$) by keys generated independently at random, reducing to the relevant security properties of the respective parts of CSS. The advantage of $\mathcal{A}$ in the final game then reduces to the left-or-right IND-CCA security of the AEAD scheme used for file encryption.

Games G_0–G_3: Replacing *rw* by random. We begin with game G_0, which is equivalent to $\mathbf{G}^{\mathrm{C2CConfS}}_{\mathsf{CSS},n,\mathcal{PW}_n}(\mathcal{A})$, except that whenever a protocol uses the hash functions H_1 or H_2, the game instead queries the random oracles RO_1 or RO_2, respectively. The effect of this change is only to model the hash functions as random oracles. Hence

$$\mathbf{Adv}^{\mathrm{C2CConfS}}_{\mathsf{CSS},n,\mathcal{PW}_n}(\mathcal{A}) := 2 \cdot \Pr[\mathrm{G}_0] - 1\ . \tag{1}$$

In the first sequence of games, the goal is to move to game G_3, where the client-side output $rw \in \{0,1\}^{kl}$ from the OPRF flow in protocols $\mathsf{CSS}_{\mathsf{areg}}$ and $\mathsf{CSS}_{\mathsf{auth}}$ is replaced by an independent random string in $\{0,1\}^{kl}$ for all honest (uncompromised) users.

We claim that the change from G_0 to G_3 is indistinguishable to an adversary, except with some small probability. Formally, we want to bound $\Pr[\mathrm{G}_0] - \Pr[\mathrm{G}_3]$. We do this through a sequence of game hops over two additional games, G_1 and G_2, which allow us to show that $\Pr[\mathrm{G}_0] - \Pr[\mathrm{G}_3] \leq \Pr[\mathsf{Bad}]$, where (informally) Bad is the event that adversary $\mathcal{A}$ made a query $\mathrm{RO}_1(aid, pw)$ or $\mathrm{RO}_2(aid, pw, *)$ such that aid and pw are the account ID and password of an uncompromised user. We then construct an adversary $\mathcal{B}_{\mathrm{pg}}$ such that $\Pr[\mathsf{Bad}] \leq \mathbf{Adv}^{\mathrm{PG}}_{n,\mathcal{PW}_n}(\mathcal{A})$. Combined with standard equation rewriting, this gives

$$\Pr[\mathrm{G}_0] \leq \mathbf{Adv}^{\mathrm{PG}}_{n,\mathcal{PW}_n}(\mathcal{A}) + \Pr[\mathrm{G}_3]\ . \tag{2}$$

Games G_4–G_7: Random keys and nonce collisions. In the following hops, we apply PRF and AEAD security to ensure that keys in the key hierarchies of honest users are uniformly random and independent from anything the server sees. We also handle nonce collisions in AEAD encryptions.

First, in the hop to G_4, we replace k_{kek} and k_{mac} derived from rw with PRF F by random strings. This hop is bounded by

$$\Pr[\mathrm{G}_3] - \Pr[\mathrm{G}_4] \leq \mathbf{Adv}^{\mathrm{PRF}}_{\mathsf{F}}(\mathcal{B}_{\mathsf{prf}}). \tag{3}$$

G_5 then replaces the encrypted master keys c_{mk} of honest clients by independent random strings, simultaneously ensuring that any deviation from the protocol by the server in authentication leads to a decryption failure when the client attempts to retrieve its master key. This hop is bounded by

$$\Pr[\mathrm{G}_4] - \Pr[\mathrm{G}_5] \leq \mathbf{Adv}^{\mathrm{IND\$}}_{\mathsf{AEAD}}(\mathcal{B}^{1}_{\mathsf{aead}}). \tag{4}$$

In G_6 we handle nonce collisions in file and file key encryptions. We introduce a bad event if any two nonces for the same key collide, and in G_6 sample a guaranteed fresh nonce as replacement. The bad event, and hence hop from G_5 to G_6 is bounded (through two birthday bounds) by

$$\Pr[\mathrm{G}_5] - \Pr[\mathrm{G}_6] \leq \frac{(\mathcal{Q}_{\text{UPD1}}(\mathcal{A}) + 1)^2 + (q_{\text{PUT1}}(\mathcal{A}) + q_{\text{ACPT1}}(\mathcal{A}))^2}{2^{nl+1}}. \tag{5}$$

In the final game hop, to G_7, we replace the encrypted file keys c_{fk} of all honest users by random strings, again ensuring that only honest keys can be decrypted in get, update and share operations. This step is bounded by

$$\Pr[\mathrm{G}_6] - \Pr[\mathrm{G}_7] \leq \mathbf{Adv}^{\mathrm{IND\$}}_{\mathsf{AEAD}}(\mathcal{B}^2_{\mathsf{aead}}). \tag{6}$$

Bounding G_7**:** IND-CCA **security of file encryption.** In the last step of the proof, we bound the success probability of $\mathcal{A}$ in game G_7 by the advantage of an adversary $\mathcal{B}^3_{\mathsf{aead}}$ against the IND-CCA security of AEAD. This step requires careful checking of the trivial attack conditions, to ensure that $\mathcal{B}^3_{\mathsf{aead}}$ can properly simulate the game unless $\mathcal{A}$ performs a trivial attack. We conclude that

$$2 \cdot \Pr[\mathrm{G}_7] - 1 \leq \mathbf{Adv}^{\mathrm{IND\text{-}CCA}}_{\mathsf{AEAD}}(\mathcal{B}^3_{\mathsf{aead}}). \tag{7}$$

Merging adversaries $\mathcal{B}^1_{\mathsf{aead}}$ and $\mathcal{B}^2_{\mathsf{aead}}$ into an adversary $\mathcal{B}^{12}_{\mathsf{aead}}$ and putting Equations (1)–(7) together yields the theorem statement. □

Theorem 6.2 (C2CIntS **of** CSS) *Let* CSS *be the cloud storage scheme defined in Figures 8–10, with building blocks specified in Section 6.1 and hash functions* H_1*,* H_2 *modeled as random oracles. Let* $\mathcal{A}$ *be an adversary against the* C2CIntS *security of* CSS *making at most* p *queries to oracle* PUT1*. We construct adversaries* $\mathcal{B}_{\mathrm{pg}}$*,* $\mathcal{B}_{\mathsf{prf}}$*,* $\mathcal{B}^1_{\mathsf{aead}}$*,* $\mathcal{B}^2_{\mathsf{aead}}$*,* $\mathcal{B}^3_{\mathsf{aead}}$ *and* $\mathcal{B}^4_{\mathsf{aead}}$ *such that*

$$\begin{aligned}\mathbf{Adv}^{\mathrm{C2CIntS}}_{\mathsf{CSS},n,\mathcal{PW}_n}(\mathcal{A}) &\leq \mathbf{Adv}^{\mathrm{PG}}_{n,\mathcal{PW}_n}(\mathcal{B}_{\mathrm{pg}}) + \mathbf{Adv}^{\mathrm{PRF}}_{\mathsf{F}}(\mathcal{B}_{\mathsf{prf}}) + \mathbf{Adv}^{\mathrm{IND\$}}_{\mathsf{AEAD}}(\mathcal{B}^1_{\mathsf{aead}}) \\ &+ \frac{(\mathcal{Q}_{\text{UPD1}}(\mathcal{A}) + 1)^2 + (q_{\text{PUT1}}(\mathcal{A}) + q_{\text{ACPT1}}(\mathcal{A}))^2}{2^{nl+1}} + \mathbf{Adv}^{\mathrm{INT\text{-}CTXT}}_{\mathsf{AEAD}}(\mathcal{B}^2_{\mathsf{aead}}) \\ &+ \mathrm{p} \cdot \left(\mathbf{Adv}^{\mathrm{IND\$}}_{\mathsf{AEAD}}(\mathcal{B}^3_{\mathsf{aead}}) + \mathbf{Adv}^{\mathrm{INT\text{-}CTXT}}_{\mathsf{AEAD}}(\mathcal{B}^4_{\mathsf{aead}})\right).\end{aligned}$$

The query counts for adversaries $\mathcal{B}_{\mathrm{pg}}$*,* $\mathcal{B}_{\mathsf{prf}}$ *and* $\mathcal{B}^1_{\mathsf{aead}}$ *are as in the proof of Theorem 6.1, the others are described in Appendix C.2. The running times of all adversaries are approximately that of* $\mathcal{A}$*.*

The proof is analogous to that of Theorem 6.1, except for the last few steps. That is, the proof proceeds through a sequence of games G_0–G_8, which step by step replaces the keys in the key hierarchy of honest users by keys generated independently at random, reducing to password-guessing, PRF and AEAD advantages along the way. In contrast to in the confidentiality proof, these hops to not incur a factor 2 in the bound, due to the advantage definition for C2CInt (Definition 4.3). Additionally, there is an intermediate hop to guess the file ID for which $\mathcal{A}$ performs its forgery attempts, incurring a loss by a factor p. The advantage of $\mathcal{A}$ in the final game reduces to the INT-CTXT security of the AEAD scheme used for file encryption. The full details of the proof are in Appendix C.2.

6.5 Limitations and Extensions of CSS

CSS is meant a as a proof-of-concept cloud storage scheme, illustrating our formalism and security model. For the scope of this work, the design focuses on core cloud storage operations and security mechanisms. We had to analyse CSS in the weaker selective security game to avoid commitment issues, leaving open the challenging task of achieving adaptive security for a scheme relying on standard cryptographic components. Moreover, in a real-world implementation, one would have to consider further functional requirements and additional security mechanisms, e.g., to strengthen the system's behavior against external adversaries under an honest cloud service. In the following, we discuss limitations of our design in this regards and possible extensions.

ADAPTIVE SECURITY. Plausibly, the flexible password-based encryption (FPBE) approach by Bellare and Shea described in [12] could be used to achieve adaptive security when applied to cloud storage protocols. FPBE's flat key hierarchy omits the key-encrypting ciphertexts that would cause issues in an adaptive analysis of CSS. However, there are concerns about the practicality of FPBE: Bellare and Shea omit any consideration of file-sharing. Without decoupling file keys from the password through key wrapping, it is unclear how to efficiently share files if all of a user's keys are derived from their password (as in FPBE). Without naïve file replication, users sharing a file would need to derive the same key but from different passwords. Moreover, caching the password in clients (as it is needed for every operation) may be an operational security concern. Additionally, directly deriving file keys from user passwords makes password rotation very expensive: if a user wants to change their password, they have to re-encrypt all of their files.

However, FPBE is only one avenue towards achieving adaptive security. It might well be the case that CSS is adaptively secure; the challenge lies in constructing a proof which does not suffer an exponential loss (from naïve complexity leveraging) when addressing the commitment issues. We leave it as an open problem to prove stronger security of CSS, or to construct an adaptively-secure cloud storage scheme which avoids the commitment issues while still supporting all core functionality and not compromising on practicality.

ENFORCING UNIQUE IDENTIFIERS. In our security model, we assume unique account identifiers for honest users (e.g., email accounts) and file identifiers (e.g., by prefixing account IDs). This allows us to easily define what constitutes a "user" and a "file": they correspond to an account ID and a file ID, respectively. As a consequence, we also rule out trivial collision attacks by a malicious cloud provider. Our CSS protocol hence, for reduced complexity in presentation, does not check for colliding identifiers (e.g., in $\mathsf{CSS}_{\mathsf{areg}}$); in an actual implementation, an honest server would reasonably check for those to prevent malicious clients from overwriting data of other (honest) clients. Providing provably secure schemes under weaker assumptions on the uniqueness of account or file identifiers is an interesting direction for future work.

PASSWORD-AUTHENTICATED CHANNEL ESTABLISHMENT. Instead of relying on a secure channel protocol (like TLS), a protocol might expand the OPRF usage to a full password-based key exchange, e.g., employing the OPAQUE [37] protocol, possibly in combination with TLS [31].

7 Conclusion and Future Work

The deployment of E2EE systems began over a decade ago and brought major security improvements to areas like secure messaging (e.g., the Signal protocol) and browsing (TLS). Despite cloud storage seeing comparable adoption, these advancements have not translated to this area, as recent attacks on the largest E2EE cloud providers [6, 30, 3, 2] have shown. The vulnerabilities in the end-to-end security of these cloud storages are at least in part due to the complexity of the setting – which, when taking into consideration features like file sharing, is at least as intricate as key exchange – leaving it easy for providers to introduce subtle mistakes in protocols. These mistakes motivate the need for a strong cryptographic foundation for E2EE cloud storage.

In this paper, we initiated the formal study of E2EE cloud storage. As central contributions towards this, we introduced a syntax capturing the core functionality of cloud storage, as well as game-based security notions for the desired confidentiality and integrity against a malicious service provider. We put our model to test by both capturing recent attacks on MEGA as well as proving secure our own E2EE cloud storage scheme CSS.

In order to tame the complexity of the problem, we had to make some simplifying assumptions, leaving many opportunities for future work, including both model and protocol extensions. (See Sections 4.4 and 6.5, respectively.)

A natural next step is to consider Client-to-Server (C2S) security of CSS, i.e., security against external adversaries. We have sketched how to adapt our existing models to this C2S setting, and we expect the C2S security of CSS to follow smoothly. Working with an honest server and assuming a secure channel between clients and server allows C2S to avoid many complexities (e.g. malicious choices of the OPRF keys, offline password brute-forcing, etc).

Second, we had to analyse CSS in the weaker selective security setting to avoid commitment issues, leaving open the challenging task of achieving adaptive security for a scheme relying on standard cryptographic components. Plausibly, the flexible password-based encryption (FPBE) approach described in [12] could be used to achieve this, because FPBE's flat key hierarchy omits the key-encrypting ciphertexts that would cause issues in an adaptive analysis of CSS. However, FPBE omits any consideration of file-sharing, and it not clear how to efficiently achieve this if all of a user's keys are derived from their password (as in FPBE), other than by naive file replication.

Third, the core functionality considered in this paper could be extended – at the cost of higher complexity – with extra features such as file "unsharing" and deletion, password recovery, session deauthentication (and ensuing security notions for adversaries with access to expired sessions), and account deregistration.

Finally, many more advanced security properties could be considered. For example, it is an open question how post-compromise and forward security apply to the cloud storage setting, and how much is achievable; key rotation has been considered in prior works [36], as has fine-grained forward security via puncturing techniques [5, 53], but not in combination with the full feature set we target. Furthermore, CSS does not attempt to protect against roll-back attacks, nor provide advanced metadata-hiding, cf. [22, 21].

This paper introduces formally analyzed protocols instead of relying on *ad hoc* and/or proprietary designs. We hope that users will eventually enjoy the benefits of a single, well-analyzed scheme with provable security guarantees, akin to E2EE communication today. Standardization of such a scheme would resolve the long-standing issues that untrusted cloud providers need to be trusted with the design of a secure system and serving benign client code. A standardized scheme could have independent client implementations (i.e., not under the control of the potentially malicious provider) and even lead to interoperability between cloud storage providers. Our work is a first step towards bringing modern cryptographic guarantees to cloud storage.

Acknowledgments

We thank Mihir Bellare for his ideas and contributions in the early phases of the project, in particular on formal syntax and definitions. We also thank Dennis Hofheinz for valuable discussions, as well as all anonymous reviewers for their comments and feedback.

The research of Miro Haller was supported by an Amazon Research Award Fall 2023. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not reflect the views of Amazon.

References

[1] pCloud International AG. pCloud – the most secure cloud storage. https://www.pcloud.com/. (Cited on page 3.)

[2] Martin R. Albrecht, Matilda Backendal, Daniele Coppola, and Kenneth G. Paterson. Share with care: Breaking E2EE in Nextcloud. Cryptology ePrint Archive, Paper 2024/546, 2024. https://eprint.iacr.org/2024/546. (Cited on page 3, 4, 36.)

[3] Martin R. Albrecht, Miro Haller, Lenka Mareková, and Kenneth G. Paterson. Caveat implementor! Key recovery attacks on MEGA. In Carmit Hazay and Martijn Stam, editors, *EUROCRYPT 2023, Part V*, volume 14008 of *LNCS*, pages 190–218. Springer, Heidelberg, April 2023. (Cited on page 3, 4, 25, 26, 27, 36, 42, 47, 48, 49, 50.)

[4] Apple. Advanced data protection for iCloud. https://support.apple.com/guide/security/advanced-data-protection-for-icloud-sec973254c5f/web, Dec 2022. visited Feb 10, 2024. (Cited on page 3, 6.)

[5] Matilda Backendal, Felix Günther, and Kenneth G. Paterson. Puncturable key wrapping and its applications. In Shweta Agrawal and Dongdai Lin, editors, *ASIACRYPT 2022, Part II*, volume 13792 of *LNCS*, pages 651–681. Springer, Heidelberg, December 2022. (Cited on page 7, 25, 37.)

[6] Matilda Backendal, Miro Haller, and Kenneth G. Paterson. MEGA: Malleable encryption goes awry. In *2023 IEEE Symposium on Security and Privacy*, pages 146–163. IEEE Computer Society Press, May 2023. (Cited on page 3, 4, 25, 26, 27, 36, 42, 44, 46, 47, 48, 49, 50.)

[7] Ali Bagherzandi, Stanislaw Jarecki, Nitesh Saxena, and Yanbin Lu. Password-protected secret sharing. In Yan Chen, George Danezis, and Vitaly Shmatikov, editors, *ACM CCS 2011*, pages 433–444. ACM Press, October 2011. (Cited on page 22.)

[8] Mihir Bellare, Ran Canetti, and Hugo Krawczyk. Pseudorandom functions revisited: The cascade construction and its concrete security. In *37th FOCS*, pages 514–523. IEEE Computer Society Press, October 1996. (Cited on page 50.)

[9] Mihir Bellare, Thomas Ristenpart, and Stefano Tessaro. Multi-instance security and its application to password-based cryptography. In Reihaneh Safavi-Naini and Ran Canetti, editors, *CRYPTO 2012*, volume 7417 of *LNCS*, pages 312–329. Springer, Heidelberg, August 2012. (Cited on page 7, 15.)

[10] Mihir Bellare and Phillip Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In Dorothy E. Denning, Raymond Pyle, Ravi Ganesan, Ravi S. Sandhu, and Victoria Ashby, editors, *ACM CCS 93*, pages 62–73. ACM Press, November 1993. (Cited on page 8.)

[11] Mihir Bellare and Phillip Rogaway. The security of triple encryption and a framework for code-based game-playing proofs. In Serge Vaudenay, editor, *EUROCRYPT 2006*, volume 4004 of *LNCS*, pages 409–426. Springer, Heidelberg, May / June 2006. (Cited on page 8, 54, 58, 64.)

[12] Mihir Bellare and Laura Shea. Flexible password-based encryption: Securing cloud storage and provably resisting partitioning-oracle attacks. In Mike Rosulek, editor, *CT-RSA 2023*,

volume 13871 of *LNCS*, pages 594–621. Springer, Heidelberg, April 2023. (Cited on page 7, 15, 29, 36, 37.)

[13] Alysson Bessani, Miguel Correia, Bruno Quaresma, Fernando André, and Paulo Sousa. Depsky: Dependable and secure storage in a cloud-of-clouds. *ACM Trans. Storage*, 9(4), nov 2013. (Cited on page 7.)

[14] Jan Camenisch, Anna Lysyanskaya, and Gregory Neven. Practical yet universally composable two-server password-authenticated secret sharing. In Ting Yu, George Danezis, and Virgil D. Gligor, editors, *ACM CCS 2012*, pages 525–536. ACM Press, October 2012. (Cited on page 22.)

[15] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *42nd FOCS*, pages 136–145. IEEE Computer Society Press, October 2001. (Cited on page 5, 22.)

[16] Ran Canetti, Palak Jain, Marika Swanberg, and Mayank Varia. Universally composable end-to-end secure messaging. In Yevgeniy Dodis and Thomas Shrimpton, editors, *CRYPTO 2022, Part II*, volume 13508 of *LNCS*, pages 3–33. Springer, Heidelberg, August 2022. (Cited on page 22.)

[17] Ran Canetti and Hugo Krawczyk. Universally composable notions of key exchange and secure channels. In Lars R. Knudsen, editor, *EUROCRYPT 2002*, volume 2332 of *LNCS*, pages 337–351. Springer, Heidelberg, April / May 2002. (Cited on page 5, 22.)

[18] Ran Canetti, Eyal Kushilevitz, and Yehuda Lindell. On the limitations of universally composable two-party computation without set-up assumptions. *Journal of Cryptology*, 19(2):135–167, April 2006. (Cited on page 5.)

[19] Sílvia Casacuberta, Julia Hesse, and Anja Lehmann. SoK: Oblivious pseudorandom functions. In *7th IEEE European Symposium on Security and Privacy, EuroS&P 2022*, pages 625–646. IEEE, 2022. (Cited on page 7.)

[20] Long Chen, Ya-Nan Li, Qiang Tang, and Moti Yung. End-to-same-end encryption: Modularly augmenting an app with an efficient, portable, and blind cloud storage. In Kevin R. B. Butler and Kurt Thomas, editors, *USENIX Security 2022*, pages 2353–2370. USENIX Association, August 2022. (Cited on page 7.)

[21] Weikeng Chen, Thang Hoang, Jorge Guajardo, and Attila A. Yavuz. Titanium: A metadata-hiding file-sharing system with malicious security. In *29th Annual Network and Distributed System Security Symposium, NDSS 2022, San Diego, California, USA, April 24-28, 2022*. The Internet Society, 2022. (Cited on page 7, 25, 37.)

[22] Weikeng Chen and Raluca Ada Popa. Metal: A metadata-hiding file-sharing system. In *NDSS 2020*. The Internet Society, February 2020. (Cited on page 7, 25, 37.)

[23] Poulami Das, Julia Hesse, and Anja Lehmann. DPaSE: Distributed password-authenticated symmetric-key encryption, or how to get many keys from one password. In Yuji Suga, Kouichi Sakurai, Xuhua Ding, and Kazue Sako, editors, *ASIACCS 22*, pages 682–696. ACM Press, May / June 2022. (Cited on page 7.)

[24] Gareth T. Davies, Sebastian H. Faller, Kai Gellert, Tobias Handirk, Julia Hesse, Máté Horváth, and Tibor Jager. Security analysis of the WhatsApp end-to-end encrypted backup protocol.

In Helena Handschuh and Anna Lysyanskaya, editors, *CRYPTO 2023, Part IV*, volume 14084 of *LNCS*, pages 330–361. Springer, Heidelberg, August 2023. (Cited on page 6.)

[25] Adam Everspaugh, Rahul Chatterjee, Samuel Scott, Ari Juels, and Thomas Ristenpart. The pythia PRF service. In Jaeyeon Jung and Thorsten Holz, editors, *USENIX Security 2015*, pages 547–562. USENIX Association, August 2015. (Cited on page 7.)

[26] Michael J. Freedman, Yuval Ishai, Benny Pinkas, and Omer Reingold. Keyword search and oblivious pseudorandom functions. In Joe Kilian, editor, *TCC 2005*, volume 3378 of *LNCS*, pages 303–324. Springer, Heidelberg, February 2005. (Cited on page 27.)

[27] Eu-Jin Goh, Hovav Shacham, Nagendra Modadugu, and Dan Boneh. SiRiUS: Securing remote untrusted storage. In *NDSS 2003*. The Internet Society, February 2003. (Cited on page 7.)

[28] Miro Haller. Cloud storage systems: From bad practice to practical attacks. Master's thesis, ETH Zurich, Jun 2022. (Cited on page 42.)

[29] Sven Hebrok, Simon Nachtigall, Marcel Maehren, Nurullah Erinola, Robert Merget, Juraj Somorovsky, and Jörg Schwenk. We really need to talk about session tickets: A Large-Scale analysis of cryptographic dangers with TLS session tickets. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 4877–4894, Anaheim, CA, August 2023. USENIX Association. (Cited on page 25.)

[30] Nadia Heninger and Keegan Ryan. The hidden number problem with small unknown multipliers: Cryptanalyzing MEGA in six queries and other applications. In Alexandra Boldyreva and Vladimir Kolesnikov, editors, *PKC 2023, Part I*, volume 13940 of *LNCS*, pages 147–176. Springer, Heidelberg, May 2023. (Cited on page 3, 4, 25, 26, 27, 36, 47, 49.)

[31] Julia Hesse, Stanislaw Jarecki, Hugo Krawczyk, and Christopher Wood. Password-authenticated TLS via OPAQUE and post-handshake authentication. In Carmit Hazay and Martijn Stam, editors, *EUROCRYPT 2023, Part V*, volume 14008 of *LNCS*, pages 98–127. Springer, Heidelberg, April 2023. (Cited on page 36.)

[32] Stanislaw Jarecki, Aggelos Kiayias, and Hugo Krawczyk. Round-optimal password-protected secret sharing and T-PAKE in the password-only model. In Palash Sarkar and Tetsu Iwata, editors, *ASIACRYPT 2014, Part II*, volume 8874 of *LNCS*, pages 233–253. Springer, Heidelberg, December 2014. (Cited on page 4, 7, 27, 29.)

[33] Stanislaw Jarecki, Aggelos Kiayias, and Hugo Krawczyk. Round-optimal password-protected secret sharing and t-pake in the password-only model. In Palash Sarkar and Tetsu Iwata, editors, *Advances in Cryptology – ASIACRYPT 2014*, pages 233–253, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg. (Cited on page 22.)

[34] Stanislaw Jarecki, Aggelos Kiayias, Hugo Krawczyk, and Jiayu Xu. Highly-efficient and composable password-protected secret sharing (or: How to protect your bitcoin wallet online). In *IEEE European Symposium on Security and Privacy, EuroS&P 2016*, pages 276–291. IEEE, 2016. (Cited on page 4, 27, 29.)

[35] Stanislaw Jarecki, Aggelos Kiayias, Hugo Krawczyk, and Jiayu Xu. TOPPSS: Cost-minimal password-protected secret sharing based on threshold OPRF. In Dieter Gollmann, Atsuko Miyaji, and Hiroaki Kikuchi, editors, *ACNS 17*, volume 10355 of *LNCS*, pages 39–58. Springer, Heidelberg, July 2017. (Cited on page 22.)

[36] Stanislaw Jarecki, Hugo Krawczyk, and Jason K. Resch. Updatable oblivious key management for storage systems. In Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, *ACM CCS 2019*, pages 379–393. ACM Press, November 2019. (Cited on page 7, 25, 37.)

[37] Stanislaw Jarecki, Hugo Krawczyk, and Jiayu Xu. OPAQUE: An asymmetric PAKE protocol secure against pre-computation attacks. In Jesper Buus Nielsen and Vincent Rijmen, editors, *EUROCRYPT 2018, Part III*, volume 10822 of *LNCS*, pages 456–486. Springer, Heidelberg, April / May 2018. (Cited on page 7, 36.)

[38] Russell W. F. Lai, Christoph Egger, Manuel Reinert, Sherman S. M. Chow, Matteo Maffei, and Dominique Schröder. Simple password-hardened encryption services. In William Enck and Adrienne Porter Felt, editors, *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*, pages 1405–1421. USENIX Association, 2018. (Cited on page 7.)

[39] Russell W. F. Lai, Christoph Egger, Dominique Schröder, and Sherman S. M. Chow. Phoenix: Rebirth of a cryptographic password-hardening service. In Engin Kirda and Thomas Ristenpart, editors, *USENIX Security 2017*, pages 899–916. USENIX Association, August 2017. (Cited on page 7.)

[40] MEGA Limited. MEGA – about us. https://mega.io/about. visited Jan 25, 2024. (Cited on page 3, 25.)

[41] MEGA Limited. MEGA security white paper – third edition. https://mega.nz/SecurityWhitepaper.pdf, Jun 2022. (Cited on page 42.)

[42] ID Cloud Services LTD. Icedrive – next-generation cloud storage. https://icedrive.net/. (Cited on page 3.)

[43] Seafile Ltd. Seafile – open source file sync and share software. https://www.seafile.com/en/home/. (Cited on page 3.)

[44] Team MEGA. MEGA security update. https://blog.mega.io/mega-security-update, Jun 2022. visited Feb 9, 2024. (Cited on page 25.)

[45] Mega Limited. MEGA web client. https://github.com/meganz/webclient. visited Feb 6, 2024. (Cited on page 42.)

[46] Kathleen Moriarty, Burt Kaliski, and Andreas Rusch. PKCS #5: Password-Based Cryptography Specification Version 2.1. RFC 8018, January 2017. (Cited on page 42.)

[47] Moni Naor and Omer Reingold. Number-theoretic constructions of efficient pseudo-random functions. In *38th FOCS*, pages 458–467. IEEE Computer Society Press, October 1997. (Cited on page 27.)

[48] Nextcloud. End-to-End Encryption Design. https://nextcloud.com/c/uploads/2022/03/Nextcloud-end-to-end-encryption-Whitepaper.pdf, September 2017. (Cited on page 3, 15.)

[49] OnePassword. 1Password Security Design (Release v0.4.6). https://1passwordstatic.com/files/security/1password-white-paper.pdf, October 2023. (Cited on page 15.)

[50] Raluca Ada Popa, Emily Stark, Steven Valdez, Jonas Helfer, Nickolai Zeldovich, and Hari Balakrishnan. Building web applications on top of encrypted data using mylar. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 157–172, Seattle, WA, April 2014. USENIX Association. (Cited on page 7.)

[51] Phillip Rogaway. Authenticated-encryption with associated-data. In Vijayalakshmi Atluri, editor, *ACM CCS 2002*, pages 98–107. ACM Press, November 2002. (Cited on page 50.)

[52] Tresorit. End-to-end encrypted cloud storage for businesses. https://tresorit.com/. (Cited on page 3.)

[53] Nirvan Tyagi, Muhammad Haris Mughees, Thomas Ristenpart, and Ian Miers. BurnBox: Self-revocable encryption in a world of compelled access. In William Enck and Adrienne Porter Felt, editors, *USENIX Security 2018*, pages 445–461. USENIX Association, August 2018. (Cited on page 7, 25, 37.)

[54] WhatsApp. End-to-end encrypted backups on WhatsApp. https://blog.whatsapp.com/end-to-end-encrypted-backups-on-whatsapp. visited Feb 10, 2024. (Cited on page 6.)

A Details of Formalizing (the Attacks on) MEGA

This section formalizes a simplified version of the cryptographic protocol implemented by MEGA in our syntax. We use MEGA's whitepaper [41], their client source code [45], and previous analyses of their system [6, 28] as sources to aggregate a description of their protocol. The server code is closed-source and the official documentation only describes the server API. Thus, the server-side in our protocol is inferred from the cryptographic operations on the client and may differ from the actual implementation. For simplicity, we focus on cryptographic operations and omit various input validation and error messages. While this is sufficient to capture the attacks from [6] on MEGA, we remark that modeling the error messages *would be essential* for a full security proof. For instance, Albrecht et al. [3] exploited error messages in a now-patched client implementation of MEGA to build a plaintext recovery attack for a malicious server.

A.1 Formalizing MEGA as a Cloud Storage Scheme

We define the MEGA cloud storage scheme M as a tuple of seven protocols $\mathsf{M} = (\mathsf{M}_{\mathsf{areg}}, \mathsf{M}_{\mathsf{auth}}, \mathsf{M}_{\mathsf{put}}, \mathsf{M}_{\mathsf{upd}}, \mathsf{M}_{\mathsf{get}}, \mathsf{M}_{\mathsf{share}}, \mathsf{M}_{\mathsf{accept}})$ to register accounts, authenticate clients, as well as upload, update, download, share, and accept files.

The following simplified descriptions of MEGA build on the symmetric encryption schemes AES-ECB, AES-CBC, and AES-CCM, as well as the asymmetric RSA encryption scheme. MEGA further uses the password-based key derivation function PBKDF [46] and a collision-resistant hash function H.

Our modeling focuses on the version of MEGA that was vulnerable to the attacks by Backendal et al. [6]. To this end, we omit functionality such as ephemeral accounts, two-factor authentication, account recovery, remote session deauthentication, and public sharing links (see MEGA whitepaper Sections 3.3 to 3.6 and 5 [41]). We describe the core protocols of MEGA in the remainder of this section, before discussing attacks.

Account registration $\mathsf{M}_{\mathsf{areg}}$ (Figure 11). The client first picks a random master key k_m and client nonce n_C. It derives the encryption key k_e and authentication key k_a, from PBKDF applied to the concatenation of the user password pw and a salt s. The salt is the hash of the nonce n_C

```
client: $\mathsf{M}_{\mathsf{areg}}^{(C:1)}(\varepsilon, \varepsilon, in_C, \varepsilon)$:
pre: $in_C.\{aid, pw, tk\}$
$k_m \leftarrow\$ \{0,1\}^{128}$ ; $n_C \leftarrow\$ \{0,1\}^{128}$
$s \leftarrow \mathrm{H}(const \,\|\, n_C)$
$k_e \,\|\, k_a \leftarrow \mathrm{PBKDF}(s, pw)$
$h_a \leftarrow \mathrm{H}(k_a)$
$c_m \leftarrow \text{AES-ECB.}\textsc{Enc}(k_e, k_m)$
$(pk, sk) \leftarrow\$ \text{RSA.}\textsc{Gen}(2048)$
$c_{rsa} \leftarrow \text{AES-ECB.}\textsc{Enc}(k_m, sk)$
$m_C \leftarrow (aid, tk, n_C, c_m, h_a, pk, c_{rsa})$
return $(\varepsilon, \varepsilon, \varepsilon, m_C)$

$\mathsf{M}_{\mathsf{areg}}^{(C:2)}(\varepsilon, st_C^{\mathrm{tmp}}, m_S)$:
if $m_S = \bot$: fail$_C$ else: success$_C$
```

```
server: $\mathsf{M}_{\mathsf{areg}}^{(S:1)}(st_S, \varepsilon, in_S, m_C)$:
pre: $st_S.\{tokens, acc\}$
$(aid, tk, n_C, c_m, h_a, pk, c_{rsa}) \leftarrow m_C$
if $(aid, tk) \notin tokens$: fail$_S$
$acc[aid] \leftarrow (n_C, c_m, h_a, pk, c_{rsa})$
success$_S$
```

Figure 11: MEGA's account registration protocol $\mathsf{M}_{\mathsf{areg}}$.

concatenated to a constant *const*. Furthermore, the client generates the RSA secret key *sk* and public key *pk*.[9] The client then sends n_C, k_m encrypted under k_e using AES-ECB, *pk*, c_{rsa}, and the hash of k_a to the server.

We do not model the preliminary account registration step where a user provides their email address *aid* and the server verifies that address by emailing a randomly-generated token *tk*.[10] The cryptographic account registration starts when the user already received *tk* – inputting it in in_C to the client – and returns to permanently store their encrypted key material. In $\mathsf{M}_{\mathsf{areg}}^{(S:1)}$, if the account ID *aid* and token *tk* sent by the client matches the ones stored in $st_S.tokens$, the server considers account registration to be successful and stores (n_C, c_m, h_a) in persistent state.

Authentication $\mathsf{M}_{\mathsf{auth}}$ (Figure 12). To authenticate, a client sends its email address *aid*, entered by the user in in_C, to the server. If *aid* is stored in the user table $st_S.acc$, the server answers with the correct salt. As a countermeasure against email enumeration, the server generates a random salt for non-existent users.

The client uses the salt to derive the encryption and authentication keys, k_e and k_a, respectively. It sends k_a in plaintext to the server. If the hash of k_a matches h_a (stored during registration), the server responds with the encrypted master key c_m and the (half-encrypted) RSA key pair (pk, c_{rsa}). Moreover, the server sends the randomized session ID $in_S.sid$ (picked by the server in the bigger context of the protocol) encrypted for the RSA public key *pk* of the user that tries to authenticate.

The client recovers its key material (k_m and *sk*) from the ciphertexts sent by the server and uses them to decrypt the session ID *sid*. It sends *sid* back to the server, which considers a correct value as proof that the client successfully recovered its key material. Additionally, the client stores k_m, *sk*, and *sid* in the persistent client session state st_C to enable file operations.

Upload $\mathsf{M}_{\mathsf{put}}$ (Figure 13). For every file, the MEGA client picks a new file key k_f and nonce n_f. It then encrypts the file $in_C.f$ with a custom authenticated encryption algorithm that we call MFILE

[9] We deviate slightly from MEGA's protocol actual registration protocol here for the sake of simplicity. In the real system, the key material is only generated after the first successful login.

[10] Formalizing this would require a syntax extension to allow individual client-side inputs per protocol-step to capture the email out-of-band channel. (See "Protocol inputs" in Section 4.4.) Since the generated token in MEGA does not seem to depend on client input, we opt for a simplified representation taking the token as initial client input.

```
client: $\mathsf{M}_{\mathsf{auth}}^{(C:1)}(st_C, \varepsilon, in_C, \varepsilon)$:
    pre: $in_C.\{aid, pw\}$
    $st_C^{\mathrm{tmp}}.pw \leftarrow pw$
    return $(st_C, st_C^{\mathrm{tmp}}, \varepsilon, aid)$

$\mathsf{M}_{\mathsf{auth}}^{(C:2)}(st_C, st_C^{\mathrm{tmp}}, m_S)$:
    pre: $st_C^{\mathrm{tmp}}.\{pw\}$
    $k_e \,\|\, k_a \leftarrow \mathrm{PBKDF}(m_S, pw)$
    $st_C^{\mathrm{tmp}}.k_e \leftarrow k_e$
    return $(st_C, st_C^{\mathrm{tmp}}, \varepsilon, k_a)$

$\mathsf{M}_{\mathsf{auth}}^{(C:3)}(st_C, st_C^{\mathrm{tmp}}, m_S)$:
    pre: $st_C^{\mathrm{tmp}}.\{k_e\}$
    $(c_m, (pk, c_{rsa}), c_{sid}) \leftarrow m_S$
    $k_m \leftarrow \text{AES-ECB.}\textsc{Dec}(k_e, c_m)$
    $sk \leftarrow \text{AES-ECB.}\textsc{Dec}(k_m, c_{rsa})$
    $st_C.sid \leftarrow \text{RSA.}\textsc{Dec}(sk, c_{sid})$
    $st_C.k_m \leftarrow k_m$
    $st_C.pk \leftarrow pk$ ; $st_C.sk \leftarrow sk$
    return $(st_C, st_C^{\mathrm{tmp}}, out_C, st_C.sid)$

$\mathsf{M}_{\mathsf{auth}}^{(C:4)}(st_C, st_C^{\mathrm{tmp}}, m_S)$:
    if $m_S = \bot$: $\mathbf{fail}_C$ else: $\mathbf{success}_C$
```

```
server: $\mathsf{M}_{\mathsf{auth}}^{(S:1)}(st_S, \varepsilon, in_S, m_C)$:
    pre: $st_S.\{acc\}$, $in_S.\{sid\}$
    $st_S^{\mathrm{tmp}}.sid \leftarrow sid$
    if $m_C \in acc$:
        $st_S^{\mathrm{tmp}}.aid \leftarrow m_C$
        $(n_C, c_m, h_a, pk, c_{rsa}) \leftarrow acc[m_C]$
        $s \leftarrow \mathrm{H}(const \,\|\, n_C)$
    else:
        $s \leftarrow \mathrm{H}(m_C \,\|\, const' \,\|\, st_S.n_S)$
    return $(st_S, st_S^{\mathrm{tmp}}, \varepsilon, s)$

$\mathsf{M}_{\mathsf{auth}}^{(S:2)}(st_S, st_S^{\mathrm{tmp}}, m_C)$:
    pre: $st_S.\{acc\}$, $st_S^{\mathrm{tmp}}.\{aid, sid\}$
    $(n_C, c_m, h_a, pk, c_{rsa}) \leftarrow acc[aid]$
    if $\mathrm{H}(m_C) \neq h_a$: $\mathbf{fail}_S$
    $c_{sid} \leftarrow \text{RSA.}\textsc{Enc}(pk, sid)$
    $m_S \leftarrow (c_m, pk, c_{rsa}, c_{sid})$
    return $(st_S, st_S^{\mathrm{tmp}}, \varepsilon, m_S)$

$\mathsf{M}_{\mathsf{auth}}^{(S:3)}(st_S, st_S^{\mathrm{tmp}}, m_C)$:
    pre: $st_S.\{sess\}$, $st_S^{\mathrm{tmp}}.\{sid\}$
    if $m_C \neq sid$: $\mathbf{fail}_S$
    $sess[sid] \leftarrow st_S^{\mathrm{tmp}}.aid$
    $\mathbf{success}_S$
```

Figure 12: MEGA's authentication protocol $\mathsf{M}_{\mathsf{auth}}$.

which returns the file ciphertext c_f and authentication tag τ_f.[11] The client encrypts the file key material d_{fkm} – consisting of file key, nonce, and tag – with AES-ECB using the master key k_m. The client metadata d_{md}, including file size and parent handle for the file's position in the file tree, is encrypted with AES-CBC using the file key k_f. Finally, the client sends the session ID *sid*, file ID *fid*, and ciphertexts for the file key material, metadata, and file (c_{fkm}, c_{md}, and c_f, respectively) to the server.

For a valid session ID, the server persistently stores the file ID *fid*, encrypted key material c_{fkm}, and metadata c_{md} in the file tree of the user (identified by their email *aid*).[12] Furthermore, the file ciphertext c_f is also stored in st_S under the user's files.

Update $\mathsf{M}_{\mathsf{upd}}$ (Figure 13). Updating files is almost identical to $\mathsf{M}_{\mathsf{put}}$, except that it reuses the file key k_f and nonce n_f picked during the first upload. These values come from a file tree $st_C.tree$ that is cached in the client state and updated on $\mathsf{M}_{\mathsf{get}}$ operations (described next).

Download $\mathsf{M}_{\mathsf{get}}$ (Figure 14). The client first downloads the file tree *tree* containing the encrypted file key material c_{fkm} and metadata c_{md} for every file and stores it in the client state st_C (e.g., to be used by $\mathsf{M}_{\mathsf{upd}}$).[13] Note that fetching *tree* requires knowledge of the session token *sid*; the

[11]We abstract away the details of the file encryption as they do not matter for our purposes. Figure 2–4 of Backendal et al. [6] describes the chunkwise file encryption using a variant of AES-CCM performed by MFILE.

[12]The server data structures are proprietary and, thus, unknown to us. Our abstractions $st_S.tree$ and $st_S.files$ are merely convenient to express MEGA's protocols.

[13]In practice, the client regularly polls for updates of this file tree, independent of $\mathsf{M}_{\mathsf{get}}$. For the purposes of our security analysis, this is equivalent to only fetching updates for the file tree on a file download request.

```
client: $\boxed{\mathsf{M}_{\mathsf{put}}^{(C:1)}}$ $\mathsf{M}_{\mathsf{upd}}^{(C:1)}$ $(st_C, \varepsilon, in_C, \varepsilon)$:
    pre: $in_C.\{fid, f\} in_C.\{md\}$
    $\boxed{k_f \leftarrow_\$ \{0,1\}^{128}\ ;\ n_f \leftarrow_\$ \{0,1\}^{64}}$
    $(\cdot, (k_f, n_f, \cdot)) \leftarrow st_C.tree[fid]$
    $(c_f, \tau_f) \leftarrow E^{\text{MFILE}}(k_f, n_f, f)$
    $d_{fkm} \leftarrow (k_f, n_f, \tau_f)$
    $c_{fkm} \leftarrow$ AES-ECB.ENC$(st_C.k_{mk}, d_{fkm})$
    $c_{md} \leftarrow$ AES-CBC.ENC$(k_f, md)$
    $m_C \leftarrow (st_C.sid, fid, c_{fkm}, c_{md}, c_f)$
    return $(st_C, \varepsilon, \varepsilon, m_C)$

$\boxed{\mathsf{M}_{\mathsf{put}}^{(C:2)}}$ $\mathsf{M}_{\mathsf{upd}}^{(C:2)}$ $(st_C, st_C^{\text{tmp}}, m_S)$:
    if $m_S = \perp$: fail$_C$ else: success$_C$

server: $\boxed{\mathsf{M}_{\mathsf{put}}^{(S:1)}}$ $\mathsf{M}_{\mathsf{upd}}^{(S:1)}$ $(st_S, \varepsilon, in_S, m_C)$:
    pre: $st_S.\{sess, tree, files\}$
    $(sid, fid, c_{fkm}, c_{md}, c_f) \leftarrow m_C$
    $aid \leftarrow sess[sid]$
    $tree[aid] \xleftarrow{\cup} \{(fid, c_{fkm}, c_{md})\}$
    $files[fid] \leftarrow c_f$
    success$_S$
```

Figure 13: MEGA's file upload and update protocols $\mathsf{M}_{\mathsf{put}}$ and $\mathsf{M}_{\mathsf{upd}}$. The $\boxed{\text{boxed}}$ line is only included in $\mathsf{M}_{\mathsf{put}}$, the dashed line only in $\mathsf{M}_{\mathsf{upd}}$.

```
client: $\mathsf{M}_{\mathsf{get}}^{(C:1)}(st_C, \varepsilon, in_C, \varepsilon)$:
    pre: $in_C.\{fid\} st_C.\{sid\}$
    $st_C^{\text{tmp}}.fid \leftarrow fid$
    return $(st_C, st_C^{\text{tmp}}, \varepsilon, sid)$

$\mathsf{M}_{\mathsf{get}}^{(C:2)}(st_C, st_C^{\text{tmp}}, m_S)$:
    pre: $st_C.\{k_{mk}\},\ st_C^{\text{tmp}}.\{fid\}$
    for $(fid', c_{fkm}, c_{md}) \in m_S$:
        $d_{fkm} \leftarrow$ AES-ECB.DEC$(k_{mk}, c_{fkm})$
        $(k_f, n_f, \tau_f) \leftarrow d_{fkm}$
        $d_{md} \leftarrow$ AES-CBC.DEC$(k_f, c_{md})$
        $st_C.tree[fid'] \leftarrow (d_{md}, d_{fkm})$
    return $(st_C, st_C^{\text{tmp}}, out_C, fid)$

$\mathsf{M}_{\mathsf{get}}^{(C:3)}(st_C, st_C^{\text{tmp}}, m_S)$:
    pre: $st_C.\{tree\},\ st_C^{\text{tmp}}.\{fid\}$
    $(d_{md}, d_{fkm}) \leftarrow tree[fid]$
    $(f, out_C.dec) \leftarrow D^{\text{MFILE}}(d_{fkm}, m_S)$
    if $out_C.dec$: $out_C.f \leftarrow f$
    success$_C$

server: $\mathsf{M}_{\mathsf{get}}^{(S:1)}(st_S, \varepsilon, in_S, m_C)$:
    pre: $st_S.\{tree, sess\}$
    return $(st_S, \varepsilon, \varepsilon, tree[sess[m_C]])$

$\mathsf{M}_{\mathsf{get}}^{(S:2)}(st_S, st_S^{\text{tmp}}, m_C)$:
    pre: $st_S.\{files\}$
    success$_S(files[m_C])$
```

Figure 14: MEGA's file download protocol $\mathsf{M}_{\mathsf{get}}$.

server fails by returning $\perp$ for non-existent *sid*. The client decrypts the file key material with the master key stored in the client session state st_C. It then requests the file with identifier *fid*[14] from the server and decrypts it using the key material in the file tree and MEGA's custom decryption algorithm D^{MFILE}, returning the decrypted file f and a success boolean (stored directly in client protocol success decision $out_C.dec$).

[14]Clients find the file identifier *fid* corresponding to a file of a particular name in the decrypted file tree. However, our games require specifying *fid* in in_C in the first round, which can be interpreted as using the identifier from the cached file tree of a previous $\mathsf{M}_{\mathsf{get}}$ call (say, for a dummy *fid*).

```
client: $\mathsf{M}^{(C:1)}_{\mathsf{share}}(st_C, \varepsilon, in_C, \varepsilon)$:
 1  pre: $in_C.\{raid, fid\} in_C.\{rpk\}$
 2  $(d_{md}, d_{fkm}) \leftarrow st_C.tree[fid]$
 3  $c_{fkm} \leftarrow \text{RSA.Enc}(rpk, d_{fkm})$
 4  $m_C \leftarrow (st_C.sid, raid, fid, c_{fkm})$
 5  return $(st_C, \varepsilon, \varepsilon, m_C)$

$\mathsf{M}^{(C:2)}_{\mathsf{share}}(st_C, st_C^{\text{tmp}}, m_S)$:
 6  if $m_S = \perp$: fail$_C$ else: success$_C$

server: $\mathsf{M}^{(S:1)}_{\mathsf{share}}(st_S, \varepsilon, in_S, m_C)$:
 7  pre: $st_S.\{sess, acc, tree, shrtree\}$
 8  $(sid, raid, fid, c_{fkm}) \leftarrow m_C$
 9  if $sid \notin sess$ or $raid \notin acc$: fail$_S$
10  if $(fid, \cdot, \cdot) \notin tree[sess[sid]]$:
11      fail$_S$
12  $shrtree[raid][fid] \leftarrow c_{fkm}$
13  success$_S$
```

Figure 15: Simplified version of MEGA's legacy file sharing protocol $\mathsf{M}_{\mathsf{share}}$.

```
client: $\mathsf{M}^{(C:1)}_{\mathsf{accept}}(st_C, \varepsilon, in_C, \varepsilon)$:
 1  pre: $in_C.\{fid\}$, $st_C.\{sid\}$, $in_C.\{d_{md}\}$
 2  $st_C^{\text{tmp}}.d_{md} \leftarrow d_{md}$
 3  return $(st_C, \varepsilon, \varepsilon, (sid, fid))$

$\mathsf{M}^{(C:2)}_{\mathsf{accept}}(st_C, st_C^{\text{tmp}}, m_S)$:
 4  pre: $st_C.\{sk, k_{mk}\}$, $st_C^{\text{tmp}}.\{d_{md}\}$
 5  $d_{fkm} \leftarrow \text{RSA.Dec}(sk, m_S)$
 6  $c_{fkm} \leftarrow \text{AES-ECB.Enc}(k_m, d_{fkm})$
 7  $(k_f, n_f, \tau_f) \leftarrow d_{fkm}$
 8  $c_{md} \leftarrow \text{AES-CBC.Enc}(k_f, d_{md})$
 9  return $(st_C, st_C^{\text{tmp}}, out_C, (c_{fkm}, c_{md}))$

$\mathsf{M}^{(C:3)}_{\mathsf{accept}}(st_C, st_C^{\text{tmp}}, m_S)$:
10  if $m_S = \perp$: fail$_C$ else: success$_C$

server: $\mathsf{M}^{(S:1)}_{\mathsf{accept}}(st_S, \varepsilon, in_S, m_C)$:
11  pre: $st_S.\{sess, acc, tree, shrtree\}$
12  $(sid, st_S^{\text{tmp}}.fid) \leftarrow m_C$
13  $st_S^{\text{tmp}}.aid \leftarrow sess[sid]$
14  $m_S \leftarrow shrtree[st_S^{\text{tmp}}.aid][st_S^{\text{tmp}}.fid]$
15  return $(st_S, st_S^{\text{tmp}}, \varepsilon, m_S)$

$\mathsf{M}^{(S:2)}_{\mathsf{accept}}(st_S, st_S^{\text{tmp}}, m_C)$:
16  pre: $st_S.\{tree\}$, $st_S^{\text{tmp}}.\{aid, fid\}$
17  $c_{fkm}, c_{md} \leftarrow m_C$
18  $tree[aid] \leftarrow (fid, c_{fkm}, c_{md})$
19  success$_S$
```

Figure 16: Simplified version of MEGA's legacy protocol to accept shared files $\mathsf{M}_{\mathsf{accept}}$.

Sharing $\mathsf{M}_{\mathsf{share}}$ (Figure 15). Backendal et al. [6] exploited a deprecated code path that was still present in web clients before their disclosure. To show how their attack shows up in our security games, we describe a simplified version of this legacy file sharing protocol.

In MEGA's system, users establish contact relationships and sign the public key material of their contacts on first use (Trust On First Use). Instead of modeling this complex setup, we assume that the client obtained the public key rpk of receiver with account ID $raid$ over some out-of-band channel and provides it as input to $\mathsf{M}_{\mathsf{share}}$ in in_C. The client encrypts the file key material d_{fkm} with the RSA public key of the recipient and sends all encrypted file material to the server. If the session sid and receiver account $raid$ exist, and the caller owns the file to be shared (line 11 in Figure 15), the server temporarily stores the encrypted file keys c_{fkm} in the share tree $shrtree$.

Receiving $\mathsf{M}_{\mathsf{accept}}$ (Figure 16). The recipient of a file gets notified that a file with ID fid was shared with them by polling the server (we do not model this mechanism). The user then calls $\mathsf{M}_{\mathsf{accept}}$ to fetch the encrypted file key material and re-encrypt it using their RSA secret key sk (for decryption) and master key k_{mk} (for encryption), likely to avoid expensive public key operations in the future. Next, the new encrypted file key material c_{fkm} and metadata c_{md} are uploaded to the server and stored in the server's persistent file tree data structure $st_S.tree$.[15]

[15] In practice, the metadata contains user-specific values such as a pointer to the parent folder of the file in a user's storage. These values would need to be replaced when sharing a file, which we model as a user input for simplicity.

```
$\underline{\textbf{ExecHon}_{\textsc{Orc},\Pi}(in_S, st_{adv}, \textbf{arg})\text{:}}$
pre: $\textsc{Orc} \neq \textsc{Get1}$
$m' \leftarrow \textsc{Orc}(\textbf{arg})$
$(st_{adv}, st_{adv}^{\mathrm{tmp}}, out_{adv}, m) \leftarrow \Pi^{(S:1)}(st_{adv}, st_{adv}^{\mathrm{tmp}}, in_S, m')$
$st_{adv}^{\mathrm{tmp}} \leftarrow \varepsilon$ ; $i \leftarrow 2$
while $(out_{adv}.dec \neq \varepsilon$ or $out_C.dec \neq \varepsilon)$:
    $m' \leftarrow \textsc{Step}(\Pi, i_p, m)$
    $(st_{adv}, st_{adv}^{\mathrm{tmp}}, out_{adv}, m) \leftarrow \Pi^{(S:i)}(st_{adv}, st_{adv}^{\mathrm{tmp}}, m')$
    $i \leftarrow i + 1$
return $st_{adv}$, $out_{adv}$
```

Figure 17: Honest execution by using oracles and following the protocol steps.

A.2 Formal Attacks against MEGA

We proceed to show how attacks 1–3 from [6] (incorporating results from [3, 30]) against the previously-described MEGA protocol M lead to adversaries with non-negligible winning probabilities in games $\mathbf{G}^{\mathrm{C2CConfS}}_{\mathsf{M},n,\mathcal{PW}_n}$ and $\mathbf{G}^{\mathrm{C2CIntS}}_{\mathsf{M},n,\mathcal{PW}_n}$ for an arbitrary number of accounts $n \geq 1$. Moreover, we discuss a generic attack that more fundamentally prevents a security proof, even of the patched MEGA protocol.

The following attacks perform a number of operations honestly, for which we use the shared subroutine depicted in Figure 17.

<u>C2CConfS attack.</u> The adversary $\mathcal{A}_{conf}$, described in Figure 18, combines the RSA key and plaintext recovery attacks from Backendal et al. [6] to win $\mathbf{G}^{\mathrm{C2CConfS}}_{\mathsf{M},n,\mathcal{PW}_n}$ with non-negligible probability.

The adversary starts by honestly registering a user with email address *aid*. It picks two distinct files (f_0 and f_1) of the same length and uploads them as challenge. Next, $\mathcal{A}_{conf}$ runs the RSA key recovery attack from [6, Section III] to recover the prime factors p and q from the RSA modulus N. In short, the attack exploits the lack of integrity protection for the RSA secret key ciphertext c_{rsa} and properties of MEGA's RSA decryption RSA.Dec on the client to perform a binary search for the smaller factor of the modulus N. The adversary tampers with the AES-ECB ciphertext c_{rsa} on line 18 of Figure 18 to partially garble the secret key. Due to MEGA's implementation of RSA.Dec, clients leak whether the encrypted challenge value p' is less than the smaller RSA prime factor $\min(p, q)$ (by responding with $m_C = 0$) or greater (resulting in $m_C > 0$ with high probability).

With knowledge of the RSA secret key, adversary $\mathcal{A}_{conf}$ runs the plaintext recovery attack [6, Section IV] to decrypt the challenge file. On a high level, the adversary injects the encrypted file key material c_{fkm} for the challenge file in the RSA secret key ciphertext c_{rsa} that is exchanged during the authentication protocol. This is possible because both the RSA key and file keys are encrypted with the master key and AES-ECB, enabling the replacement of individual ciphertext blocks in c_{rsa} and c_{fk}. Together with a specially crafted session ID (the value $u' \cdot q'$, using the recovered RSA secret key), the client inadvertently decrypts the injected file key material (k_f, n_f, τ_f) in the session ID that it sends back to the server in $\mathsf{M}^{(C:3)}_{\mathsf{auth}}$. The attack is probabilistic and fails with probability at most 2^{-30} (as explained by [6]). As a successful file decryption always results in the adversary winning the game $\mathbf{G}^{\mathrm{C2CConfS}}_{\mathsf{M},n,\mathcal{PW}_n}$, it has advantage $\mathbf{Adv}^{\mathrm{C2CConfS}}_{\mathsf{M},n,\mathcal{PW}_n}(\mathcal{A}_{conf}) = 1 - 2^{-30}$ with 1023 calls to $\mathsf{M}_{\mathsf{auth}}$. Implementing the optimization of [6], the adversary $\mathcal{A}_{conf}$ can stop the binary search after 512 queries to $\mathsf{M}_{\mathsf{auth}}$ and recover the missing part of the secret key with lattice-based techniques. Moreover, Heninger and Ryan [30] proposed a tailored lattice-based RSA key recovery attack that further reduces the cost to 6 calls to $\mathsf{M}_{\mathsf{auth}}$.

```
adversary $\mathcal{A}_{conf}^{\textsc{AReg1},\textsc{Auth1},\textsc{Put1},\textsc{Upd1},\textsc{Get1},\textsc{Share1},\textsc{Acpt1},\textsc{Chall},\textsc{Comp}}()$:
    // Attack setup: honest account creation and challenge file upload
$aid \leftarrow$ "evil@email.com" ; $in_C.aid \leftarrow aid$ ; $i_u \leftarrow 0$ ; $i_c \leftarrow 0$ ; $fid \leftarrow 0$
$in_C.tk \leftarrow\$ \{0,1\}^{128}$ ; $st_{adv}.tokens \leftarrow \{(aid, in_C.tk)\}$
$(st_{adv}, out_{adv}) \leftarrow \mathbf{ExecHon}_{\textsc{AReg1},\mathsf{M_{areg}}}(\varepsilon, st_{adv}, (i_u, in_C))$
$sid \leftarrow\$ \mathrm{S}$ ; $in_S.sid \leftarrow\$ sid$
$(st_{adv}, out_{adv}) \leftarrow \mathbf{ExecHon}_{\textsc{Auth1},\mathsf{M_{auth}}}(in_S, st_{adv}, (i_u, in_C))$
$in_C.fid \leftarrow fid$
$f_0 \leftarrow 0$ ; $f_1 \leftarrow 1$
$(st_{adv}, out_{adv}) \leftarrow \mathbf{ExecHon}_{\textsc{Chall},\mathsf{M_{put}}}(\varepsilon, st_{adv}, (i_c, in_C, f_0, f_1, \textsc{Put1}))$
    // Read target user ciphertexts from malicious server state
$(n_C, c_m, h_a, (pk, c_{rsa})) \leftarrow st_{adv}.acc[sid]$
for $(fid', c'_{fkm}, c'_{md}) \in st_{adv}.tree[aid]$:  // Find key material ciphertext $c_{fkm}$ for the newly uploaded file
    if $fid = fid'$: $c_{fkm} \leftarrow c'_{fkm}$
$c_f \leftarrow st_{adv}.files[fid]$
    // Recover RSA private key (see [6, Section III])
$l \leftarrow 0$ ; $r \leftarrow \lfloor \sqrt{N} \rfloor$ ; $i_p \leftarrow 4$
do:
    $p' \leftarrow \lfloor \frac{r+l}{2} \rfloor$ ; $in_C.aid \leftarrow aid$
    $m_C \leftarrow \textsc{Auth1}(i_u, in_C)$
    $m_C \leftarrow \textsc{Step}(\mathsf{M_{auth}}, i_p, \mathrm{H}(const \,\|\, n_C))$
    $m_{adv} \leftarrow (c_m, (pk, c_{rsa} \oplus 1), \mathrm{RSA}.\textsc{Enc}(pk, p'))$
    $m_C \leftarrow \textsc{Step}(\mathsf{M_{auth}}, i_p, m_{adv})$
    if $m_C = 0$: $r \leftarrow p'$ else: $l \leftarrow p' + 1$
    $i_p \leftarrow i_p + 1$
until $p' \mid N$
    // Decrypt the challenge file (see [6, Section IV])
$q' \leftarrow N/p'$ ; $u' \leftarrow (p')^{-1} \bmod q'$
$in_C.aid \leftarrow aid$
$m_C \leftarrow \textsc{Auth1}(i_u, in_C)$
$m_C \leftarrow \textsc{Step}(\mathsf{M_{auth}}, i_p, \mathrm{H}(const \,\|\, n_C))$
$c'_{rsa} \leftarrow \lfloor c_{rsa} \cdot 2^{-8 \cdot 128} \rfloor \,\|\, c_{fkm} \,\|\, (c_{rsa} \bmod 2^{6 \cdot 128})$
$m_{adv} \leftarrow (c_m, (pk, c'_{rsa}), \mathrm{RSA}.\textsc{Enc}(pk, u' \cdot q'))$
$m_C \leftarrow \textsc{Step}(\mathsf{M_{auth}}, i_p, m_{adv})$
for $y = 0, 1, \ldots, 2^{16} - 1$:
    $(k_f, n_f, \tau_f) \leftarrow \lfloor \frac{y \,\|\, m_C}{q} \cdot 256^{123} \rfloor \bmod 256^{32}$
    $(f, succ) \leftarrow D^{\mathrm{MFILE}}(k_f, n_f, \tau_f, c_f)$
    if $succ$: return $\mathbb{I}_{f=f_1}$
return 0  // (attack failed; reached with probability less than $2^{30}$)
```

Figure 18: Adversary running RSA key attack (recovering prime factors p, q of the modulus $N = p \cdot q$) and the plaintext recovery attack from [6].

C2CIntS Attack. Figure 19 describes the adversary $\mathcal{A}_{int}$, running the integrity attack from Backendal et al. [6] to win $\mathbf{G}^{\mathrm{C2CIntS}}_{\mathsf{M},n,\mathcal{PW}_n}$ with non-negligible probability. The integrity attack from [6] requires knowing a plaintext-ciphertext pair. Instead of building on $\mathcal{A}_{conf}$ and decrypting a randomly picked AES-ECB ciphertext (to chain this attack with the previous two as in [6]), adversary $\mathcal{A}_{int}$ uses the AES-ECB encryption oracle from [3, Section 2.2] to obtain the ciphertext for a chosen plaintext from the share accept protocol, as follows. The adversary first honestly registers and authenticates a user, and then makes them accept a "shared" file chosen by the adversary. The client decrypts the shared file key material d_{fkm} from c_{fkm} (using RSA) and re-encrypts it under the user's master key (using AES-ECB), sending the resulting c'_{fkm} to the server.

Since the adversary picked the file key material d_{fkm} (which consists of two AES blocks, the first containing the file key k_f, and the second a nonce n_f and tag τ_f), it now knows both the plaintext blocks and the corresponding ciphertext blocks c'_{fkm}. It will use the second one of these known blocks to mount the integrity attack of [6].

Note that local file key ciphertexts are encrypted with AES-ECB, and therefore not integrity-protected. Hence, as [6] describe, the adversary can trivially create a forged file key ciphertext. For this attack, it will use the second of the ciphertext blocks from above for which it knows the corresponding plaintext (namely, the nonce n_f and tag τ_f), and create a ciphertext which consists of that block duplicated twice. That is, the forged ciphertext consists of two identical AES-ECB blocks, for which the adversary knows the plaintext value $n_f \,\|\, \tau_f$.

Due to the way d_{fkm} is parsed in MEGA's custom decryption algorithm D^{MFILE}, the duplicate ciphertext blocks cause the client to derive the all-zeros file encryption key $k''_f = 0^{128}$ when it decrypts the malicious file key ciphertext. Hence, the adversary knows which key the user derives (despite that this key was never used before), and can use it to fabricate a file and metadata ciphertext. The nonce n_f and tag τ_f, which are also derived using D^{MFILE} on the file key ciphertext, are unaffected by this vulnerability. Hence, in addition to knowing the file key, that adversary also knows the nonce n_f and tag τ_f that will be used during file decryption. This is needed for the attack, since encrypting the fabricated file must produce the tag τ_f when using nonce n_f in order to pass verification during decryption.

Figure 6 of [6] explains how it is possible to create such a file by changing only 128 bits of an arbitrary file; we call this algorithm $E^{\text{MFILE}'}$. As long as the file encrypted by $E^{\text{MFILE}'}$ is not the file that was shared in the first step of the attack, this leads to a win in $\mathbf{G}^{\text{C2CIntS}}_{\mathsf{M},n,\mathcal{PW}_n}$. Therefore, adversary $\mathcal{A}_{int}$ has advantage $\mathbf{Adv}^{\text{C2CConf}}_{\mathsf{M},n,\mathcal{PW}_n}(\mathcal{A}_{int}) = 1$ with only a single call to each of $\mathsf{M}_{\mathsf{areg}}$, $\mathsf{M}_{\mathsf{auth}}$, $\mathsf{M}_{\mathsf{accept}}$, and $\mathsf{M}_{\mathsf{get}}$.

Remaining obstacles in the patched MEGA protocol. Despite having mitigated the attacks from [6, 3, 30], MEGA still cannot be proven secure in our model. Even with the patches, the encryption scheme remains vulnerable to theoretic attacks which prevent the protocol from satisfying our security notions. Unfortunately, mitigating these attacks is infeasible for MEGA, as it would require all of their users to re-encrypt all of their files.

One example of a remaining issues shows up in Figure 19. In their patches, MEGA decided to retain the file sharing feature that exposes an AES-ECB encryption oracle unchanged, instead of switching to more expensive asymmetric encryption. Hence, an adversary could still run the the attack until line 12 of Figure 19 and obtain file key material d_{fkm} that is encrypted with the target user's master key. Instead of running the integrity attack from [6] to achieve a full break in practice, the adversary could also win $\mathbf{G}^{\text{C2CIntS}}_{\mathsf{M},n,\mathcal{PW}_n}$ with a more theoretical attack by reusing the file key material d_{fkm} to encrypt a new file. This is possible because file keys are not bound to the file or identifier *fid*, hence allowing an adversary that knows any file key material to reuse it to encrypt other files. The new file is not tracked as shared by the adversary, and therefore constitutes a valid forgery.

```
adversary $\mathcal{A}_{int}^{\text{AReg1},\text{Auth1},\text{Put1},\text{Upd1},\text{Get1},\text{Share1},\text{Acpt1},\text{Comp}}()$:
    // Attack setup: honest account creation
    $aid \leftarrow$ "evil@email.com" ; $in_C.aid \leftarrow aid$ ; $i_u \leftarrow 0$ ; $i_c \leftarrow 0$
    $in_C.tk \leftarrow\!\$\ \{0,1\}^{128}$ ; $st_{adv}.tokens \leftarrow \{(aid, in_C.tk)\}$
    $(st_{adv}, out_{adv}) \leftarrow \mathbf{ExecHon}_{\text{AReg1},\mathsf{M_{areg}}}(\varepsilon, st_{adv}, (i_u, in_C))$
    $sid \leftarrow\!\$\ \mathrm{S}$ ; $in_S.sid \leftarrow\!\$\ sid$
    $(st_{adv}, out_{adv}) \leftarrow \mathbf{ExecHon}_{\text{Auth1},\mathsf{M_{auth}}}(in_S, st_{adv}, (i_u, in_C))$
    // Prepare adversary chosen "shared file"
    $(n_C, c_m, h_a, (pk, c_{rsa})) \leftarrow st_{adv}.acc[sid]$
    $(k_f, n_f) \leftarrow\!\$\ (\{0,1\}^{128} \times \{0,1\}^{64})$
    $(c_f, \tau_f) \leftarrow E^{\text{MFILE}}(k_f, n_f, 0)$  // Encrypt any file, for instance "0"
    $d_{fkm} \leftarrow (k_f, n_f, \tau_f)$ ; $c_{fkm} \leftarrow \text{RSA.Enc}(pk, d_{fkm})$
    // Obtain valid plaintext-ciphertext pair (over file sharing, see [3, Section 2.2])
    $i_p \leftarrow 3$ ; $in_C.fid \leftarrow 0$ ; $in_C.d_{md} \leftarrow \varepsilon$
    $(st_{adv}, out_{adv}) \leftarrow \text{Acpt1}(i_c, in_C)$
    $(c'_{fkm}, c'_{md}) \leftarrow \text{Step}(\mathsf{M_{accept}}, i_p, c_{fkm})$  // Now, $c'_{fkm} = \text{AES-ECB.Enc}(k_m, d_{fkm})$
    // Fabricate file ciphertext
    $c_0 \| c_1 \leftarrow c'_{fkm}$ ; $c''_{fkm} \leftarrow c_0 \| c_0$  // for $|c_0| = |c_1| = 128$ bits
    $k''_f \leftarrow 0^{128}$  // Result of decrypting $c''_{fkm}$, an artifact of MEGA's encryption $E^{\text{MFILE}}$
    $c''_f \leftarrow E^{\text{MFILE}'}_{k''_f}(n_f, \tau_f)$  // Creating some file encryption that produces tag $\tau_f$, see [6, Figure 6].
    $c''_{md} \leftarrow \text{AES-CBC.Enc}(k''_f, \varepsilon)$
    // Serve fabricated file
    $i_p \leftarrow i_p + 1$ ; $in_C.fid \leftarrow 0$
    $(st_{adv}, out_{adv}) \leftarrow \text{Get1}(i_c, in_C)$
    $m_C \leftarrow \text{Step}(\mathsf{M_{get}}, i_p, \{(in_C.fid, c''_{fkm}, c''_{md})\})$
    $m_C \leftarrow \text{Step}(\mathsf{M_{get}}, i_p, c''_f)$  // client succeeds because $c''_f$ was encrypted with $k''_f$
    return
```

Figure 19: Adversary running the integrity attack from [6, Section V] with a valid plaintext-ciphertext pair from the sharing feature exploited in follow-up work by Albrecht et al. [3, Section 2.2].

B Building Block Details

This section specifies the building blocks used in Section 6 to prove the security of our cloud storage scheme in more detail.

B.1 Standard Building Blocks

We use the following standard building blocks:

<u>PRF.</u> Let $\mathsf{F}\colon \{0,1\}^{kl} \times \{0,1\}^* \to \{0,1\}^{kl}$ be a function family. We define $\mathbf{G}_{\mathsf{F}}^{\text{PRF-1}}$ to be the multi-user [8] PRF game in which an adversary $\mathcal{A}$ can generate new PRF keys through oracle New and through oracle Fn$(\cdot,\cdot)$, on input i, x, gets back $\mathsf{F}(k_i, x)$, where k_i is the i^{th} key generated by New. Game $\mathbf{G}_{\mathsf{F}}^{\text{PRF-0}}$ is identical, except that Fn(i,x) returns a consistently sampled random string in $\{0,1\}^{kl}$. We define the advantage of adversary $\mathcal{A}$ against the PRF security of F to be $\mathbf{Adv}_{\mathsf{F}}^{\text{PRF}}(\mathcal{A}) := \Pr\left[\mathbf{G}_{\mathsf{F}}^{\text{PRF-1}}\right] - \Pr\left[\mathbf{G}_{\mathsf{F}}^{\text{PRF-0}}\right]$.

<u>AEAD.</u> We define a nonce-based authenticated encryption scheme with associated data [51] as $\mathsf{AEAD} = (\mathsf{Enc}, \mathsf{Dec})$, where we write $c \leftarrow \mathsf{Enc}(k, n, m, ad)$ and $m \leftarrow \mathsf{Dec}(k, n, c, ad)$ for encryption resp. decryption of a message $m \in \{0,1\}^*$ resp. ciphertext $c \in \{0,1\}^*$ using key $k \in \{0,1\}^{kl}$,

```
Game $\mathbf{G}^{\text{IND\$-}b}_{\mathsf{AEAD}}(\mathcal{A})$:
global $u$, $\mathrm{S}_n$, $\mathrm{T}_{\mathsf{AEAD}}$
$b' \leftarrow\!\!\$\ \mathcal{A}^{\textsc{New},\textsc{Enc},\textsc{Dec}}()$
return $b'$

$\textsc{New}()$:
$u \leftarrow u + 1$
$k_u \leftarrow\!\!\$\ \{0,1\}^{kl}$

$\textsc{Enc}(i, n, m, ad)$:
if $(i, n) \in \mathrm{S}_n$:
    return $\bot$
$\mathrm{S}_n \overset{\cup}{\leftarrow} \{(i, n)\}$
$c_1 \leftarrow \mathsf{Enc}(k_i, n, m, ad)$
$c_0 \leftarrow\!\!\$\ \{0,1\}^{|c_1|}$
$\mathrm{T}_{\mathsf{AEAD}}[i, n, c_b, ad] \leftarrow m$
return $c_b$

$\textsc{Dec}(i, n, c, ad)$:
$m_1 \leftarrow \mathsf{Dec}(k_i, n, c, ad)$
$m_0 \leftarrow \mathrm{T}_{\mathsf{AEAD}}[i, n, c, ad]$
return $m_b$
```

```
Game $\mathbf{G}^{\text{IND-CCA-}b}_{\mathsf{AEAD}}(\mathcal{A})$:
global $u$, $\mathrm{S}_n$, $\mathrm{T}_{\mathsf{AEAD}}$
$b' \leftarrow\!\!\$\ \mathcal{A}^{\textsc{New},\textsc{Enc},\textsc{Dec}}()$
return $b'$

$\textsc{New}()$:
$u \leftarrow u + 1$
$k_u \leftarrow\!\!\$\ \{0,1\}^{kl}$

$\textsc{Enc}(i, n, m_0, m_1, ad)$:
if $(i, n) \in \mathrm{S}_n \vee |m_0| \neq |m_1|$:
    return $\bot$
$\mathrm{S}_n \overset{\cup}{\leftarrow} \{(i, n)\}$
$c_0 \leftarrow \mathsf{Enc}(k_i, n, m_0, ad)$
$c_1 \leftarrow \mathsf{Enc}(k_i, n, m_1, ad)$
$\mathrm{S}_c \overset{\cup}{\leftarrow} \{(i, n, c_b, ad)\}$
return $c_b$

$\textsc{Dec}(i, n, c, ad)$:
if $(i, n, c, ad) \in \mathrm{S}_c$:
    return $\bot$
$m \leftarrow \mathsf{Dec}(k_i, n, c, ad)$
return $m$
```

```
Game $\mathbf{G}^{\text{INT-CTXT}}_{\mathsf{AEAD}}(\mathcal{A})$:
global $\mathsf{win}$, $u$, $\mathrm{S}_n$, $\mathrm{S}_c$
$\mathcal{A}^{\textsc{New},\textsc{Enc},\textsc{Dec}}()$
return $\mathsf{win}$

$\textsc{New}()$:
$u \leftarrow u + 1$
$k_u \leftarrow\!\!\$\ \{0,1\}^{kl}$

$\textsc{Enc}(i, n, m, ad)$:
if $(i, n) \in \mathrm{S}_n$:
    return $\bot$
$\mathrm{S}_n \overset{\cup}{\leftarrow} \{(i, n)\}$
$c \leftarrow \mathsf{Enc}(k_i, n, m, ad)$
$\mathrm{S}_c \overset{\cup}{\leftarrow} \{(i, n, c, ad)\}$
return $c$

$\textsc{Dec}(i, n, c, ad)$:
$m \leftarrow \mathsf{Dec}(k_i, n, c, ad)$
if $(i, n, c, ad) \notin \mathrm{S}_c \ \wedge\ m \neq \bot$:
    $\mathsf{win} \leftarrow \mathsf{true}$
return $m$
```

Figure 20: Games formalizing authenticated encryption (IND\$), confidentiality (IND-CCA) and integrity of ciphertexts (INT-CTXT) of an authenticated encryption scheme with associated data AEAD.

---

nonce $n \in \{0,1\}^{nl}$, and associated data $ad \in \{0,1\}^*$.

Below, we formally define the advantage of an adversary $\mathcal{A}$ against the IND\$, IND-CCA and INT-CTXT security of AEAD in the multi-user setting denoted as $\mathbf{Adv}^{\text{IND\$}}_{\mathsf{AEAD}}(\mathcal{A})$, $\mathbf{Adv}^{\text{IND-CCA}}_{\mathsf{AEAD}}(\mathcal{A})$, and $\mathbf{Adv}^{\text{INT-CTXT}}_{\mathsf{AEAD}}(\mathcal{A})$, respectively. The games formalizing IND\$, IND-CCA and INT-CTXT of an AEAD scheme are given in Figure 20. We define

$$\mathbf{Adv}^{\text{IND\$}}_{\mathsf{AEAD}}(\mathcal{A}) = \Pr\left[\mathbf{G}^{\text{IND\$-1}}_{\mathsf{AEAD}}\right] - \Pr\left[\mathbf{G}^{\text{IND\$-0}}_{\mathsf{AEAD}}\right],$$
$$\mathbf{Adv}^{\text{IND-CCA}}_{\mathsf{AEAD}}(\mathcal{A}) = \Pr\left[\mathbf{G}^{\text{IND-CCA-1}}_{\mathsf{AEAD}}\right] - \Pr\left[\mathbf{G}^{\text{IND-CCA-0}}_{\mathsf{AEAD}}\right], \text{ and}$$
$$\mathbf{Adv}^{\text{INT-CTXT}}_{\mathsf{AEAD}}(\mathcal{A}) = \Pr\left[\mathbf{G}^{\text{INT-CTXT}}_{\mathsf{AEAD}}\right].$$

MAC. A message authentication code $\mathsf{MAC} = (\mathsf{Tag}, \mathsf{Vrfy})$ consists of a tag generation algorithm $\mathsf{Tag}\colon \{0,1\}^{kl} \times \{0,1\}^* \to \{0,1\}^{tl}$ and a verification algorithm $\mathsf{Vrfy}\colon \{0,1\}^{kl} \times \{0,1\}^* \times \{0,1\}^{tl} \to \{\bot, \top\}$. Both take a key k of length kl and an arbitrary length message m as input, Tag produces a tag of length tl. Correctness is defined as $\mathsf{Vrfy}(k, m, \mathsf{Tag}(k, m)) = \top$.

B.2 OPRF Building Block

We describe the 2HashDH OPRF in Figure 21, which we will use inlined in the protocols of CSS. It is based on a group G of prime order q and two hash functions $\mathrm{H}_1\colon \{0,1\}^* \to G$ and $\mathrm{H}_2\colon \{0,1\}^* \times G \to$

```
$\mathsf{2HashDH.Req}(x)$:
$r \leftarrow\!\!\$\ \mathbb{Z}_q$ ; $\alpha \leftarrow \mathrm{H}_1(x)^r$
return $((r, x), \alpha)$

$\mathsf{2HashDH.Finalize}((r, x), \beta)$:
if $\beta \notin G$: return $\bot$
$\gamma \leftarrow \beta^{1/r}$
$y \leftarrow \mathrm{H}_2(x, \gamma)$
return $y$

$\mathsf{2HashDH.KeyGen}()$:
$k \leftarrow\!\!\$\ \mathbb{Z}_q$
return $k$

$\mathsf{2HashDH.BlindEv}(k, \alpha)$:
if $\alpha \notin G$: return $\bot$
$\beta \leftarrow \alpha^k$
return $\beta$
```

Figure 21: The $\mathsf{2HashDH}$ OPRF operations (client on the left, server on the right) for a group G and hash functions $\mathrm{H}_1\colon \{0,1\}^* \to G$ and $\mathrm{H}_2\colon \{0,1\}^* \to \{0,1\}^{kl}$ which compute $\mathsf{2HashDH}_k(x) := \mathrm{H}_2(x, \mathrm{H}_1(x)^k))$ for a client-chosen value x and server key k. The colored values α and β are the messages exchanged between client and server.

$\{0,1\}^{kl}$. The server-side key $k \in \mathbb{Z}_q$ is generated using $\mathsf{KeyGen}()$. The client computes a request via $\mathsf{Req}(x)$, which samples a random blinding value $r \in \mathbb{Z}_q$, computes the request to be sent to the server as $\alpha \leftarrow \mathrm{H}_1(x)^r$ and outputs it together with local state (r, x). The server computes the blind evaluation $\mathsf{BlindEv}(k, \alpha)$ as $\beta \leftarrow \alpha^k$. The client finally in $\mathsf{Finalize}$ unblinds β using r and outputs $y \leftarrow \mathrm{H}_2(x, \gamma))$.

C Proofs of Client-To-Client Security for CSS

C.1 Proof of Theorem 6.1

This section provides the full details of the proof of the selective client-to-client confidentiality of CSS, as stated in Theorem 6.1. Recall that $\mathcal{Q}_{\text{ORC}}(\mathcal{A})$ and $q_{\text{ORC}}(\mathcal{A})$ denote the maximum number of queries to oracle ORC by adversary $\mathcal{A}$ in total and per user, respectively. We use the convention that $\mathcal{Q}_{\text{PUT1}}(\mathcal{A})$ and $\mathcal{Q}_{\text{UPD1}}(\mathcal{A})$ include indirect queries made via oracle CHALL. To count only queries via oracle CHALL, we let CPUT1 and CUPD1 denote oracle CHALL with input ORC set to PUT1 and UPD1, respectively.

The proof proceeds through a sequence of games $\mathrm{G}_0 - \mathrm{G}_7$, which all provide the oracles from game $\mathbf{G}^{\text{C2CConfS}}_{\mathsf{CSS},n,\mathcal{PW}_n}$. Each game will begin by sampling a challenge bit b, which determines the file that is encrypted in oracle CHALL. The finalize procedure in each game is equivalent to that of $\mathbf{G}^{\text{C2CConfS}}_{\mathsf{CSS},n,\mathcal{PW}_n}$. Namely, the game runs $\mathcal{A}$ to obtain b'. It returns $\neg\mathsf{trivial} \wedge (b' = b)$, where $\mathsf{trivial}$ is the predicate (defined on line 5 of Figure 5) indicating if $\mathcal{A}$ performed a trivial attack. Throughout, we let $\Pr[\,\mathrm{G}_i\,]$ denote the probability that $\mathrm{G}_i(\mathcal{A})$ returns true.

We model the hash functions H_1, H_2 used in account registration and authentication as (programmable) random oracles, implemented through lazy sampling with consistency ensured by tables $\mathrm{T}_{\mathrm{H}1}[\cdot,\cdot]$ and $\mathrm{T}_{\mathrm{H}2}[\cdot,\cdot,\cdot]$. The adversary can interact with the tables through oracles RO_1 and RO_2.

Games G_0–G_3: Replacing *rw* by random. We begin with game G_0, which is equivalent to $\mathbf{G}^{\text{C2CConfS}}_{\mathsf{CSS},n,\mathcal{PW}_n}(\mathcal{A})$. In particular, the client-side output $rw \in \{0,1\}^{kl}$ from the OPRF flow in protocols $\mathsf{CSS}_{\mathsf{areg}}$ and $\mathsf{CSS}_{\mathsf{auth}}$ is computed using the random oracles RO_1 and RO_2 in place of H_1 and H_2. For example, line 3 of Figure 8 is replaced by

```
$\mathrm{G}_0$: $\mathsf{CSS}^{(C:1)}_{\mathsf{areg}}$:
3  $rw \leftarrow \mathrm{RO}_2(aid, pw, (\mathrm{RO}_1(aid, pw))^k)$
```

where we only display the modified line.

The effect of this change is only to model the hash functions as random oracles. Hence

$$\mathbf{Adv}^{\mathrm{C2CConfS}}_{\mathsf{CSS},n,\mathcal{PW}_n}(\mathcal{A}) := 2 \cdot \Pr[\mathrm{G}_0] - 1. \tag{8}$$

In the first sequence of games, the goal is to move to game G_3, where rw is replaced by an independent random string in $\{0,1\}^{kl}$ for all honest (uncompromised) users.

Specifically, G_3 differs from G_0 in the following ways. In protocol $\mathsf{CSS}_{\mathsf{areg}}$, when the adversary registers a user $i_u \notin \mathrm{S}_{\mathsf{comp}}$ (by calling oracle AREG1 for i_u) the calls to RO_1 and RO_2 computing γ and rw are replaced with consistent random sampling from G and $\{0,1\}^{kl}$, respectively. That is, line 3 in G_0 is replaced by the following lines

```
$\mathrm{G}_3$: $\mathsf{CSS}^{(C:1)}_{\mathsf{areg}}$:
3  if $i_u \in \mathrm{S}_{\mathsf{comp}}$:
       $\gamma \leftarrow (\mathrm{RO}_1(aid, pw))^k$; $rw \leftarrow \mathrm{RO}_2(aid, pw, \gamma)$
4  else:
5      $\mathrm{T_I}[aid, pw] \leftarrow\!\!\$\ G$; $\gamma \leftarrow (\mathrm{T_I}[aid, pw])^k$
6      $\mathrm{T_{rw}}[aid, pw, \gamma] \leftarrow\!\!\$\ \{0,1\}^{kl}$; $rw \leftarrow \mathrm{T_{rw}}[aid, pw, \gamma]$
```

The same change is also applied in $\mathsf{CSS}_{\mathsf{auth}}$ for all $i_u \notin \mathrm{S}_{\mathsf{comp}}$. That is, $\alpha \leftarrow (\mathrm{RO}_1(aid, pw))^r$ on line 15 of Figure 8 is replaced by

```
$\mathrm{G}_3$: $\mathsf{CSS}^{(C:1)}_{\mathsf{auth}}$:
15 if $i_u \in \mathrm{S}_{\mathsf{comp}}$: $\alpha \leftarrow (\mathrm{RO}_1(aid, pw))^r$ else: $\alpha \leftarrow (\mathrm{T_I}[aid, pw])^r$
```

and on line 22, computing $rw \leftarrow \mathrm{RO}_2(aid, pw, \gamma)$ is replaced by

```
$\mathrm{G}_3$: $\mathsf{CSS}^{(C:2)}_{\mathsf{auth}}$:
22 if $i_u \in \mathrm{S}_{\mathsf{comp}}$: $rw \leftarrow \mathrm{RO}_2(aid, pw, \gamma)$
23 else:
24     if $\mathrm{T_{rw}}[aid, pw, \gamma] = \perp$: $\mathrm{T_{rw}}[aid, pw, \gamma] \leftarrow\!\!\$\ \{0,1\}^{kl}$
25     $rw \leftarrow \mathrm{T_{rw}}[aid, pw, \gamma]$
```

We claim that the change from G_0 to G_3 is indistinguishable to an adversary, except with some small probability. Formally, we want to bound $\Pr[\mathrm{G}_0] - \Pr[\mathrm{G}_3]$. To this end, we introduce two additional games, G_1 and G_2. In the following, we focus on the places where these games differ from $\mathbf{G}^{\mathrm{C2CConfS}}_{\mathsf{CSS}}$. To this end, we let $\mathrm{AUTH2}(i_p, m) := \mathrm{STEP}(\mathsf{CSS}_{\mathsf{auth}}, i_p, m)$ be an oracle which takes care of the query processing when $\mathcal{A}$ runs the *second* client-side step of protocol $\mathsf{CSS}_{\mathsf{auth}}$ through oracle STEP.

Games G_1 and G_2 function exactly like G_3, except in oracles RO_1 and RO_2, and where rw is computed in oracles AREG1 and AUTH2. There, as shown in Figure 22, a flag bad is set to true if oracle RO_1 resp. RO_2 is queried by the adversary on inputs (a, b) resp. (a, b, c) such that $\mathrm{T_I}[a, b] \neq \perp$, resp. $\mathrm{T_{rw}}[a, b, c] \neq \perp$. (That is, if the adversary queries the random oracles on inputs corresponding to the aid and pw of a non-compromised user that has been previously registered and/or authenticated.) Game G_1, which contains the boxed code, then overwrites $\mathrm{T_{H1}}[a, b]$ resp. $\mathrm{T_{H2}}[a, b, c]$ with $\mathrm{T_I}[a, b]$, resp. $\mathrm{T_{rw}}[a, b, c]$, whereas G_2 does nothing. Additionally, bad is also set to true in AREG1 and AUTH2 if the adversary has already queried RO_1 resp. RO_2 on input (aid, pw) resp. (aid, pw, γ), such that $\mathrm{T_{H1}}[aid, pw]$ resp. $\mathrm{T_{H2}}[aid, pw, \gamma]$ are initiated. Game G_1 then overwrites the affected entry in tables $\mathrm{T_I}$ and/or $\mathrm{T_{rw}}$ with these values, whereas G_2 again does nothing.

It is clear that G_2 is equivalent to G_3 (the only difference being that the bad flag is set, which does not affect the adversary's chances of winning), and hence that $\Pr[G_2] = \Pr[G_3]$. Let Bad denote the event that G_2 sets the bad flag. G_1 and G_2 are identical-until-Bad, so $\Pr[G_1] - \Pr[G_2] \leq \Pr[\mathsf{Bad}]$ by the fundamental lemma of game playing [11]. Finally, we claim that G_1 is equivalent to G_0. To see this, note that in G_1, all rw values are computed in a way that is consistent with the random oracles – any potential inconsistency is fixed through the programming after the bad event. Hence, $\Pr[G_0] = \Pr[G_1]$.

Through standard equation rewriting, this gives

$$\Pr[G_0] = \Pr[\mathsf{Bad}] + \Pr[G_3]. \tag{9}$$

Games G_0–G_3: Bounding Bad. Next, we proceed to bound the probability that G_2 sets bad. Let G_2' be identical to G_2, except that instead of returning $\neg\mathsf{trivial} \wedge (b = b')$ in the finalize procedure, it returns bad, such that $\Pr[\mathsf{Bad}] = \Pr[G_2']$. Next, we construct a game G_2'', which is identical to G_2', except that it additionally keeps track of the queries made by adversary $\mathcal{A}$, as follows. All queries $\mathrm{RO}_1(a, b)$ resp. $\mathrm{RO}_2(a, b, c)$ by adversary $\mathcal{A}$ (notably, not by the game itself!) are collected in sets S_{H1} and S_{H2}, respectively, through $S_{H1} \overset{\cup}{\leftarrow} \{(a, b)\}$, $S_{H2} \overset{\cup}{\leftarrow} \{(a, b, c)\}$. Queries to oracle AREG1 and AUTH2 which initialize table entries $T_I[\mathit{aid}, \mathit{pw}]$ resp. $T_{rw}[\mathit{aid}, \mathit{pw}, \gamma]$ are collected in $S_I \overset{\cup}{\leftarrow} \{(\mathit{aid}, \mathit{pw})\}$ and $S_{rw} \overset{\cup}{\leftarrow} \{(\mathit{aid}, \mathit{pw}, \gamma)\}$. Furthermore, instead of returning bad, G_2'' returns $(S_I \cap S_{H1}) \cup (S_{rw} \cap S_{H2}) \neq \emptyset$. Games G_2' and G_2'' are given in Figure 23.

We claim that the two return statements are equivalent. The flag bad is set to true for one of two reasons: (1) when the adversary[16] makes a query $\mathrm{RO}_1(a, b)$ resp. $\mathrm{RO}_2(a, b, c)$ such that $T_I[a, b]$ resp. $T_{rw}[a, b, c]$ is already initialized. This implies that $(a, b) \in S_{H1} \cap S_I$ or $(a, b, c) \in S_{H2} \cap S_{rw}$. Or (2) $\mathcal{A}$ makes a query using user values $\mathit{aid}, \mathit{pw}$ and γ such that it has previously queried $\mathrm{RO}_1(\mathit{aid}, \mathit{pw})$ or $\mathrm{RO}_2(\mathit{aid}, \mathit{pw}, \gamma)$. This implies that $(\mathit{aid}, \mathit{pw}) \in S_{H1} \cap S_I$ or $(\mathit{aid}, \mathit{pw}, \gamma) \in S_{H2} \cap S_{rw}$. Hence,

$$\Pr[\mathsf{Bad}] = \Pr\left[G_2'\right] = \Pr\left[G_2''\right]. \tag{10}$$

Next, we construct an adversary $\mathcal{B}_{pg}$ against the password-guessing game (see Figure 4) for the distribution $\mathcal{PW}_n$ such that

$$\mathbf{Adv}^{\mathrm{PG}}_{n,\mathcal{PW}_n}(\mathcal{B}_{pg}) \geq \Pr\left[G_2''\right]. \tag{11}$$

On a high level, adversary $\mathcal{B}_{pg}$ will use the RO queries from $\mathcal{A}$ to guess passwords: Whenever adversary $\mathcal{A}$ makes a query $\mathrm{RO}_1(a, b)$ or $\mathrm{RO}_2(a, b, c)$, $\mathcal{B}_{pg}$ lets $S_H \overset{\cup}{\leftarrow} \{(a, b)\}$. When $\mathcal{A}$ halts, $\mathcal{B}_{pg}$ checks for every element $(a, b) \in S_H$ if there exists a user i_u s.t. the account ID of i_u is a. If so, it queries $\mathrm{TEST}(i_u, b)$. This way, $\mathcal{B}_{pg}$ makes a password guess for every element (a, b) resp. (a, b, c) in S_{H1} resp. S_{H2} of G_2'' where a corresponds to an account ID of a registered user. In particular, it will query $\mathrm{TEST}(i_u, b)$ for each $(a, b) \in S_I \cap S_{H1}$ and each $(a, b, c) \in S_{rw} \cap S_{H2}$ such that $T_{uaid}[i_u] = a$. Since S_I and S_{rw} only contain account IDs and passwords of honestly registered users not in S_{comp}, adversary $\mathcal{B}_{pg}$ wins game $\mathbf{G}^{\mathrm{PG}}_{n,\mathcal{PW}_n}$ if $(S_I \cap S_{H1}) \cup (S_{rw} \cap S_{H2}) \neq \emptyset$, proving Equation (11).

Formally, we also need to take care that adversary $\mathcal{B}_{pg}$ can properly simulate game G_2'' for $\mathcal{A}$. This is possible thanks to the selective compromises in the password guessing game, and the fact that for non-compromised users, the OPRF values γ and rw are independent from the user's password in game G_2''. The selective compromises are used to handle queries from $\mathcal{A}$ on $i_u \in S_{\mathsf{comp}}$: In the first stage of the simulation, adversary $\mathcal{B}_{pg}$ runs $\mathcal{A}$ to get S_{comp} and returns this to its own challenger, receiving the vector $\mathbf{pw}_{\mathsf{comp}}$ of compromised passwords back. In the second stage, $\mathcal{B}_{pg}$ uses $\mathbf{pw}_{\mathsf{comp}}[i_u]$ in place of $\mathbf{pw}[i_u]$ to handle queries from $\mathcal{A}$ on $i_u \in S_{\mathsf{comp}}$.

[16] Note that this relies on the assumption that *aid*s are unique per user, so that there is no overlap between game queries to RO_1 and RO_2 for users in S_{comp} and queries from $\mathcal{A}$.

```
Games $\boxed{\mathrm{G}_1}$, $\mathrm{G}_2$:
  ⋮

AREG1($i_u$, $in_C$):
  ⋮
3  if $i_u \in \mathrm{S}_{\mathsf{comp}}$: $\gamma \leftarrow (\mathrm{RO}_1(aid, pw))^k$; $rw \leftarrow \mathrm{RO}_2(aid, pw, \gamma)$
4  else:
5      $\mathrm{T_I}[aid, pw] \leftarrow\!\!\$\ G$   // aid is unique and AREG1 can only be called once, so no resampling
6      if $\mathrm{T_{H1}}[aid, pw] \neq \bot$:
           bad ← true; $\boxed{\mathrm{T_I}[aid, pw] \leftarrow \mathrm{T_{H1}}[aid, pw]}$
7      $\gamma \leftarrow (\mathrm{T_I}[aid, pw])^k$
8      $\mathrm{T_{rw}}[aid, pw, \gamma] \leftarrow\!\!\$\ \{0,1\}^{kl}$
9      if $\mathrm{T_{H2}}[aid, pw, \gamma] \neq \bot$:
           bad ← true; $\boxed{\mathrm{T_{rw}}[aid, pw, \gamma] \leftarrow \mathrm{T_{H2}}[aid, pw, \gamma]}$
10     $rw \leftarrow \mathrm{T_{rw}}[aid, pw, \gamma]$
  ⋮

AUTH1($i_u$, $in_C$):
  ⋮
15 if $i_u \in \mathrm{S}_{\mathsf{comp}}$: $\alpha \leftarrow (\mathrm{RO}_1(aid, pw))^r$ else: $\alpha \leftarrow (\mathrm{T_I}[aid, pw])^r$
  ⋮

AUTH2($i_p$, $m_S$)   // Oracle STEP for $\mathsf{CSS}^{(C:2)}_{\mathsf{auth}}$:
  ⋮
22 if $i_u \in \mathrm{S}_{\mathsf{comp}}$: $rw \leftarrow \mathrm{RO}_2(aid, pw, \gamma)$
23 else:
24     if $\mathrm{T_{rw}}[aid, pw, \gamma] = \bot$: $\mathrm{T_{rw}}[aid, pw, \gamma] \leftarrow\!\!\$\ \{0,1\}^{kl}$
25     if $\mathrm{T_{H2}}[aid, pw, \gamma] \neq \bot$:
           bad ← true; $\boxed{\mathrm{T_{rw}}[aid, pw, \gamma] \leftarrow \mathrm{T_{H2}}[aid, pw, \gamma]}$
26     $rw \leftarrow \mathrm{T_{rw}}[aid, pw, \gamma]$
  ⋮
```

```
Game $\mathrm{G}_0$, $\mathrm{G}_3$

RO$_1(a, b)$:
27 if $\mathrm{T_{H1}}[a, b] = \bot$:
28     $\mathrm{T_{H1}}[a, b] \leftarrow\!\!\$\ G$
29 return $\mathrm{T_{H1}}[a, b]$

RO$_2(a, b, c)$:
30 if $\mathrm{T_{H2}}[a, b, c] = \bot$:
31     $\mathrm{T_{H2}}[a, b, c] \leftarrow\!\!\$\ \{0,1\}^{kl}$
32 return $\mathrm{T_{H2}}[a, b, c]$
```

```
Game $\boxed{\mathrm{G}_1}$, $\mathrm{G}_2$

RO$_1(a, b)$:
33 if $\mathrm{T_{H1}}[a, b] = \bot$: $\mathrm{T_{H1}}[a, b] \leftarrow\!\!\$\ G$
34 if $\mathrm{T_I}[a, b] \neq \bot$:
35     bad ← true; $\boxed{\mathrm{T_{H1}}[a, b] \leftarrow \mathrm{T_I}[a, b]}$
36 return $\mathrm{T_{H1}}[a, b]$

RO$_2(a, b, c)$:
37 if $\mathrm{T_{H2}}[a, b, c] = \bot$: $\mathrm{T_{H2}}[a, b, c] \leftarrow\!\!\$\ \{0,1\}^{kl}$
38 if $\mathrm{T_{rw}}[a, b, c] \neq \bot$:
39     bad ← true; $\boxed{\mathrm{T_{H2}}[a, b, c] \leftarrow \mathrm{T_{rw}}[a, b, c]}$
40 return $\mathrm{T_{H2}}[a, b, c]$
```

Figure 22: Games and additional oracles for the first game hops in proof of Theorem 6.1. G is a group of prime order q with generator g.

```
$\underline{\text{Games } \mathrm{G}_2', \mathrm{G}_2'':}$
    ⋮
    return bad                                                                    // $\mathrm{G}_2'$
    return $(\mathrm{S_I} \cap \mathrm{S_{H1}}) \cup (\mathrm{S_{rw}} \cap \mathrm{S_{H2}}) \neq \emptyset$                               // $\mathrm{G}_2''$

$\underline{\text{AREG1}(i_u, in_C):}$
    ⋮
    $k \leftarrow\$ \mathbb{Z}_q$
    if $i_u \in \mathrm{S_{comp}}$:
        if $\mathrm{T_{H1}}[aid, pw] = \perp$: $\mathrm{T_{H1}}[aid, pw] \leftarrow\$ G$
        $\gamma \leftarrow (\mathrm{T_{H1}}[aid, pw])^k$
        if $\mathrm{T_{H2}}[aid, pw, \gamma] = \perp$: $\mathrm{T_{H2}}[aid, pw, \gamma] \leftarrow\$ \{0,1\}^{kl}$
        $rw \leftarrow \mathrm{T_{H2}}[aid, pw, \gamma]$
    else: $\mathrm{T_I}[aid, pw] \leftarrow\$ G$
        if $\mathrm{T_{H1}}[aid, pw] \neq \perp$: bad $\leftarrow$ true  // $(aid, pw) \in \mathrm{S_{H1}}$           // $\mathrm{G}_2'$
        $\mathrm{S_I} \xleftarrow{\cup} \{(aid, pw)\}$                                                 // $\mathrm{G}_2''$
        $\gamma \leftarrow (\mathrm{T_I}[aid, pw])^k$
        $\mathrm{T_{rw}}[aid, pw, \gamma] \leftarrow\$ \{0,1\}^{kl}$
        if $\mathrm{T_{H2}}[aid, pw, \gamma] \neq \perp$: bad $\leftarrow$ true;  // $(aid, pw, \gamma) \in \mathrm{S_{H2}}$   // $\mathrm{G}_2'$
        $\mathrm{S_{rw}} \xleftarrow{\cup} \{(aid, pw, \gamma)\}$                                          // $\mathrm{G}_2''$
        $rw \leftarrow \mathrm{T_{rw}}[aid, pw, \gamma]$
    ⋮

$\underline{\text{AUTH1}(i_u, in_C):}$
    ⋮
    if $i_u \in \mathrm{S_{comp}}$: $\alpha \leftarrow (\mathrm{T_{H1}}[aid, pw])^r$ else: $\alpha \leftarrow (\mathrm{T_I}[aid, pw])^r$
    ⋮

$\underline{\text{AUTH2}(i_p, m_S):}$
    ⋮
    if $i_u \in \mathrm{S_{comp}}$:
        if $\mathrm{T_{H2}}[aid, pw, \gamma] = \perp$: $\mathrm{T_{H2}}[aid, pw, \gamma] \leftarrow\$ \{0,1\}^{kl}$
        $rw \leftarrow \mathrm{T_{H2}}[aid, pw, \gamma]$
    else:
        if $\mathrm{T_{rw}}[aid, pw, \gamma] = \perp$: $\mathrm{T_{rw}}[aid, pw, \gamma] \leftarrow\$ \{0,1\}^{kl}$
        if $\mathrm{T_{H2}}[aid, pw, \gamma] \neq \perp$: bad $\leftarrow$ true;  // $(aid, pw, \gamma) \in \mathrm{S_{H2}}$   // $\mathrm{G}_2'$
        $\mathrm{S_{rw}} \xleftarrow{\cup} \{(aid, pw, \gamma)\}$                                          // $\mathrm{G}_2''$
        $rw \leftarrow \mathrm{T_{rw}}[aid, pw, \gamma]$
    ⋮
```

```
$\underline{\text{Games } \mathrm{G}_2', \mathrm{G}_2''}$

$\underline{\mathrm{RO}_1(a, b):}$
    if $\mathrm{T_{H1}}[a, b] = \perp$: $\mathrm{T_{H1}}[a, b] \leftarrow\$ G$
    if $\mathrm{T_I}[a, b] \neq \perp$: bad $\leftarrow$ true  // $(a, b) \in \mathrm{S_I}$     // $\mathrm{G}_2'$
    $\mathrm{S_{H1}} \xleftarrow{\cup} \{(a, b)\}$                                   // $\mathrm{G}_2''$
    return $\mathrm{T_{H1}}[a, b]$

$\underline{\mathrm{RO}_2(a, b, c):}$
    if $\mathrm{T_{H2}}[a, b, c] = \perp$: $\mathrm{T_{H2}}[a, b, c] \leftarrow\$ \{0,1\}^{kl}$
    if $\mathrm{T_{rw}}[a, b, c] \neq \perp$: bad $\leftarrow$ true // $(a, b, c) \in \mathrm{S_{rw}}$   // $\mathrm{G}_2'$
    $\mathrm{S_{H2}} \xleftarrow{\cup} \{(a, b, c)\}$                                // $\mathrm{G}_2''$
    return $\mathrm{T_{H2}}[a, b, c]$
```

Figure 23: Games G_2' and G_2'' for the first game hops in proof of Theorem 6.1. G is a group of prime order q with generator g.

If $i_u \in (\mathsf{S}_{\mathsf{hon}} \setminus \mathsf{S}_{\mathsf{comp}})$, $\mathcal{B}_{\text{pg}}$ instead samples γ and/or *rw* (depending on the oracle) uniformly at random and stores them for consistency in $\mathrm{T_I}[i_u]$ and $\mathrm{T_{rw}}[i_u, \gamma]$, respectively. Note here that swapping *aid*, *pw* for i_u in the key for the tables is sound thanks to the assumption that account IDs are globally unique and the fact that the game enforces a single *aid* per registered user. With this strategy, adversary $\mathcal{B}_{\text{pg}}$ perfectly simulates game G_2'' for $\mathcal{A}$, making at most $\mathcal{Q}_{\mathrm{RO}_1}(\mathcal{A}) + \mathcal{Q}_{\mathrm{RO}_2}(\mathcal{A})$ queries to oracle TEST.

Game G_4: Replace k_{kek} and k_{mac} with random.
We begin with a change in bookkeeping. Let G_3' be identical to G_3, except that table $\mathrm{T_{rw}}$ tracking the *rw* values of honest clients is indexed by i_u, γ rather than *aid*, *pw*, γ. Thanks to the one-to-one mapping between *aid* and i_u, this has no effect on the functionality of the game, implying $\Pr[\mathrm{G}_3] = \Pr[\mathrm{G}_3']$.

Next, G_4 replaces k_{kek} and k_{mac}, derived from *rw* using F on lines 4, 7 of $\mathsf{CSS}_{\mathsf{areg}}$, and 23, 24 of $\mathsf{CSS}_{\mathsf{auth}}^{(C:2)}$ (Figure 8) with consistently sampled random strings of length *kl*. We construct an adversary $\mathcal{B}_{\mathsf{prf}}$ such that

$$\Pr[\mathrm{G}_3'] - \Pr[\mathrm{G}_4] \leq \mathbf{Adv}_{\mathsf{F}}^{\mathrm{PRF}}(\mathcal{B}_{\mathsf{prf}}). \tag{12}$$

Adversary $\mathcal{B}_{\mathsf{prf}}$ acts as the challenger in game G_3'. In particular, it samples a bit b and uses it to determine which file to encrypt in queries from $\mathcal{A}$ to oracle CHALL. In contrast to the challenger in game G_3', adversary $\mathcal{B}_{\mathsf{prf}}$ does not itself sample the table entries in $\mathrm{T_{rw}}$ used as keys for F. Instead, it calls oracle NEW for every new i_u, γ combination (where $i_u \notin \mathsf{S}_{\mathsf{comp}}$) that it sees in $\mathcal{A}$'s queries to AREG1 and AUTH2, storing the index of the new key in place of the key itself in table $\mathrm{T_{rw}}[i_u, \gamma]$. Furthermore, instead of computing F itself for uncompromised clients, it calls oracle $\mathrm{FN}(\mathrm{T_{rw}}[i_u, \gamma], x)$ for $x \in \{\text{“kek”}, \text{“mac”}\}$ to compute k_{kek} and k_{mac}. This way, $\mathcal{B}_{\mathsf{prf}}$ simulates game G_3' when playing $\mathbf{G}_{\mathsf{F}}^{\text{PRF-1}}$ and G_4 when playing $\mathbf{G}_{\mathsf{F}}^{\text{PRF-0}}$, making at most $\mathcal{Q}_{\mathrm{AREG1}}(\mathcal{A}) + \mathcal{Q}_{\mathrm{AUTH1}}(\mathcal{A})$ queries to both oracles NEW and FN. (Note that $\mathcal{Q}_{\mathrm{AUTH2}}(\mathcal{A}) \leq \mathcal{Q}_{\mathrm{AUTH1}}(\mathcal{A})$, since the second step of a protocol can only be run if it the first has been executed, allowing us to bound the number subsequent protocol steps by the initial queries.)

When $\mathcal{A}$ halts and returns bit b', adversary $\mathcal{B}_{\mathsf{prf}}$ halts and returns 1 if $b' = b$, else it returns 0. This gives Equation (12).

Game G_5: Replace encrypted master key with random.
Next, in game G_5, the goal is to apply AEAD security for the encryption of the client master key k_{mk} in $\mathsf{CSS}_{\mathsf{areg}}^{(C:1)}$ for honest users. All of the following changes apply only if $i_u \notin \mathsf{S}_{\mathsf{comp}}$; otherwise the game remains unchanged.

In game G_5, the encryption c_{mk} of the master key is replaced by a randomly sampled string c_{mk}' of length $|c_{mk}|$ in line 6 of $\mathsf{CSS}_{\mathsf{areg}}^{(C:1)}$. Additionally, the game stores the master key k_{mk} in table $\mathrm{T}_{\mathsf{AEAD}}[(i_u, \gamma), n_{mk}, c_{mk}', (aid, \text{“mk”})]$, and instead of using Dec to decrypt c_{mk} in line 29 of $\mathsf{CSS}_{\mathsf{auth}}^{(C:3)}$, the table entry corresponding to the inputs from the server is used. Note that this means that if the table entry has not been initialized in $\mathsf{CSS}_{\mathsf{areg}}$, k_{mk} is set to $\perp$ in $\mathsf{CSS}_{\mathsf{auth}}$.

We construct an adversary $\mathcal{B}_{\mathsf{aead}}^1$ against the IND\$ security of AEAD such that

$$\Pr[\mathrm{G}_4] - \Pr[\mathrm{G}_5] \leq \mathbf{Adv}_{\mathsf{AEAD}}^{\mathrm{IND\$}}(\mathcal{B}_{\mathsf{aead}}^1). \tag{13}$$

Adversary $\mathcal{B}_{\mathsf{aead}}^1$ acts like the challenger in game G_4, with the following exceptions. First, instead of sampling k_{kek} at random for each i_u, γ pair s.t. $i_u \notin \mathsf{S}_{\mathsf{comp}}$ in AREG1 and the second step of AUTH, adversary $\mathcal{B}_{\mathsf{aead}}^1$ queries oracle NEW and stores the index of the new key in $\mathrm{T}_{\mathsf{F}}[i_u, \gamma, \text{“kek”}]$.

This gives $\mathcal{Q}_{\text{NEW}}(\mathcal{B}^1_{\mathsf{aead}}) \leq \mathcal{Q}_{\text{AREG1}}(\mathcal{A}) + \mathcal{Q}_{\text{AUTH1}}(\mathcal{A})$. Second, instead of encrypting k_{mk} itself in AREG1, adversary $\mathcal{B}^1_{\mathsf{aead}}$ queries oracle ENC in the IND\$ game with key index $\mathrm{T}_{\mathsf{F}}[i_u, \gamma, \text{“kek”}]$ and message input k_{mk} and uses the resulting ciphertext as c_{mk}. Note that thanks to the precondition $i_u \notin \mathrm{S}_{\mathsf{reg}}$ in AREG1 and the fact that $\mathsf{CSS}_{\mathsf{areg}}$ always succeeds on the client, $\mathcal{A}$ can query oracle AREG1 at most once per i_u. Hence adversary $\mathcal{B}^1_{\mathsf{aead}}$ will make at most one ENC query per key in the IND\$ game, ensuring that nonce collisions are not a problem. Last, instead of decrypting c_{mk} in the third step of AUTH, $\mathcal{B}^1_{\mathsf{aead}}$ queries oracle DEC in its own game with key index $\mathrm{T}_{\mathsf{F}}[i_u, \gamma, \text{“kek”}]$. This way, $\mathcal{B}^1_{\mathsf{aead}}$ simulates game G_4 and G_5 for $\mathcal{A}$ when the hidden bit in the IND\$ game is 1 and 0, respectively. We have $\mathcal{Q}_{\text{ENC}}(\mathcal{B}^1_{\mathsf{aead}}) \leq \mathcal{Q}_{\text{AREG1}}(\mathcal{A})$ and $\mathcal{Q}_{\text{DEC}}(\mathcal{B}^1_{\mathsf{aead}}) \leq \mathcal{Q}_{\text{AUTH1}}(\mathcal{A})$.

Let b denote the bit sampled by $\mathcal{B}^1_{\mathsf{aead}}$ at the start of the simulation and $b_{\mathcal{B}}$ the bit sampled by the challenger in $\mathbf{G}^{\text{IND\$}}_{\mathsf{AEAD}}$. When $\mathcal{A}$ halts and returns b', $\mathcal{B}^1_{\mathsf{aead}}$ returns 1 if $b' = b$, otherwise it returns 0. This gives Equation (13).

Game G_6: Avoid nonce collisions.

In G_6, we take care of nonce collisions for AEAD encryptions. These occur with a probability that relates to the nonce length nl and the number of operations per user which sample nonces. In CSS, the nonces for file resp. file key encryptions are sampled in subroutines $\mathsf{putFile}$ resp. $\mathsf{putHeader}$, which are called by protocols $\mathsf{CSS}_{\mathsf{put}}$ (both) and $\mathsf{CSS}_{\mathsf{upd}}$ resp. $\mathsf{CSS}_{\mathsf{accept}}$. Formally, we introduce a bad flag which is set to true whenever nonces collide in both G_5 and G_6, and game G_6 only differs from G_5 in that if bad is set, the nonce is resampled such that it is guaranteed to be fresh. That is, line 1 of both $\mathsf{putFile}$ and $\mathsf{putHeader}$ in Figure 10 is replaced by

```
G_5, [G_6]: putFile, putHeader:
1  $n \leftarrow\$ \{0,1\}^{nl}$
2  if $n \in \mathrm{S}^{\mathsf{pF}}_{n,i_u}\ \mathrm{S}^{\mathsf{pH}}_{n,i_u}$:
       $\mathsf{bad} \leftarrow \mathsf{true};\ \boxed{n \leftarrow\$ \{0,1\}^{nl} \setminus \mathrm{S}^{\mathsf{pF}}_{n,i_u}\ \mathrm{S}^{\mathsf{pH}}_{n,i_u}}$        // Boxed code only in G_6
3  $\mathrm{S}^{\mathsf{pF}}_{n,i_u}\ \mathrm{S}^{\mathsf{pH}}_{n,i_u} \xleftarrow{\cup} \{n\}$
```

Here $i_u \leftarrow \mathrm{T}_{\mathrm{u}}[i_c]$ is the user index tracked by the game corresponding to the client index i_c input by $\mathcal{A}$ to oracles PUT1 and ACPT1. Since the games are identical until the bad flag is set, $\Pr[\mathrm{G}_5] - \Pr[\mathrm{G}_6] \leq \Pr[\mathrm{G}_5 \text{ sets } \mathsf{bad}]$ by the fundamental lemma of game-playing [11].

Let $\mathsf{Bad}_{\mathsf{pF}}$ resp. $\mathsf{Bad}_{\mathsf{pH}}$ be the event that G_5 sets bad in $\mathsf{putFile}$ resp. $\mathsf{putHeader}$. We have that $\Pr[\mathrm{G}_5 \text{ sets } \mathsf{bad}] = \Pr[\mathsf{Bad}_{\mathsf{pF}} \vee \mathsf{Bad}_{\mathsf{pH}}] = \Pr[\mathsf{Bad}_{\mathsf{pF}}] + \Pr[\mathsf{Bad}_{\mathsf{pH}}]$, since the two bad events are independent. We now proceed to bound $\Pr[\mathsf{Bad}_{\mathsf{pF}}]$ and $\Pr[\mathsf{Bad}_{\mathsf{pH}}]$. Recall that $q_{\text{PUT1}}(\mathcal{A})$, $q_{\text{UPD1}}(\mathcal{A})$ and $q_{\text{ACPT1}}(\mathcal{A})$ are the maximum number of queries made by $\mathcal{A}$ to oracles PUT1 and ACPT1 *per user* i_u, respectively. $\Pr[\mathsf{Bad}_{\mathsf{pH}}]$ is then bounded from above by the collision probability $C(2^{nl}, q_{\text{PUT1}}(\mathcal{A}) + q_{\text{ACPT1}}(\mathcal{A}))$, which denotes the probability that from $q_{\text{PUT1}}(\mathcal{A}) + q_{\text{ACPT1}}(\mathcal{A})$ elements sampled uniformly at random out of a set of size 2^{nl}, any two collide. By a birthday bound, this gives

$$\Pr[\mathsf{Bad}_{\mathsf{pH}}] \leq \frac{(q_{\text{PUT1}}(\mathcal{A}) + q_{\text{ACPT1}}(\mathcal{A}))^2}{2^{nl+1}}.$$

For $\mathsf{Bad}_{\mathsf{pF}}$, the tightest upper bound would be achieved by considering the number of nonces sampled *per file*, which is at most the number of calls to UPD1 for that file (across all users), plus one (for the nonce sampled when the file is first put). However, since the number of update calls per file is at most the total number of update queries $\mathcal{Q}_{\text{UPD1}}(\mathcal{A})$ (across all files), we bound $\Pr[\mathsf{Bad}_{\mathsf{pF}}]$

by $C(2^{nl}, \mathcal{Q}_{\text{UPD1}}(\mathcal{A}) + 1)$, for simplicity. This gives

$$\Pr\left[\mathsf{Bad}_{\mathsf{pF}}\right] \leq \frac{(\mathcal{Q}_{\text{UPD1}}(\mathcal{A}) + 1)^2}{2^{nl+1}} \ ,$$

again by a birthday bound. Together, this gives

$$\Pr[\mathrm{G}_5] - \Pr[\mathrm{G}_6] \leq \frac{(\mathcal{Q}_{\text{UPD1}}(\mathcal{A}) + 1)^2 + (q_{\text{PUT1}}(\mathcal{A}) + q_{\text{ACPT1}}(\mathcal{A}))^2}{2^{nl+1}}. \tag{14}$$

Game G_7: Replace encrypted file keys with random.
Next, we apply AEAD security to the file key encryptions of honest users. Briefly, the aim of this step is to replace each file key ciphertext of honest users with a randomly sampled string of the same length, such that the key used to encrypt the file is independent from the ciphertext sent to the server.

Formally, game G_7 differs from G_6 in line 3 of putHeader (Figure 10), which in G_7 is replaced by

```
G7: putHeader:
3  $\cancel{c_{fk} \leftarrow \mathsf{Enc}(st_C.k_{mk}, n, k_f, md)}$; $c_{fk} \leftarrow\$ \{0,1\}^{|c_{fk}|}$; $\mathrm{T}_{k_f}[i_u, n, c_{fk}, md] \leftarrow k_f$
```

if called from oracle PUT1 or ACPT1 on input a client index i_c s.t. $\mathrm{T}_\mathrm{u}[i_c] \notin \mathrm{S}_{\mathsf{comp}}$. Note that putFile remains unchanged. In particular, the file is still encrypted with the file key k_f given as input to the subroutine. To ensure that file decryption works, game G_7 also replaces line 17 of getHeader and getFile (Figure 10) with

```
G7: getFile, getHeader:
16  $k_f \leftarrow \mathrm{T}_{k_f}[i_u, n, c_{fk}, md]$
```

if called from the oracles running procedures $\mathsf{CSS}_{\mathsf{upd}}$, $\mathsf{CSS}_{\mathsf{get}}$ and $\mathsf{CSS}_{\mathsf{share}}$ with a client index i_c s.t. $\mathrm{T}_\mathrm{u}[i_c] \notin \mathrm{S}_{\mathsf{comp}}$. Note that this means that k_f is set to $\perp$ if the table entry has not been initialized.

We construct an adversary $\mathcal{B}^2_{\mathsf{aead}}$ against the IND\$ security of AEAD such that

$$\Pr[\mathrm{G}_6] - \Pr[\mathrm{G}_7] \leq \mathbf{Adv}^{\mathrm{IND\$}}_{\mathsf{AEAD}}(\mathcal{B}^2_{\mathsf{aead}}). \tag{15}$$

Adversary $\mathcal{B}^2_{\mathsf{aead}}$ acts like the challenger in game G_6, with the following exceptions on operations for users $i_u \notin \mathrm{S}_{\mathsf{comp}}$. First, instead of sampling k_{mk} at random in AREG1, adversary $\mathcal{B}^2_{\mathsf{aead}}$ queries oracle NEW and stores the index of the new key in $\mathrm{T}_{\mathsf{AEAD}}[(i_u, \gamma), n_{mk}, c'_{mk}, (aid, \text{“mk”})]$, where γ, n_{mk}, c'_{mk} and aid are defined as in game G_6. This gives $\mathcal{Q}_{\text{NEW}}(\mathcal{B}^2_{\mathsf{aead}}) \leq \mathcal{Q}_{\text{AREG1}}(\mathcal{A})$. Second, when $\mathcal{A}$ runs procedure $\mathsf{CSS}_{\mathsf{auth}}$ via AUTH1 and STEP, adversary $\mathcal{B}^2_{\mathsf{aead}}$ sets $st_C.k_{mk} \leftarrow \mathrm{T}_{\mathsf{AEAD}}[(i_u, \gamma), n_{mk}, c'_{mk}, (aid, \text{“mk”})]$. (If the table entry is $\perp$, $\mathcal{B}^2_{\mathsf{aead}}$ will respond with $\perp$ to all future queries within the session which uses $st_C.k_{mk}$.)

Next, when $\mathcal{A}$ queries PUT1 or ACPT1, adversary $\mathcal{B}^2_{\mathsf{aead}}$ queries oracle $\text{ENC}(st_C.k_{mk}, n, k_f, md)$ to encrypt the newly sampled k_f and uses the resulting ciphertext as c_{fk}. Thanks to how nonce sampling in putHeader is handled in G_6, these queries from $\mathcal{B}^2_{\mathsf{aead}}$ are guaranteed to be nonce-respecting. Finally, instead of decrypting c_{fk} when simulating oracles which call getFile and getHeader, $\mathcal{B}^2_{\mathsf{aead}}$ queries oracle DEC in its own game with key index $st_C.k_{mk}$ on the inputs provided by $\mathcal{A}$. This way, $\mathcal{B}^2_{\mathsf{aead}}$ simulates game G_6 and G_7 for $\mathcal{A}$ when the hidden bit in the IND\$ game is 1 and 0, respectively. It makes at most $\mathcal{Q}_{\text{PUT1}}(\mathcal{A}) + \mathcal{Q}_{\text{ACPT1}}(\mathcal{A})$ queries to oracle ENC and at most $\mathcal{Q}_{\text{UPD1}}(\mathcal{A}) + \mathcal{Q}_{\text{SHARE1}}(\mathcal{A}) + \mathcal{Q}_{\text{GET1}}(\mathcal{A})$ to oracle DEC.

Let b denote the bit sampled by $\mathcal{B}^2_{\mathsf{aead}}$ at the start of the simulation and $b_{\mathcal{B}}$ the bit sampled by the challenger in $\mathbf{G}^{\mathrm{IND\$}}_{\mathsf{AEAD}}$. When $\mathcal{A}$ halts and returns b', $\mathcal{B}^1_{\mathsf{aead}}$ returns 1 if $b' = b$, otherwise it returns 0. This gives Equation (15).

File encryption: Bounding game G_7**.** The final reduction in the proof is concerned with the encryption of challenge files in G_7. We construct an adversary $\mathcal{B}^3_{\mathsf{aead}}$ against the IND-CCA security of AEAD such that

$$2 \cdot \Pr[\mathrm{G}_7] - 1 \leq \mathbf{Adv}^{\mathrm{IND\text{-}CCA}}_{\mathsf{AEAD}}(\mathcal{B}^3_{\mathsf{aead}}). \tag{16}$$

Adversary $\mathcal{B}^3_{\mathsf{aead}}$ acts as the challenger in G_7, with some exceptions. First, $\mathcal{B}^3_{\mathsf{aead}}$ does not sample a bit b. Second, $\mathcal{B}^3_{\mathsf{aead}}$ will use the oracles in its own game to handle some file operations, as follows. Let i_u be the user index of the client for which $\mathcal{A}$ runs a protocol through its oracle queries. We consider three cases for operations on a file with file ID *fid*:

(a) $\mathit{fid} \notin \mathrm{S}_{\mathsf{chall}}$: In this case, $\mathcal{B}^3_{\mathsf{aead}}$ will sample the file key itself when the file is first put, and perform subsequent file encryptions and decryptions exactly as the challenger in G_7.

(b) $(\mathit{fid} \in \mathrm{S}_{\mathsf{chall}}) \wedge (i_u \notin \mathrm{S}_{\mathsf{comp}} \cup \{\iota_{\mathrm{mal}}\})$: In this case, $\mathcal{B}^3_{\mathsf{aead}}$ will query oracle $\textsc{New}$ to request a new file key when the file is first put, and then use oracles $\textsc{Enc}$ and $\textsc{Dec}$ in game $\mathbf{G}^{\mathrm{IND\text{-}CCA}}_{\mathsf{AEAD}}$ to encrypt and decrypt, respectively.

(c) $(\mathit{fid} \in \mathrm{S}_{\mathsf{chall}}) \wedge (i_u \in \mathrm{S}_{\mathsf{comp}} \cup \{\iota_{\mathrm{mal}}\})$: If adversary $\mathcal{A}$ makes a query that falls into this case, $\mathcal{B}^3_{\mathsf{aead}}$ will either return $\perp$ to $\mathcal{A}$ (if that is what G_7 would do), or halt and return 0 (if it cannot continue to simulate).

Next, we will go through the implications of this strategy, protocol by protocol, focusing on cases (b) and (c), since that is where $\mathcal{B}^3_{\mathsf{aead}}$ differs from the challenger in G_7. Hence, the following applies only when $\mathit{fid} \in \mathrm{S}_{\mathsf{chall}}$.

Challenge put. In case (b), adversary $\mathcal{B}^3_{\mathsf{aead}}$ generates a new key via oracle $\textsc{New}$ in game $\mathbf{G}^{\mathrm{IND\text{-}CCA}}_{\mathsf{AEAD}}$, and stores the index i in $\mathrm{T}_{k_f}[i_u, n_{\mathit{fk}}, c_{\mathit{fk}}, \mathit{md}]$. Here c_{fk} is a random string of appropriate length (as per game G_7), n_{fk} is the file key nonce sampled in $\mathsf{putHeader}$ and $\mathit{md} = (\mathit{st}_C.\mathit{aid}, \mathit{fid}, \text{“fk”})$ includes the file ID *fid*. The file ciphertext is generated as $c_f \leftarrow \textsc{Enc}(i, n, f_0, f_1, (\mathit{fid}, \text{“file”}))$, where n is the nonce sampled in $\mathsf{putFile}$. Thanks to nonces being sampled without collisions (see G_6), adversary $\mathcal{B}^3_{\mathsf{aead}}$ is guaranteed to be nonce-respecting. This correctly simulates game G_7.

In case (c), adversary $\mathcal{B}^3_{\mathsf{aead}}$ halts and returns 0. We note that in this case, i_u is added to $\mathrm{S}_{\mathsf{challOwners}}$ on line 29 of Figure 6. Since $i_u \in \mathrm{S}_{\mathsf{comp}} \cup \{\iota_{\mathrm{mal}}\}$ in this case, $\mathsf{trivial}$ (defined on line 5 of Figure 5) will evaluate to true.

Challenge update. In case (b), $i \leftarrow \mathrm{T}_{k_f}[i_u, n_{\mathit{fk}}, c_{\mathit{fk}}, \mathit{md}]$, retrieved by $\mathcal{B}^3_{\mathsf{aead}}$ in $\mathsf{getHeader}$, either contains the index i of a key registered with oracle $\textsc{New}$ (if the file has been put or its sharing accepted by user i_u before, and $\mathcal{A}$ inputs the honest c_{fk} and n_{fk}) or is $\perp$ (if i_u is not an owner of this file or $\mathcal{A}$ does not provide honest inputs). In either case, $\mathcal{B}^3_{\mathsf{aead}}$ attempts to encrypt the challenge files with $c_f \leftarrow \textsc{Enc}(i, n, f_0, f_1, (\mathit{fid}, \text{“file”}))$ (returning $\perp$ if $i = \perp$). This correctly simulates game G_7.

In case (c), $\mathcal{B}^3_{\mathsf{aead}}$ returns $\perp$ in place of c_f if $\mathrm{T}_{k_f}[i_u, n_{\mathit{fk}}, c_{\mathit{fk}}, \mathit{md}] = \perp$. (This occurs if i_u is not an owner of *fid* from a prior put or accept operation, or if $\mathcal{A}$ does not provide honest inputs.) This correctly simulates game G_7. Otherwise, if the table entry exists, $\mathcal{B}^3_{\mathsf{aead}}$ halts and returns 0. We note that in this case, i_u was either added to $\mathrm{T}_{\mathrm{f}}[\mathit{fid}].\mathrm{U}$ when it first put the file in oracle $\textsc{Chall}$ and is therefore added to $\mathrm{S}_{\mathsf{challOwners}}$ on line 29, or, if the file was shared with i_u through the game, they were added to $\mathsf{S}_{\mathsf{challOwners}}$ as a recipient on line 19, both in Figure 6. In the case that i_u accepts the file as a share directly from the adversary, the adversary ι_{mal} is added to $\mathsf{S}_{\mathsf{challOwners}}$ on line 25 of Figure 6. In either case, $\mathsf{trivial} = \mathsf{true}$.

We note that in case (a), $\mathcal{B}^3_{\mathsf{aead}}$ will act as the challenger in the game and return $\perp$ if UPD1 is called through oracle CHALL on $\mathit{fid} \notin \mathsf{S}_{\mathsf{chall}}$. The reason that such queries are silenced is that $\mathcal{B}^3_{\mathsf{aead}}$ will have sampled the file key itself when the file was first put, and hence could not embed a challenge to its own game in this case. (That is, allowing this would encode a form of adaptive security, whch is not present in the selective game, as it would lead to a commitment issue.)

Get. In case (b), $\mathcal{B}^3_{\mathsf{aead}}$ lets $i \leftarrow \mathrm{T}_{k_f}[i_u, n_{\mathit{fk}}, c_{\mathit{fk}}, \mathit{md}]$ and attempts to decrypt the file ciphertext with $f \leftarrow \mathrm{DEC}(i, n_f, c_f, (\mathit{fid}, \text{“file”}))$. For the same reasons as in case (b) of Update, this correctly simulates game G_7.

In case (c), $\mathcal{B}^3_{\mathsf{aead}}$ returns $\perp$ in place of the file. Thanks to line 20 of Figure 5, this correctly simulates game G_7.

Share. In case (b), the file is being shared by an honest user. $\mathcal{B}^3_{\mathsf{aead}}$ then acts differently depending on the identity of the receiver: If the receiver is also honest ($i_r \notin \mathsf{S}_{\mathsf{comp}} \cup \{\iota_{\mathrm{mal}}\}$), $\mathcal{B}^3_{\mathsf{aead}}$ lets $i \leftarrow \mathrm{T}_{k_f}[i_u, n_{\mathit{fk}}, c_{\mathit{fk}}, \mathit{md}]$ and sends i over the OOB channel. For the same reasons as in case (b) of Update, this correctly simulates game G_7. If instead $i_r \in \mathsf{S}_{\mathsf{comp}} \cup \{\iota_{\mathrm{mal}}\}$, then $\mathcal{B}^3_{\mathsf{aead}}$ halts and returns 0. We note that in this case, i_r is added to $\mathsf{S}_{\mathsf{challOwners}}$ on line 19, Figure 6. Hence $\mathsf{trivial}$ is true in this case. Note that as a consequence of this strategy, the OOB channel $\mathrm{T}_{\mathrm{oob}}[(i_r, \cdot)]$ of a user $i_r \in \mathsf{S}_{\mathsf{comp}}$ only ever contains file keys sampled by $\mathcal{B}^3_{\mathsf{aead}}$ itself (for non-challenge files).

In case (c), $\mathcal{B}^3_{\mathsf{aead}}$ halts and returns 0. We note that in this case, i_u was added to $\mathsf{S}_{\mathsf{challOwners}}$ on line 19 of Figure 6, causing $\mathsf{trivial}$ to be true.

Accept. Note that we call the share-receiving user i_r here rather than i_u, to stay consistent with the game.

In case (b), if $\mathit{in}_C.\mathit{oob} = \perp$, $\mathcal{B}^3_{\mathsf{aead}}$ samples a file key nonce n_{fk} and ciphertext c_{fk} independently at random, as would the challenger in G_7. It then lets $\mathrm{T}_{k_f}[i_u, n_{\mathit{fk}}, c_{\mathit{fk}}, \mathit{md}] \leftarrow \mathrm{T}_{\mathrm{oob}}[(i_r, \mathit{fid})]$. This correctly simulates game G_7. If $\mathit{in}_C.\mathit{oob} \neq \perp$, then $\mathcal{B}^3_{\mathsf{aead}}$ halts and returns 0. We note that in this case, ι_{mal} was added to $\mathsf{S}_{\mathsf{challOwners}}$ on line 25 of Figure 6, causing $\mathsf{trivial}$ to be true.

In case (c), $\mathcal{B}^3_{\mathsf{aead}}$ checks if both $\mathit{in}_C.\mathit{oob}$ and $\mathrm{T}_{\mathrm{oob}}[(i_r, \mathit{fid})]$ are $\perp$. If so, it returns $\perp$. Otherwise, it halts and returns 0. We note that in this case, ι_{mal} is either added to $\mathsf{S}_{\mathsf{challOwners}}$ on line 25, Figure 6 (if $\mathit{in}_C.\mathit{oob} \neq \perp$), or i_r was already added on line 19 (when $\mathrm{T}_{\mathrm{oob}}[(i_r, \mathit{fid})]$ was initialized in SHARE1). Since $i_r \in \mathsf{S}_{\mathsf{comp}} \cup \{\iota_{\mathrm{mal}}\}$ in this case, this implies that $\mathsf{trivial}$ is true.

If adversary $\mathcal{A}$ makes any other query not covered above that sets $\mathsf{trivial}$ to true, adversary $\mathcal{B}^3_{\mathsf{aead}}$ halts and returns 0. Otherwise, adversary $\mathcal{B}^3_{\mathsf{aead}}$ simulates the game as above until adversary $\mathcal{A}$ halts and returns b', in which case $\mathcal{B}^3_{\mathsf{aead}}$ does the same. This means that $\mathcal{B}^3_{\mathsf{aead}}$ either returns the same bit guess as $\mathcal{A}$ (if $\neg\mathsf{trivial}$) or 0 (if $\mathsf{trivial}$). In terms of query counts, we have $\mathcal{Q}_{\mathrm{NEW}}(\mathcal{B}^3_{\mathsf{aead}}) \leq \mathcal{Q}_{\mathrm{CPUT1}}(\mathcal{A})$, $\mathcal{Q}_{\mathrm{ENC}}(\mathcal{B}^3_{\mathsf{aead}}) \leq \mathcal{Q}_{\mathrm{CPUT1}}(\mathcal{A}) + \mathcal{Q}_{\mathrm{CUPD1}}(\mathcal{A})$, $\mathcal{Q}_{\mathrm{DEC}}(\mathcal{B}^3_{\mathsf{aead}}) \leq \mathcal{Q}_{\mathrm{GET1}}(\mathcal{A})$.

Let G_7^b denote G_7 with hidden bit b. With this strategy, adversary $\mathcal{B}^3_{\mathsf{aead}}$ perfectly simulates G_7^b for $\mathcal{A}$ when playing game $\mathbf{G}^{\text{IND-CCA-}b}_{\mathsf{AEAD}}$, unless it halts before $\mathcal{A}$ does. In particular, $\mathcal{B}^3_{\mathsf{aead}}$ is able to simulate queries to oracle COMP for $i_u \in \mathsf{S}_{\mathsf{comp}}$; it samples pw and k_{mk} itself (like the challenger in G_7, thanks to prior game hops) and the OOB channel only contains file keys that $\mathcal{B}^3_{\mathsf{aead}}$ sampled itself, as noted in case (b) of Share.

Let T_b be the event that $\mathsf{trivial}$ is set to true by $\mathcal{A}$ when adversary $\mathcal{B}^3_{\mathsf{aead}}$ plays game $\mathbf{G}^{\text{IND-CCA-}b}_{\mathsf{AEAD}}$.

By applying the law of total probability to the IND-CCA advantage definition, we have

$$\begin{aligned}\mathbf{Adv}^{\text{IND-CCA}}_{\mathsf{AEAD}}(\mathcal{B}^3_{\mathsf{aead}}) &= \Pr\left[\mathbf{G}^{\text{IND-CCA-1}}_{\mathsf{AEAD}}\right] - \Pr\left[\mathbf{G}^{\text{IND-CCA-0}}_{\mathsf{AEAD}}\right] \\ &= \Pr\left[\,\mathbf{G}^{\text{IND-CCA-1}}_{\mathsf{AEAD}} \mid \mathsf{T}_1\,\right]\cdot\Pr\left[\mathsf{T}_1\right] - \Pr\left[\,\mathbf{G}^{\text{IND-CCA-0}}_{\mathsf{AEAD}} \mid \mathsf{T}_0\,\right]\cdot\Pr\left[\mathsf{T}_0\right] \\ &+ \Pr\left[\,\mathbf{G}^{\text{IND-CCA-1}}_{\mathsf{AEAD}} \mid \neg\mathsf{T}_1\,\right]\cdot\Pr\left[\neg\mathsf{T}_1\right] - \Pr\left[\,\mathbf{G}^{\text{IND-CCA-0}}_{\mathsf{AEAD}} \mid \neg\mathsf{T}_0\,\right]\cdot\Pr\left[\neg\mathsf{T}_0\right].\end{aligned}$$

Furthermore, since $\mathcal{B}^3_{\mathsf{aead}}$ returns 0 if $\mathsf{trivial} = \mathsf{true}$,

$$\Pr\left[\,\mathbf{G}^{\text{IND-CCA-1}}_{\mathsf{AEAD}} \mid \mathsf{T}_1\,\right]\cdot\Pr\left[\mathsf{T}_1\right] - \Pr\left[\,\mathbf{G}^{\text{IND-CCA-0}}_{\mathsf{AEAD}} \mid \mathsf{T}_0\,\right]\cdot\Pr\left[\mathsf{T}_0\right] = 0.$$

Next, by the definition of conditional probability, and since $\mathcal{B}^3_{\mathsf{aead}}$ returns the same bit as $\mathcal{A}$ in case $\neg\mathsf{trivial}$,

$$\begin{aligned}&\Pr\left[\,\mathbf{G}^{\text{IND-CCA-1}}_{\mathsf{AEAD}} \mid \neg\mathsf{T}_1\,\right]\cdot\Pr\left[\neg\mathsf{T}_1\right] - \Pr\left[\,\mathbf{G}^{\text{IND-CCA-0}}_{\mathsf{AEAD}} \mid \neg\mathsf{T}_0\,\right]\cdot\Pr\left[\neg\mathsf{T}_0\right] \\ &\qquad = \Pr\left[\mathcal{A} \Rightarrow 1 \text{ in } \mathrm{G}^1_7 \wedge \neg\mathsf{T}_1\right] - \Pr\left[\mathcal{A} \Rightarrow 1 \text{ in } \mathrm{G}^0_7 \wedge \neg\mathsf{T}_0\right] \\ &\qquad = 2\cdot\Pr\left[\mathrm{G}_7\right] - 1,\end{aligned}$$

where the last equality follows from standard advantage rewriting, conditioning on the value of b.

Conclusion. Combining Equations (8)–(16) gives

$$\begin{aligned}\mathbf{Adv}^{\text{C2CConfS}}_{\mathsf{CSS},n,\mathcal{PW}_n}(\mathcal{A}) \le 2\cdot\Big(&\mathbf{Adv}^{\text{PG}}_{\mathcal{PW}}(\mathcal{B}_{\text{pg}}) + \mathbf{Adv}^{\text{PRF}}_{\mathsf{F}}(\mathcal{B}_{\mathsf{prf}}) \\ &+ \frac{(\mathcal{Q}_{\text{UPD1}}(\mathcal{A})+1)^2 + (q_{\text{PUT1}}(\mathcal{A}) + q_{\text{ACPT1}}(\mathcal{A}))^2}{2^{nl+1}} \\ &+ \mathbf{Adv}^{\text{IND\$}}_{\mathsf{AEAD}}(\mathcal{B}^1_{\mathsf{aead}}) + \mathbf{Adv}^{\text{IND\$}}_{\mathsf{AEAD}}(\mathcal{B}^2_{\mathsf{aead}})\Big) + \mathbf{Adv}^{\text{IND-CCA}}_{\mathsf{AEAD}}(\mathcal{B}^3_{\mathsf{aead}})\,. \quad (17)\end{aligned}$$

We merge $\mathcal{B}^1_{\mathsf{aead}}$ and $\mathcal{B}^2_{\mathsf{aead}}$ into $\mathcal{B}^{12}_{\mathsf{aead}}$, as follows. $\mathcal{B}^{12}_{\mathsf{aead}}$ samples a bit $\beta \leftarrow\!\!\$\ \{0,1\}$ and runs $\mathcal{B}^{\beta+1}_{\mathsf{aead}}$, forwarding any oracle queries to its own oracles and returning whatever $\mathcal{B}^{\beta+1}_{\mathsf{aead}}$ returns. This gives

$$\mathbf{Adv}^{\text{IND-CCA}}_{\mathsf{AEAD}}(\mathcal{B}^{12}_{\mathsf{aead}}) = \frac{1}{2}\cdot\mathbf{Adv}^{\text{IND-CCA}}_{\mathsf{AEAD}}(\mathcal{B}^1_{\mathsf{aead}}) + \frac{1}{2}\cdot\mathbf{Adv}^{\text{IND-CCA}}_{\mathsf{AEAD}}(\mathcal{B}^2_{\mathsf{aead}}), \quad (18)$$

with query counts $\mathcal{Q}_{\text{ORC}}(\mathcal{B}^{12}_{\mathsf{aead}}) \le \max(\mathcal{Q}_{\text{ORC}}(\mathcal{B}^1_{\mathsf{aead}}) + \mathcal{Q}_{\text{ORC}}(\mathcal{B}^2_{\mathsf{aead}}))$ for $\text{ORC} \in \{\text{NEW}, \text{ENC}, \text{DEC}\}$. Combining Equations (17) and (18) gives the bound claimed in Theorem 6.1 with the following query counts in summary:

$$\mathcal{Q}_{\text{TEST}}(\mathcal{B}_{\text{pg}}) \le \mathcal{Q}_{\text{RO}_1}(\mathcal{A}) + \mathcal{Q}_{\text{RO}_2}(\mathcal{A}),$$

$$\mathcal{Q}_{\text{NEW}}(\mathcal{B}_{\mathsf{prf}}) \le \mathcal{Q}_{\text{FN}}(\mathcal{B}_{\mathsf{prf}}) \le \mathcal{Q}_{\text{AREG1}}(\mathcal{A}) + \mathcal{Q}_{\text{AUTH1}}(\mathcal{A})$$

$$\begin{cases}\mathcal{Q}_{\text{NEW}}(\mathcal{B}^{12}_{\mathsf{aead}}) & \le \mathcal{Q}_{\text{AREG1}}(\mathcal{A}) + \mathcal{Q}_{\text{AUTH1}}(\mathcal{A})\,, \\ \mathcal{Q}_{\text{ENC}}(\mathcal{B}^{12}_{\mathsf{aead}}) & \le \max(\mathcal{Q}_{\text{AREG1}}(\mathcal{A}), \mathcal{Q}_{\text{PUT1}}(\mathcal{A}) + \mathcal{Q}_{\text{ACPT1}}(\mathcal{A}))\,, \\ \mathcal{Q}_{\text{DEC}}(\mathcal{B}^{12}_{\mathsf{aead}}) & \le \max(\mathcal{Q}_{\text{AUTH1}}(\mathcal{A}), \mathcal{Q}_{\text{UPD1}}(\mathcal{A}) + \mathcal{Q}_{\text{SHARE1}}(\mathcal{A}) + \mathcal{Q}_{\text{GET1}}(\mathcal{A}))\,,\end{cases}$$

$$\begin{cases}\mathcal{Q}_{\text{NEW}}(\mathcal{B}^{3}_{\mathsf{aead}}) & \le \mathcal{Q}_{\text{CPUT1}}(\mathcal{A})\,, \\ \mathcal{Q}_{\text{ENC}}(\mathcal{B}^{3}_{\mathsf{aead}}) & \le \mathcal{Q}_{\text{CPUT1}}(\mathcal{A}) + \mathcal{Q}_{\text{CUPD1}}(\mathcal{A})\,, \text{ and} \\ \mathcal{Q}_{\text{DEC}}(\mathcal{B}^{3}_{\mathsf{aead}}) & \le \mathcal{Q}_{\text{GET1}}(\mathcal{A}).\end{cases}$$

C.2 Proof of Theorem 6.2

This section provides the details of the last steps of the proof of the selective client-to-client integrity of CSS. The proof is analogous to the proof of Theorem 6.1 up until and including G_6, with the difference that games G_0–G_6 provide the oracles named in game $\mathbf{G}^{\mathrm{C2CIntS}}_{\mathsf{CSS},n,\mathcal{PW}_n}$, and also match the finalize procedure thereof. That is, the games run (the stateful) adversary $\mathcal{A}$ with no oracle access to obtain $\mathrm{S}_{\mathsf{comp}}$, and then run $\mathcal{A}$ again with access to all oracles until the adversary halts. They return win; a flag set to true (on line 19 of Figure 5) if $\mathcal{A}$ makes a client successfully retrieve a file which has not been added via the PUT oracle, or which is only owned by honest users, but the file content does not match what is expected. Throughout, we let $\Pr[\,G_i\,]$ denote the probability that $G_i(\mathcal{A})$ returns true, and note that, by definition,

$$\mathbf{Adv}^{\mathrm{C2CIntS}}_{\mathsf{CSS},n,\mathcal{PW}_n}(\mathcal{A}) = \Pr[\,G_0\,]. \tag{19}$$

Furthermore, with only slight adaptations to the reductions provided in the proof of Theorem 6.1, we have

$$\Pr[G_0] - \Pr[G_3] \leq \mathbf{Adv}^{\mathrm{PG}}_{n,\mathcal{PW}_n}(\mathcal{B}_{\mathrm{pg}}), \tag{20}$$

$$\Pr[G_3] - \Pr[G_4] \leq \mathbf{Adv}^{\mathrm{PRF}}_{\mathsf{F}}(\mathcal{B}_{\mathsf{prf}}), \tag{21}$$

$$\Pr[G_4] - \Pr[G_5] \leq \mathbf{Adv}^{\mathrm{IND\$}}_{\mathsf{AEAD}}(\mathcal{B}^1_{\mathsf{aead}}),\ \text{and} \tag{22}$$

$$\Pr[G_5] - \Pr[G_6] \leq \frac{(\mathcal{Q}_{\mathrm{UPD1}}(\mathcal{A})+1)^2 + (q_{\mathrm{PUT1}}(\mathcal{A}) + q_{\mathrm{ACPT1}}(\mathcal{A}))^2}{2^{nl+1}}. \tag{23}$$

Adversary $\mathcal{B}_{\mathrm{pg}}$ works exactly as in the proof of Theorem 6.1. Adversaries $\mathcal{B}_{\mathsf{prf}}$ and $\mathcal{B}^1_{\mathsf{aead}}$ no longer sample a bit at the start of the game; instead, they run $\mathcal{A}$ and return 1 in their respective games if $\mathcal{A}$ sets win to true in the simulated game. This gives the bounds above.

We now provide the final steps in the proof of Theorem 6.2. Recall that in game G_6 (as per the proof of Theorem 6.1), the OPRF client secret rw, the key encryption key k_{kek} derived therefrom and the registered master key k_{mk} of all honest clients are uniformly random and independent from anything that the server sees. The master key recovered by honest clients during authentication and used in the subsequent sessions is additionally ensured to be equal to their registered master key. That is, malicious inputs from the server during authentication of an honest user will result in the client failing to derive the master key. Furthermore, in game G_6 all nonces sampled during file key and file encryption are guaranteed to be unique per encryption key.

Game G_7: Rule out completely forged file headers. In this step, we rule out the possibility of win being set to true for a file ID which has not been uploaded via oracle PUT1. To this end, we introduce a bad flag in G_6, which is set to true in procedure Check if either a get or share operation is successful, when run for an honest user, on a file ID which has no tracked content and no or only honest owners. That is, we cover both the case when the file ID is not tracked by the game at all, as well as when the file ID has not been uploaded through oracle PUT1, but shared or received, in which case the file has owners but no content.

In the game, we modify lines 18–20 and add lines 23–24 to subroutine Check in Figure 5, as shown below.

```
$\mathrm{G}_6$, $\boxed{\mathrm{G}_7}$: Check($\Pi, i_p, out_C, m'$):
17 if $\Pi = \Pi_{\mathsf{get}}$:
18   if $\mathrm{T_u}[i_c] \in \mathrm{S_{hon}} \setminus \mathrm{S_{comp}} \wedge out_C.dec \wedge \mathrm{T_f}[fid].\mathrm{U} \subseteq \mathrm{S_{hon}} \setminus \mathrm{S_{comp}}$:
19     if $\mathrm{T_f}[fid].\mathrm{F} = \emptyset$: bad $\leftarrow$ true; $\boxed{\textbf{return } \bot}$
20     win $\leftarrow out_C.f \notin \mathrm{T_f}[fid].\mathrm{F}$
21   return $(out_C.f, m')$
22 if ($\Pi = \Pi_{\mathsf{share}}$ and $out_C.dec$): $\mathrm{T_{oob}}[(i_r, fid)] \leftarrow out_C.oob$
23   if $\mathrm{T_u}[i_c] \in \mathrm{S_{hon}} \setminus \mathrm{S_{comp}} \wedge \mathrm{T_f}[fid].\mathrm{F} = \emptyset \wedge \mathrm{T_f}[fid].\mathrm{U} \subseteq \mathrm{S_{hon}} \setminus \mathrm{S_{comp}}$:
24     bad $\leftarrow$ true; $\boxed{\mathrm{T_{oob}}[(i_r, fid)] \leftarrow \bot;\ \mathrm{T_f}[fid].\mathrm{U} \leftarrow \mathrm{T_f}[fid].\mathrm{U} \setminus \{\mathrm{T_u}[i_c]\};\ \textbf{return } \bot}$
```

We additionally modify the return statement on line 11 of Figure 5 to

```
$\mathrm{G}_6$, $\boxed{\mathrm{G}_7}$:
11 if bad: $\boxed{\text{win} \leftarrow \text{false}}$
12 return win
```

where, in both of the above, the boxed code is not present in game G_6. We note that neither of these changes affect the winning condition, hence the modified G_6 is equivalent to the original and any adversary has the same probability of winning in both. We now create game G_7, which is identical to G_6, except that it contains the boxed code shown on lines 19, 24 and 11 above.

Let Bad denote the event that G_6 sets the bad flag. G_6 and G_7 are identical-until-Bad, so by the fundamental lemma of game playing [11], $\Pr[\mathrm{G}_6] - \Pr[\mathrm{G}_7] \leq \Pr[\mathsf{Bad}]$.

We construct an adversary $\mathcal{B}^2_{\mathsf{aead}}$ against the INT-CTXT security of AEAD such that

$$\Pr[\mathsf{Bad}] \leq \mathbf{Adv}^{\text{INT-CTXT}}_{\mathsf{AEAD}}(\mathcal{B}^2_{\mathsf{aead}}). \tag{24}$$

Adversary $\mathcal{B}^2_{\mathsf{aead}}$ acts as the challenger in game G_6, with the following differences on operations for users $i_u \notin \mathrm{S_{comp}}$. First, instead of sampling k_{mk} at random in AREG1, adversary $\mathcal{B}^2_{\mathsf{aead}}$ queries oracle NEW and stores the index of the new key in a table indexed by the relevant messages in the transcript of the registration protocol. (Cf. adversary $\mathcal{B}^2_{\mathsf{aead}}$ in the proof of Theorem 6.1, hop from G_6 to G_7.) This gives $\mathcal{Q}_{\mathrm{NEW}}(\mathcal{B}^2_{\mathsf{aead}}) \leq \mathcal{Q}_{\mathrm{AREG1}}(\mathcal{A})$. Second, when $\mathcal{A}$ runs procedure $\mathsf{CSS_{auth}}$ via AUTH1 and STEP, adversary $\mathcal{B}^2_{\mathsf{aead}}$ sets $st_C.k_{mk}$ to the indexed stored in the table. (If the table entry is $\bot$ due to malicious server messages during authentication, $\mathcal{B}^2_{\mathsf{aead}}$ will respond with $\bot$ to all future queries within the session which uses $st_C.k_{mk}$.)

Next, when adversary $\mathcal{A}$ queries oracles PUT1 or ACPT1, adversary $\mathcal{B}^2_{\mathsf{aead}}$ queries oracle $\mathrm{ENC}(st_C.k_{mk}, n, k_f, md)$ to encrypt the newly sampled k_f or the key from $in_C.oob$ or $\mathrm{T_{oob}}$, respectively. It uses the resulting ciphertext as c_{fk}. Thanks to how nonce sampling in putHeader is handled in G_6, these queries from $\mathcal{B}^2_{\mathsf{aead}}$ are guaranteed to be nonce-respecting. Finally, instead of decrypting c_{fk} when simulating oracles which call getFile and getHeader, $\mathcal{B}^2_{\mathsf{aead}}$ queries oracle DEC in its own game with key index $st_C.k_{mk}$ on the inputs provided by $\mathcal{A}$.

Adversary $\mathcal{B}^2_{\mathsf{aead}}$ handles queries to oracle COMP as would the challenger in game G_6, since – due to the fact that $\mathcal{A}$ announces all of its compromise queries in advance – adversary $\mathcal{B}^2_{\mathsf{aead}}$ knows all of the values to be returned in response to such a query. (It only does not know the master key of honest users, which are never revealed to the adversary in the selective security game.) This way, $\mathcal{B}^2_{\mathsf{aead}}$ simulates game G_6 for $\mathcal{A}$ and makes at most $\mathcal{Q}_{\mathrm{PUT1}}(\mathcal{A}) + \mathcal{Q}_{\mathrm{ACPT1}}(\mathcal{A})$ queries to oracle ENC and at most $\mathcal{Q}_{\mathrm{UPD1}}(\mathcal{A}) + \mathcal{Q}_{\mathrm{SHARE1}}(\mathcal{A}) + \mathcal{Q}_{\mathrm{GET1}}(\mathcal{A})$ to oracle DEC.

We claim that $\Pr[\mathsf{Bad}] \leq \mathbf{Adv}^{\text{INT-CTXT}}_{\mathsf{AEAD}}(\mathcal{B}^2_{\mathsf{aead}})$. To see this, observe that bad is set to true in G_6 if the adversary has successfully run the get or share protocols on a file ID *fid* such that

$$(\mathrm{T_u}[i_c] \in \mathrm{S_{hon}} \setminus \mathrm{S_{comp}}) \wedge (out_C.dec) \wedge (\mathrm{T_f}[fid].\mathrm{F} = \emptyset) \wedge (\mathrm{T_f}[fid].\mathrm{U} \subseteq \mathrm{S_{hon}} \setminus \mathrm{S_{comp}})\ .$$

In order, these conditions correspond to (1) the get or share operation is performed by an honest user, (2) the protocol concluded successfully on the client, (3) the file content is not tracked by the game, and (4) the file is only owned by honest users. We now claim that together, these conditions correspond to a file header forgery for the file ID *fid* which sets bad. To see this, note that point (3) implies that *fid* has not been queried to oracle PUT1 and consider the following two cases for point (4):

1. $\mathrm{T_f}[\mathit{fid}].\mathrm{U} = \emptyset$. (The set of file owners is empty.) In this case, *fid* has not been queried to oracle SHARE1 or ACPT1, and, hence, putHeader has not been performed for *fid*. However, a successful get or share operation means that protocol getFile or getHeader, respectively, has been run successfully for an honest user. This in turn means that adversary $\mathcal{B}^2_{\mathsf{aead}}$ queried oracle $\mathrm{DEC}(st_C.k_{mk}, n_{\mathit{fk}}, c_{\mathit{fk}}, md)$ on the master key index of the honest client, the file header $(n_{\mathit{fk}}, c_{\mathit{fk}})$ supplied by adversary $\mathcal{A}$ and the metadata $md = (st_C.aid, \mathit{fid}, \text{“fk”})$ and that the decryption succeeded. Since no file header has been put for *fid*, and hence not queried to oracle ENC by adversary $\mathcal{B}^2_{\mathsf{aead}}$, this is a valid forgery in the INT-CTXT game of AEAD. (Note that *fid* is part of the metadata *md*. Hence, *fid* being fresh implies that the decryption query does not count as a trivial attack on line 29 of game $\mathbf{G}^{\text{INT-CTXT}}_{\mathsf{AEAD}}$ in Figure 20.)

2. $\emptyset \neq \mathrm{T_f}[\mathit{fid}].\mathrm{U} \subseteq \mathrm{S_{hon}} \setminus \mathrm{S_{comp}}$. (The set of file owners is non-empty, but only contains honest users.) In this case, the file has been queried to oracle SHARE1 and/or ACPT1 (where the share query can be the one that sets bad) with only honest users as sharer and receiver, respectively. We identify the following sub-cases:

   (a) *fid* has not been queried to oracle ACPT1. Then it is again the case that putHeader has not been performed for *fid*, as putHeader is only run in $\mathsf{CSS_{put}}$ and $\mathsf{CSS_{accept}}$. Hence, by the same reasoning as in case 1, this implies that adversary $\mathcal{B}^2_{\mathsf{aead}}$ successfully forged a file header.

   (b) *fid* has been queried to oracle ACPT1. Then, by the precondition for oracle ACPT1, on line 21 of Figure 6, either $in_C.oob \neq \perp$, or $\mathrm{T_{oob}}[i_r, \mathit{fid}] \neq \perp$. If $in_C.oob \neq \perp$, then ι_{mal} is added to the set of owners, and hence the file is no longer only owned by honest users, ruling out this possibility. If $\mathrm{T_{oob}}[i_r, \mathit{fid}] \neq \perp$, then the OOB table entry must have been previously initialized by some user via a successful query to oracle SHARE1. This user must be honest (otherwise the file would be owned by the compromised sharer), implying that there was a point in time when *fid* had *not* yet been queried to oracle ACPT1, but was queried to oracle SHARE1 with all of the conditions necessary for setting bad being true. Hence, we are back in case 2(a).

Therefore, Bad implies that adversary $\mathcal{B}^2_{\mathsf{aead}}$ set win to true in game $\mathbf{G}^{\text{INT-CTXT}}_{\mathsf{AEAD}}$, yielding Equation (24).

Game G_8: Guess a forgery attempt. We now observe that in game G_7, the file ID *fid* of any successful forgery attempt must have appeared in a query to oracle PUT1. To see this, note that, thanks to the modified return statement in game G_7, the condition for setting the win flag can be changed to

```
G_7: Check($\Pi, i_p, out_C, m'$):
  ⋮
17 if $\Pi = \Pi_{\mathsf{get}}$:
18     win ← $(\mathrm{T_u}[i_c] \in \mathrm{S_{hon}} \setminus \mathrm{S_{comp}}) \wedge (out_C.dec) \wedge (\mathrm{T_f}[\mathit{fid}].\mathrm{F} \neq \emptyset) \wedge (out_C.f \notin \mathrm{T_f}[\mathit{fid}].\mathrm{F})$
           $\wedge (\mathrm{T_f}[\mathit{fid}].\mathrm{U} \subseteq \mathrm{S_{hon}} \setminus \mathrm{S_{comp}})$
```

without changing the outcome of the game. That is, adversary $\mathcal{A}$ can only win if it successfully gets a file *fid* through the client of an honest user, such that the file content is tracked (meaning the file has been put or updated at least once), the decrypted file content does not match any tracked version of the file, and the file is owned only by honest users. Now consider the possible ways in which the adversary can initialize $\mathrm{T_f}[\mathit{fid}].\mathrm{F}$, while ensuring that *fid* is only owned by honest users.

$\mathrm{T_f}[\mathit{fid}].\mathrm{F}$ is only set in oracles PUT1 and UPD1. In the latter, it is only updated if the calling user is already an owner of *fid* tracked in $\mathrm{T_F}[\mathit{fid}].\mathrm{U}$. $\mathrm{T_F}[\mathit{fid}].\mathrm{U}$, in turn, is only updated in oracles PUT1, SHARE1 and ACPT1. This leads to the following cases:

1. *fid* was first uploaded via oracle PUT1.

2. *fid* was updated via oracle UPD1 without first having been put. Then it must be the case that the updating user is already an owner of the file, and hence either:

   (a) The updating user was added as an owner in a query to oracle SHARE1 while $\mathrm{T_f}[\mathit{fid}].\mathrm{F} = \emptyset$. If the sharing user is compromised, the owner set no longer only contains honest users, so this case is ruled out. If instead the sharing client is an honest user, then by line 24 of game G_7 (as shown in the previous game hop above) the sharer is not added to $\mathrm{T_F}[\mathit{fid}].\mathrm{U}$. Hence this case is also ruled out.

   (b) The updating user was added as an owner while receiving *fid* as i_r in oracle ACPT1. By the precondition of oracle ACPT1, it is either the case that $in_C.oob \neq \bot$, or $\mathrm{T_{oob}}[i_r, \mathit{fid}] \neq \bot$. In the former case, ι_{mal} is added to the set of owners, and hence the file is no longer only owned by honest users, ruling out this possibility. In the latter case, $\mathrm{T_{oob}}[i_r, \mathit{fid}]$ must have been previously initialized by some user via a query to oracle SHARE1. But this is again ruled out by the same reasoning as in case 2(a), since either the sharer was compromised, in which case the file is no longer only owned by honest users, or $\mathrm{T_{oob}}[i_r, \mathit{fid}]$ has been set to $\bot$ on line 24 of game G_7.

Hence, we are left only with case 1, that *fid* was first uploaded via oracle PUT1. We now proceed to guess among the files uploaded via oracle PUT1 one that will constitute a successful forgery.

Let S_f denote the set of file IDs of all files uploaded to the server by an honest user during the course of the game. That is, $\mathrm{S}_f = \{\mathit{fid} \mid \mathit{fid}$ was queried to oracle PUT1 with $\mathrm{T_u}[i_c] \in \mathsf{S_{hon}} \setminus \mathsf{S_{comp}}\}$. Furthermore, let $\mathrm{S}_f^* \subseteq \mathrm{S}_f$ be the subset of uploaded file IDs for which $\mathcal{A}$ performed a successful forgery. That is, S_f^* contains the file IDs of every file for which win evaluated to true during the course of game G_7. (As noted above, win can only be set to true for file IDs in S_f.) Assume without loss of generality that $\mathrm{S}_f^* \neq \emptyset$. (Otherwise $\mathcal{A}$ has 0 advantage.) As per the theorem statement, let p be an upper bound on the number of queries to PUT1 by adversary $\mathcal{A}$. By definition, $\mathrm{p} \geq |\mathrm{S}_f| \geq |\mathrm{S}_f^*|$.

In game G_8, the challenger begins by sampling $j \in \{1, \ldots, \mathrm{p}\}$. When, during the came, $\mathcal{A}$ makes the j-th call to PUT1, the challenger remembers the file ID fid^* it uses. Furthermore, in the Check procedure, the win condition is modified to

```
G_8: Check(Π, $i_p$, $out_C$, $m'$):
  ⋮
17 if Π = Π_get:
18     win ← $(\mathrm{T_u}[i_c] \in \mathsf{S_{hon}} \setminus \mathsf{S_{comp}}) \wedge (out_C.dec) \wedge (\mathrm{T_f}[\mathit{fid}].\mathrm{F} \neq \bot) \wedge (out_C.f \notin \mathrm{T_f}[\mathit{fid}].\mathrm{F})$
           $\wedge(\mathrm{T_f}[\mathit{fid}].\mathrm{U} \subseteq \mathsf{S_{hon}} \setminus \mathsf{S_{comp}}) \wedge (\mathit{fid} = \mathit{fid}^*)$
```

That is, $\mathit{fid} = \mathit{fid}^*$ is added as a condition for win to be set to true. Hence, in game G_8, adversary $\mathcal{A}$ wins only if it performs a successful forgery of fid^*, the file ID uploaded in its j-th PUT1 query.

Let E be the event that $fid^* \in \mathrm{S}^*_f$. Then $\Pr[\,\mathsf{E}\,] \geq 1/\mathrm{p}$, since by definition $\mathrm{p} \geq |\mathrm{S}^*_f| > 0$ and j is sampled uniformly at random among the p queries to Put1. Furthermore, $\Pr[\,\mathrm{G}_8 \,|\, \mathsf{E}\,] = \Pr[\,\mathrm{G}_7 \,|\, \mathsf{E}\,] = \Pr[\,\mathrm{G}_7\,]$, since the outcome of the games is equivalent if E occurs, and the outcome of G_7 is independent of E. Hence

$$\Pr[\,\mathrm{G}_7\,] \leq \mathrm{p} \cdot \Pr[\,\mathrm{G}_8\,]. \tag{25}$$

Game G_9: Replace the encrypted file key of fid^* with random. Next, we apply AEAD security to the file key encryption of file fid^*, the file uploaded through the j-th Put1 query by an honest user. As in the hop to game G_7 in the confidentiality proof, the aim of this step is to replace the file key ciphertext with a randomly sampled string of the same length, such that the key used to encrypt the file is independent from the ciphertext sent to the server.

We construct game G_9, which replaces line 3 of putHeader (Figure 10) in G_8 with

```
G_9: putHeader:
3  $c_{fk} \leftarrow \mathsf{Enc}(st_C.k_{mk}, n, k_f, md)$; $c_{fk} \leftarrow\!\!\$\ \{0,1\}^{|c_{fk}|}$; $\mathrm{T}_{k_f}[i_u, n, c_{fk}, md] \leftarrow k_f$
```

when called for the j-th time in oracle Put1 for an honest user (where j is sampled by the challenger at the start of the game as per game G_8 of this proof), or subsequently when called through oracle Acpt1 for any honest user on input file ID fid^*, as recorded in the j-th put query. Note that putFile remains unchanged. In particular, the file is still encrypted with the file key k_f given as input to the subroutine. To ensure that file decryption works, game G_9 also replaces line 17 of getHeader and getFile (Figure 10) with

```
G_9: getFile, getHeader:
16 $k_f \leftarrow \mathrm{T}_{k_f}[i_u, n, c_{fk}, md]$
```

if called for an honest user from the oracles running procedures $\mathsf{CSS}_{\mathsf{upd}}$, $\mathsf{CSS}_{\mathsf{get}}$ and $\mathsf{CSS}_{\mathsf{share}}$ on input file ID fid^*. Note that this means that k_f is set to $\perp$ if the table entry has not been initialized.

We construct an adversary $\mathcal{B}^3_{\mathsf{aead}}$ against the IND\$ security of AEAD such that

$$\Pr[\mathrm{G}_8] - \Pr[\mathrm{G}_9] \leq \mathbf{Adv}^{\mathrm{IND\$}}_{\mathsf{AEAD}}(\mathcal{B}^3_{\mathsf{aead}}). \tag{26}$$

Adversary $\mathcal{B}^3_{\mathsf{aead}}$ acts like the challenger in game G_8, with the following exceptions on operations for users $i_u \notin \mathrm{S}_{\mathsf{comp}}$. First, instead of sampling k_{mk} at random in AReg1, adversary $\mathcal{B}^3_{\mathsf{aead}}$ queries oracle New and stores the index of the new key in table $\mathrm{T}_{\mathsf{AEAD}}[(i_u, \gamma), n_{mk}, c'_{mk}, (aid, \text{“mk”})]$, indexed by the relevant messages in the transcript of the registration protocol. (Cf. adversary $\mathcal{B}^2_{\mathsf{aead}}$ in the proof of Theorem 6.1, hop from G_6 to G_7.) This gives $\mathcal{Q}_{\mathrm{New}}(\mathcal{B}^3_{\mathsf{aead}}) \leq \mathcal{Q}_{\mathrm{AReg1}}(\mathcal{A})$. Second, when $\mathcal{A}$ runs procedure $\mathsf{CSS}_{\mathsf{auth}}$ via Auth1 and Step, adversary $\mathcal{B}^3_{\mathsf{aead}}$ sets $st_C.k_{mk} \leftarrow \mathrm{T}_{\mathsf{AEAD}}[(i_u, \gamma), n_{mk}, c'_{mk}, (aid, \text{“mk”})]$. (If the table entry is $\perp$, $\mathcal{B}^3_{\mathsf{aead}}$ will respond with $\perp$ to all future queries within the session which uses $st_C.k_{mk}$.)

Next, when $\mathcal{A}$ makes the j-th query to oracle Put1 for an honest user, adversary $\mathcal{B}^3_{\mathsf{aead}}$ queries oracle $\mathrm{Enc}(st_C.k_{mk}, n, k_f, md)$ to encrypt the newly sampled k_f and uses the resulting ciphertext as c_{fk}. It records the file ID of the new file as fid^*. If adversary $\mathcal{A}$ makes subsequent queries to oracle Acpt1 for an honest client on input file ID fid^*, adversary $\mathcal{B}^3_{\mathsf{aead}}$ again queries oracle $\mathrm{Enc}(st_C.k_{mk}, n, k_f, md)$ to encrypt the key from $\mathrm{T}_{\mathrm{oob}}$ and uses the resulting ciphertext as c_{fk}. Thanks to how nonce sampling in putHeader is handled from the hop to G_6, the encryption queries from $\mathcal{B}^3_{\mathsf{aead}}$ are guaranteed to be nonce-respecting.

Finally, instead of decrypting c_{fk} when simulating oracles which call getFile and getHeader for an honest client on input file ID fid^*, $\mathcal{B}^3_{\mathsf{aead}}$ queries oracle Dec in its own game with key index

$st_C.k_{mk}$ on the inputs provided by $\mathcal{A}$. This way, $\mathcal{B}^3_{\mathsf{aead}}$ simulates game G_8 and G_9 for $\mathcal{A}$ when the hidden bit in the IND\$ game is 1 and 0, respectively. It makes at most $1 + \mathcal{Q}_{\textsc{Acpt1}}(\mathcal{A})$ queries to oracle Enc and at most $\mathcal{Q}_{\textsc{Upd1}}(\mathcal{A}) + \mathcal{Q}_{\textsc{Share1}}(\mathcal{A}) + \mathcal{Q}_{\textsc{Get1}}(\mathcal{A})$ to oracle Dec.

When $\mathcal{A}$ halts and returns b', $\mathcal{B}^3_{\mathsf{aead}}$ returns 1 if adversary $\mathcal{A}$ managed to set win to true (as defined in game G_9), otherwise it returns 0. This gives Equation (26).

File integrity: Bounding game G_9**.** In the final step of the proof, we construct an adversary $\mathcal{B}^4_{\mathsf{aead}}$ such that

$$\Pr[\,\mathrm{G}_9\,] \leq \mathbf{Adv}^{\text{INT-CTXT}}_{\mathsf{AEAD}}(\mathcal{B}^4_{\mathsf{aead}}). \tag{27}$$

Adversary $\mathcal{B}^4_{\mathsf{aead}}$ acts as the challenger in game G_9, with some exceptions, which only apply if fid^* first appears through a query to Put1 for an honest client (i.e., $\mathrm{T_u}[i_c] \in \mathrm{S}_{\mathsf{hon}} \setminus \mathrm{S}_{\mathsf{comp}}$). The first exception is that when fid^* is first uploaded through oracle Put1, adversary $\mathcal{B}^4_{\mathsf{aead}}$ does not sample the file key used to encrypt file fid^* itself, but instead queries oracle New to set up a new key in the integrity game. Then, adversary $\mathcal{B}^4_{\mathsf{aead}}$ uses oracle Enc to encrypt the file under the new key. This encryption step is repeated (for the new file content) in case of subsequent updates to fid^*. Second, when adversary $\mathcal{A}$ performs the second step of protocol get for fid^*, adversary $\mathcal{B}^4_{\mathsf{aead}}$ forwards the inputs provided by $\mathcal{A}$ to oracle Dec to decrypt the file.

If at some point adversary $\mathcal{A}$ shares fid^* with a recipient $i_r \in \mathrm{S}_{\mathsf{comp}}$, and subsequently attempts to accept the share, then adversary $\mathcal{B}^4_{\mathsf{aead}}$ halts and aborts the simulation. Note that at this point, adversary $\mathcal{A}$ can no longer set the win flag to true in game G_9, since fid^* is owned by a compromised user, per line 23 of Figure 6.

With this strategy, $\mathcal{B}^4_{\mathsf{aead}}$ simulates G_9 (until $\mathcal{A}$ performs an operation which prevents win* from being set to true). Furthermore, if $\mathcal{A}$ sets win* to true, then by definition it must have provided inputs to protocol $\mathsf{CSS}_{\mathsf{get}}$ such that the decryption of fid^* is successful and does not result in any of the plaintexts provided as input to the encryption oracle by $\mathcal{B}^4_{\mathsf{aead}}$ in simulations of put and update operations. By correctness of the AEAD scheme, the file content being new means that the combination of nonce, file ciphertext or associated data input to decryption must be new. Hence $\mathcal{B}^4_{\mathsf{aead}}$ has performed a successful forgery in the INT-CTXT game, proving Equation (27).

Adversary $\mathcal{B}^4_{\mathsf{aead}}$ has query counts $\mathcal{Q}_{\textsc{New}}(\mathcal{B}^4_{\mathsf{aead}}) \leq 1$, $\mathcal{Q}_{\textsc{Enc}}(\mathcal{B}^4_{\mathsf{aead}}) \leq 1+\mathcal{Q}_{\textsc{Upd1}}(\mathcal{A})$ and $\mathcal{Q}_{\textsc{Dec}}(\mathcal{B}^4_{\mathsf{aead}}) \leq \mathcal{Q}_{\textsc{Get1}}(\mathcal{A})$.

Conclusion. Combining Equations (19) through (27) yields the bound from Theorem 6.2 with the following query counts in summary:

$$\mathcal{Q}_{\textsc{Test}}(\mathcal{B}_{\mathrm{pg}}) \leq \mathcal{Q}_{\mathrm{RO}_1}(\mathcal{A}) + \mathcal{Q}_{\mathrm{RO}_2}(\mathcal{A}) \,,$$

$$\mathcal{Q}_{\textsc{New}}(\mathcal{B}_{\mathsf{prf}}) \leq \mathcal{Q}_{\textsc{Fn}}(\mathcal{B}_{\mathsf{prf}}) \leq \mathcal{Q}_{\textsc{AReg1}}(\mathcal{A}) + \mathcal{Q}_{\textsc{Auth1}}(\mathcal{A}) \,,$$

$$\begin{cases} \mathcal{Q}_{\textsc{New}}(\mathcal{B}^1_{\mathsf{aead}}) & \leq \mathcal{Q}_{\textsc{AReg1}}(\mathcal{A}) + \mathcal{Q}_{\textsc{Auth1}}(\mathcal{A}) \,, \\ \mathcal{Q}_{\textsc{Enc}}(\mathcal{B}^1_{\mathsf{aead}}) & \leq \mathcal{Q}_{\textsc{AReg1}}(\mathcal{A}) \,, \\ \mathcal{Q}_{\textsc{Dec}}(\mathcal{B}^1_{\mathsf{aead}}) & \leq \mathcal{Q}_{\textsc{Auth1}}(\mathcal{A}) \,, \end{cases}$$

$$\begin{cases} \mathcal{Q}_{\textsc{New}}(\mathcal{B}^2_{\mathsf{aead}}) & \leq \mathcal{Q}_{\textsc{AReg1}}(\mathcal{A}) \,, \\ \mathcal{Q}_{\textsc{Enc}}(\mathcal{B}^2_{\mathsf{aead}}) & \leq \mathcal{Q}_{\textsc{Put1}}(\mathcal{A}) + \mathcal{Q}_{\textsc{Acpt1}}(\mathcal{A}) \,, \\ \mathcal{Q}_{\textsc{Dec}}(\mathcal{B}^2_{\mathsf{aead}}) & \leq \mathcal{Q}_{\textsc{Upd1}}(\mathcal{A}) + \mathcal{Q}_{\textsc{Share1}}(\mathcal{A}) + \mathcal{Q}_{\textsc{Get1}}(\mathcal{A}) \,, \end{cases}$$

$$\begin{cases} \mathcal{Q}_{\textsc{New}}(\mathcal{B}^3_{\mathsf{aead}}) & \leq \mathcal{Q}_{\textsc{AReg1}}(\mathcal{A}) \ , \\ \mathcal{Q}_{\textsc{Enc}}(\mathcal{B}^3_{\mathsf{aead}}) & \leq 1 + \mathcal{Q}_{\textsc{Acpt1}}(\mathcal{A}) \ , \\ \mathcal{Q}_{\textsc{Dec}}(\mathcal{B}^3_{\mathsf{aead}}) & \leq \mathcal{Q}_{\textsc{Upd1}}(\mathcal{A}) + \mathcal{Q}_{\textsc{Share1}}(\mathcal{A}) + \mathcal{Q}_{\textsc{Get1}}(\mathcal{A}) \ , \end{cases}$$

$$\begin{cases} \mathcal{Q}_{\textsc{New}}(\mathcal{B}^4_{\mathsf{aead}}) & \leq 1 \ , \\ \mathcal{Q}_{\textsc{Enc}}(\mathcal{B}^4_{\mathsf{aead}}) & \leq 1 + \mathcal{Q}_{\textsc{Upd1}}(\mathcal{A}) \ , \text{ and} \\ \mathcal{Q}_{\textsc{Dec}}(\mathcal{B}^4_{\mathsf{aead}}) & \leq \mathcal{Q}_{\textsc{Get1}}(\mathcal{A}) \ . \end{cases}$$