Toward Shared Ownership in the Cloud

Hubert Ritzdorf, Claudio Soriente, *Member, IEEE*, Ghassan O. Karame, *Member, IEEE*, Srdjan Marinovic, Damian Gruber, and Srdjan Capkun, *Member, IEEE*

***Abstract*— Cloud storage platforms promise a convenient way for users to share files and engage in collaborations, yet they require all files to have a single owner who unilaterally makes access control decisions. Existing clouds are, thus, agnostic to the notion of shared ownership. This can be a significant limitation in much collaboration because, for example, one owner can delete files and revoke access without consulting the other collaborators. In this paper, we first formally define a notion of *shared* ownership within a file access control model. We then propose two possible instantiations of our proposed shared ownership model. Our first solution, called Commune, relies on secure file dispersal and collusion-resistant secret sharing to ensure that all access grants in the cloud require the support of an agreed threshold of owners. As such, Commune can be used in existing clouds without modifications to the platforms. Our second solution, dubbed Comrade, leverages the blockchain technology in order to reach consensus on access control decision. Unlike Commune, Comrade requires that the cloud is able to translate access control decisions that reach consensus in the blockchain into storage access control rules, thus requiring minor modifications to existing clouds. We analyze the security of our proposals and compare/evaluate their performance through implementations using Amazon S3.**

***Index Terms*— Cloud security, shared ownership, distributed enforcement, blockchain technology.**

I. Introduction

EVEN though the cloud promises a convenient way for users to share files and effortlessly engage in collaborations, it still retains the notion of *individual* file ownership. That is, each file stored in the cloud is owned by a single user, who can *unilaterally* decide whether to grant or deny any access request to that file. However, the individual ownership is not suitable for numerous cloud-based applications and collaborations. Consider a scenario where a number of research organizations and industrial partners want to set up a shared cloud repository to collaborate on a joint research project. If all participants contribute their research efforts to the project, then they may want to share the ownership over the collaboration files so that all access decisions are agreed upon among the owners. There are two main arguments why this may be preferred to individual ownership. First, a sole owner can abuse his rights by unilaterally making access control decisions. The community features a number of anecdotes where users revoke access to shared files from other collaborators. Second, even if owners are willing to elect and trust one of them to make access control decisions, the elected owner may not want to be held accountable for collecting and correctly evaluating other owners' policies. For example, incorrect evaluations may incur negative reputation or financial penalties.

In contrast to individual ownership, we introduce a novel notion of *shared ownership* where n users jointly own a file and each file access request must be granted by a pre-arranged threshold of t owners. We remark that existing cloud platforms, such as Amazon S3 or Dropbox, provide no support for shared ownership policies, and offer only basic access control lists. In short, they are *agnostic* to the concept of shared ownership. Furthermore, state-of-the-art trust management systems that can support shared ownership policies (e.g., SecPAL [1], KeyNote [2], Delegation Logic [3]) make all access decisions using a *centralized* Policy Decision Point (PDP). This is not suitable for enforcing our shared ownership model, because the user who administrates the PDP can arbitrarily change the policy rules set by the owners and enforce his own policies.

In this paper, we address the problem of *distributed enforcement of shared ownership within cloud storage providers.* By distributed enforcement, we mean enforcement where access to files in a shared repository is granted if and only if t out of n owners separately support the grant decision. Therefore, we introduce the Shared-Ownership file access control Model (SOM) to define our notion of shared ownership, and to formally state the given enforcement problem. We then propose two instantiations of the SOM model to enforce shared ownership policies in a distributed fashion.

This paper extends our previous work [4]. More specifically, we provide additional formal details about the SOM model. We also propose a new instantiation of the SOM model, Comrade, that leverages functionality from the blockchain in order to reach consensus on access control decisions. Unlike the Commune framework proposed in [4], Comrade requires cooperation from the cloud provider that is expected to translate access control decisions that reached consensus in the blockchain into storage access control rules. Comrade, however, exhibits considerably better performance than Commune. We deploy a smart contract instantiating

Manuscript received October 26, 2017; revised February 19, 2018 and April 10, 2018; accepted May 3, 2018. Date of publication May 17, 2018; date of current version June 5, 2018. This work was supported in part by the TREDISEC Project under Grant 644412 and in part by the European Union under the Information and Communication Technologies theme of the Horizon 2020 Research and Innovation Program. The associate editor coordinating the review of this manuscript and approving it for publication was Prof. Jiankun Hu. *(Corresponding author: Ghassan O. Karame.)*

H. Ritzdorf is with ChainSecurity, 8044 Zürich, Switzerland (e-mail: hubert@chainsecurity.com).

C. Soriente is with NEC Laboratories Europe, 28108 Madrid, Spain (e-mail: claudio.sorienten@emea.nec.com).

G. O. Karame and D. Gruber are with NEC Laboratories Europe, 69115 Heidelberg, Germany (e-mail: ghassan@karame.org).

S. Marinovic is with The Wireless Registry Inc., San Francisco, CA 94107 USA (e-mail: srdjan@wirelessregistry.com).

S. Capkun is with ETH Zürich, 8092 Zürich, Switzerland (e-mail: srdjan.capkun@inf.ethz.ch).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TIFS.2018.2837648

Comrade within the Ethereum blockchain, connect it to Amazon cloud storage [5], and compare its performance to the one of Commune [4] with respect to the file size and the number of users. We summarize our contributions as follows:

- We formalize the notion of shared ownership within a file access control model named SOM, and use it to define a novel access control problem of distributed enforcement of shared ownership in existing clouds.
- We propose a first solution, called Commune, which distributively enforces SOM and can be deployed in an agnostic cloud platform. Commune ensures that *(i)* a user cannot read a file from a shared repository unless that user is granted read access by at least t of the owners, and *(ii)* a user cannot write a file to a shared repository unless that user is granted write access by at least t of the owners.
- We propose a second solution, dubbed Comrade, which leverages functionality from the blockchain technology in order to reach consensus on access control decision. Comrade improves the performance of Commune, but requires that the cloud is able to translate access control decisions that reached consensus in the blockchain into storage access control rules, thus requiring minor modifications of existing clouds.
- We build prototypes of Commune and Comrade and evaluate their performance within Amazon S3 with respect to the file size and the number of users.

The remainder of the paper is organized as follows. Section II introduces our notion of shared ownership in a file access control model. Section III details Commune and analyzes its security. In Section IV, we introduce Comrade and analyze its provisions. Section V evaluates the performance of Commune and Comrade through an implementation within Amazon S3. In Section VI, we discuss further insights with respect to Commune and Comrade. Section VII reviews related work, and we conclude in Section VIII.

II. SOM: Shared-Ownership File Access Control Model

In this section, we define the concept of shared ownership, and formally instantiate it in a file access control model dubbed SOM. Our main motivation for constructing this model is three-fold: *(i)* to precisely define the *ideal* set of features that we believe a model, which enforces shared ownership, should provide; *(ii)* to formulate the problem of distributed enforcement more precisely by focusing on SOM's formal description; and *(iii)* to provide a point of reference to scrutinize SOM's enforcement solutions, including our own.

A. The Notion of Shared Ownership

In a file system, we see the notion of shared ownership as follows. Each file can have one or more owners, and they collaboratively make an access decision.

To make this notion more precise, let an owner credential denote a pair (O, R), where R is a tuple $(Subject, File, Action)$, and O is one of $File$'s owners. Intuitively, an owner credential represents a (unilateral) decision by an owner O to grant a request R.

We then define a *T-out-of-N* file access control policy, also called a *threshold* policy, as follows:

Definition 1 (Threshold Policy): A T-out-of-N *(threshold) access control policy for a file $File$ is a tuple $(T, Owners, File)$ where T is a number representing a threshold, Owners are the $File$'s owners.*

We define an enforcement function $g : Reqs \times TPolicies \times \mathcal{P}(Creds) \mapsto \{grant, deny\}$, where $Reqs$ is a set of requests, $TPolicies$ is a set of threshold policies, and $Creds$ is a set of all possible credentials. Now we define shared ownership enforcement as follows:

Definition 2 (Shared Ownership Enforcement): An enforcement function g enforces shared ownership over $File$ and its threshold policy TPolicy when $g(R, TPolicy, Creds)$ maps to grant iff there are at least T many distinct credentials $(O_1, R), \ldots, (O_T, R)$ in $Creds$, where each O_i is in $Owners$ and no two O_i refer to the same owner.

Intuitively, we say that a file access control model enforces shared ownership if it implements a function g that correctly enforces shared ownership.

B. SOM's Overview

Given the general notion of shared ownership enforcement from Definition 2, in the following we present a file access control model that adopts this concept in the context of a file access control model. It also further defines how ownership can be delegated and revoked, and how files' thresholds can be changed.

Our model, dubbed SOM, takes files as the only protected resources. We do not focus on directories (or other file groupings). Each file is created by one user with the following request:

$$\text{U } \textbf{reqs} \text{ Create(F)}$$

Upon receiving this request, SOM tells a file system to create a file F, assign the user U as the sole owner, and initiate the file's threshold to 1. SOM grants requests for file creation from authenticated users as long as the new file name is unique. To this end, we assume that the file system authenticates U before processing his requests.

SOM allows the ownership over a file to be further shared with, and also revoked from, a user U through the following operations:

- Delegate(F, U) – Delegate ownership of the file F to the user U, i.e., make U one of the owners of F.
- Revoke(F, U) – Revoke ownership of the file F from U, i.e., remove U as an owner of F.

If an owner O wishes to delegate or revoke ownership from U over F, then he issues a credential of the form:

$$O \textbf{ says } \text{Action(F, U)},$$

where Action is either *Delegate* or *Revoke*. Intuitively, one can think of a credential as a certificate by an owner to support an action.

To decide whether a request for an ownership distribution or revocation is in fact enforced for U, SOM consults the file's threshold t to determine how many different credentials U

$$
\begin{aligned}
\textit{credential} &::= \mathit{user}\ \textbf{says}\ (\textit{accessAct} \mid \textit{ownsOp} \mid \mathit{newTOp})\\
\textit{request} &::= \mathit{user}\ \textbf{reqs}\ (\mathrm{Create}(f) \mid \textit{accessOp} \mid \textit{ownsOp} \mid \mathit{newTOp})\\
\textit{accessAct} &::= \mathit{user}\ \textbf{can}\ \mathit{accessOp}\\
\textit{accessOp} &::= \mathrm{Read}(f) \mid \mathrm{Write}(f) \mid \mathrm{Delete}(f)\\
\textit{ownsOp} &::= \mathrm{Delegate}(f,u) \mid \mathrm{Revoke}(f,u)\\
\textit{newTOp} &::= \mathrm{New}_{\mathrm{T}}(f,t,t)\\
u &::= \mathrm{String}\\
t &::= \mathbb{N}\\
f &::= \mathrm{String}
\end{aligned}
$$

Fig. 1. SOM's credential and request grammar. Words in *italics* are non-terminating symbols.

needs from the file's owners. For example, to gain ownership of a file F with $t = 2$, U submits her request:

$$\mathrm{U}\ \textbf{reqs}\ \mathrm{Delegate}(\mathrm{F}, \mathrm{U})$$

which is granted if two distinct owners of F, for example O and O', issue the following credentials:

$$O\ \textbf{says}\ \mathrm{Delegate}(\mathrm{F}, \mathrm{U})$$
$$O'\ \textbf{says}\ \mathrm{Delegate}(\mathrm{F}, \mathrm{U})$$

The full credential and request grammar are defined in Figure 1. To access a file, a user submits the following requests: *(i)* Read(F) – Obtain F's content; *(ii)* Write(F) – Change F's content; *(iii)* Delete(F) – Delete F. For example, to read F, U submits:

$$\mathrm{U}\ \textbf{reqs}\ \mathrm{Read}(\mathrm{F})$$

Similarly to granting and revoking ownership, file access requests are granted if t out of N owners issue the corresponding credentials. For example, if the threshold for F is still 2, then U can *read* F, if the following credentials are present:

$$O\ \textbf{says}\ \mathrm{U}\ \textbf{can}\ \mathrm{Read}(\mathrm{F})$$
$$O'\ \textbf{says}\ \mathrm{U}\ \textbf{can}\ \mathrm{Read}(\mathrm{F})$$

where O and O' are F's owners. We note that in SOM, each of a file's owners can, by default, read that file. However, writing and deleting are still subject to a threshold even for an owner. We find this to be a natural interpretation of shared ownership when compared to unilateral ownership, where an owner has full rights.

Note that successful additions and revocations of the ownership effectively change the number of owners. This, however, does not change the file's threshold. Namely, since adding new owners does not change the threshold t, then the original fraction of owners required to approve file actions is lower. To enable the owners to restore the ratio, or indeed set a new one, the new$_{\mathrm{T}}$ action can be used as follows:

$$O\ \textbf{says}\ \mathrm{New}_T(\mathrm{F}, t_{old}, t_{new})$$

C. Formal Account

Intuitively, we formalize SOM's semantics as follows. We represent a file system state consisting of files, owners and thresholds as a Datalog database [6]. This database consists of a set of relations describing each file's owners and its threshold, and a set of clauses that axiomatize the definitions of shared ownership. We translate a request and credentials into Datalog clauses, which are evaluated over the current state and threshold axioms. For example, file access is granted if a set of credentials supports the grant (expressed as a Datalog query) evaluated over the current state. Facts are added or removed when a set of credentials supports a change of ownership or a change of a particular threshold.

Since SOM's semantics heavily depend on Datalog, we first give a brief overview of Datalog and refer the reader to the more extensive surveys [6]. A Datalog program is a finite set of clauses of the form:

$$S \leftarrow L_1, L_2, \ldots, L_m$$

where S and L_i are function-free first-order literals of the form $predicate(\mathrm{arg}_1, \ldots, \mathrm{arg}_n)$. We refer to S as the head of the clause, and to L_i as a body literal. We adopt the following notation: a variable starts with the ? character, a constant starts with a capital letter, and a predicate name starts with a lowercase letter.

A clause with no body literals is called a *fact*. All clauses are safe: all variables that appear in a head literal also appear in at least one body literal. A Datalog program can be split into two sets of clauses: *EDB* and *IDB*. *EDB* is a set of facts whose head literals do not appear as head literals in any other clause. All other clauses are in the *IDB* set. Intuitively, we think of an *EDB* as an input for computing all implied facts by the clauses in the *IDB* set. The declarative semantics of a Datalog program are given by interpreting each clause as a first-order sentence: $\forall \bar{x} L_1 \wedge \cdots \wedge L_i \rightarrow S$, and then taking a program to be a conjunction of all its clauses. For each program $\mathcal{P} = IDB \cup EDB$ let $\sigma(IDB \cup EDB) = \{fact \mid \mathcal{I}(\mathcal{P}) \models fact\}$, where $\mathcal{I}(\mathcal{P})$ represents the first-order translation of $\mathcal{P}$, and $\models$ is the logical implication.

We formally define SOM's semantics of request evaluations in terms of a labeled transition system (LTS) $(S, L, \rightarrow)$.

A state $s \in S$ is a tuple $(Files, Users, Owns, Thresholds)$ where *Files* denotes a set of strings representing file names, $Users$ is a set of users, *Owns* is a subset of $2^{Users\times Files}$. The *Thresholds* set is a subset of $2^{Files\times\mathbb{N}}$. For the sake of brevity and presentation, we write $Files_s$, to denote the *Files* set of the state s (and similarly for other sets of s as well). We can represent a state s as an (*EDB*) Datalog program s_{EDB} consisting of only the following facts:

$$
\begin{aligned}
\mathit{file}(F).&\ \text{ only if } \{\mathrm{F}\} \subseteq Files_s\\
user(U).&\ \text{ only if } \{\mathrm{U}\} \subseteq Users_s\\
owns(U, F).&\ \text{ only if } \{(\mathrm{U}, \mathrm{F})\} \subseteq Owns_s\\
threshold(F, N).&\ \text{ only if } \{(\mathrm{F}, N)\} \subseteq Thresholds_s
\end{aligned}
$$

For the sake of simplicity, we assume a fixed set of *Users* across all states, and we take s_0 to be $(\{\}, Users, \{\}, \{\})$.

A label $e \in L$ is a tuple $(R, \mathcal{C})$, where R is a request credential submitted by a user, and $\mathcal{C}$ is a set of available credentials. Credentials can be either submitted by a user, or kept in a separate storage and simply appended to each request.

$$\frac{\begin{array}{c} e=(\text{U } \textbf{reqs } \mathit{accessOp}(\text{F}),\mathcal{C}) \\ \sigma(\mathcal{T}(\mathcal{C})\cup\mathcal{A}[s]\cup s_{EDB})\models\mathcal{T}(R) \\ (\text{F}\notin \mathit{Files}_{s'}\text{ , if } \mathit{accessOp}=\text{Delete}) \end{array}}{s\xrightarrow{e}s'}\text{[FAction]} \qquad \frac{\begin{array}{c} e=(\text{U } \textbf{reqs } \text{New}_{\text{T}}(\text{F},t',t),\mathcal{C}) \\ \sigma(\mathcal{T}(\mathcal{C})\cup\mathcal{A}[s]\cup s_{EDB})\models\mathcal{T}(R) \\ (\text{F},t)\in \mathit{Thresholds}_s,\, (\text{F},t')\in \mathit{Thresholds}_{s'},\, (\text{F},t)\notin \mathit{Thresholds}_{s'} \end{array}}{s\xrightarrow{e}s'}\text{[NewT]}$$

$$\frac{\begin{array}{c} e=(\text{U } \textbf{reqs } \text{Delegate}(\text{F},\text{U}'),\mathcal{C}) \\ \sigma(\mathcal{T}(\mathcal{C})\cup\mathcal{A}[s]\cup s_{EDB})\models\mathcal{T}(R) \\ \mathit{Owns}_{s'}=\mathit{Owns}_s\cup\{(\text{U}',\text{F})\} \end{array}}{s\xrightarrow{e}s'}\text{[Delegate]} \quad \frac{\begin{array}{c} e=(\text{U } \textbf{reqs } \text{Revoke}(\text{F},\text{U}'),\mathcal{C}) \\ \sigma(\mathcal{T}(\mathcal{C})\cup\mathcal{A}[s]\cup s_{EDB})\models\mathcal{T}(R) \\ \mathit{Owns}_{s'}=\mathit{Owns}_s\setminus\{(\text{U}',\text{F})\} \end{array}}{s\xrightarrow{e}s'}\text{[Revoke]} \quad \frac{\begin{array}{c} e=(\text{U } \textbf{reqs } \text{Create}(\text{F},\text{U}),\mathcal{C}) \\ \text{F}\notin \mathit{Files}_s,\ \text{F}\in \mathit{Files}_{s'},\ \text{U}\in \mathit{Users}_s \\ \mathit{Owns}_{s'}=\mathit{Owns}_s\cup\{(\text{U},\text{F})\} \\ \mathit{Thresholds}_{s'}=\mathit{Thresholds}_s\cup\{(\text{F},1)\} \end{array}}{s\xrightarrow{e}s'}\text{[Create]}$$

where $\mathit{accessOp}=\{\text{Read,Write,Delete}\}$.

Fig. 2. Transition rules for the → set of SOM's LTS.

The → set contains all (*valid*) transitions, defined as a triple (s, e, s'). All the necessary and sufficient conditions for valid transitions are given in Figure 2. We note that s and s' are equal in all aspects except if otherwise indicated.

Intuitively, all transition rules (except Create) require that $\sigma(\mathcal{T}(\mathcal{C}) \cup \mathcal{A}[s] \cup s_{EDB}) \models \mathcal{T}(R)$, where s is the state in which the request is received. $\mathcal{T}(C)$ are Datalog clauses generated from e:

$$\begin{aligned} \mathcal{T}(\text{U } \textbf{says } \text{U}' \textbf{ can } \mathit{accessOp}(\text{F})) &= \mathit{says}(U, U', \mathit{accessOp}, F) \\ \mathcal{T}(\text{U } \textbf{says } \text{U}' \mathit{ ownsOp}(\text{F}, \text{U}')) &= \mathit{says}(U, U', \mathit{ownsOp}, F) \\ \mathcal{T}(\text{U } \textbf{says } \mathit{New}_T(\text{F}, t, t')) &= \mathit{says}(U, \mathit{New}_T, F, T, T') \end{aligned}$$

In the given translation, $\mathit{accessOp}$ ranges over Write, Read, and Delete; ownsOp ranges over Delegate, and Revoke. The translation of R follows the same idea, except that we do not generate says facts but rather queries that should follow from the submitted speech acts.

$$\begin{aligned} \mathcal{T}(\text{U } \textbf{reqs } \mathit{accessOp}(\text{F}) &= \mathit{can}(U, \mathit{accessOp}, F) \\ \mathcal{T}(\text{U } \textbf{reqs } \mathit{ownsOp}(\text{F}, \text{U}')) &= \mathit{ownsOp}(U', F) \\ \mathcal{T}(\text{U } \textbf{reqs } \text{New}_{\text{T}}(\text{F}, t, t')) &= \mathit{changeT}(F, T, T') \\ \mathcal{T}(\text{U } \textbf{reqs } \text{Create(F)}) &= \mathit{create}(U, F) \end{aligned}$$

The set $\mathcal{A}[s]$ is a *parameterized* (on s) *IDB* program containing necessary clauses to enforce a *T-out-of-N* access control policy.

The first axiom allows owners to read their files:

$$\mathit{can}(?U, \mathit{Read}, ?F) \leftarrow \mathit{file}(?F), \mathit{owns}(?U, ?F)$$

The second axiom is a template for the *accessOp* operations Read, Write, and Delete:

$$\begin{aligned} &\mathit{can}\ (?U, \mathit{accessOp}, ?F) \leftarrow \mathit{file}(?F), \mathit{user}(?U), \\ &\quad \mathit{threshold}(?F, ?T), \\ &\quad [\![\mathit{says}(?U_1, ?U, \mathit{accessOp}, ?F), \ldots, \mathit{says}(?U_{?T}, ?U, \\ &\quad \mathit{accessOp}, F), \mathit{owns}(?U_1, ?F), \ldots, \mathit{owns}(?U_T, ?F), \\ &\quad ?U_1 \neq ?U_2, \ldots, ?U_1 \neq ?U_{?T}, \ldots, ?U_{?T-1} \neq ?U_{?T}]\!] \end{aligned}$$

Intuitively, this *template* axiom generates the necessary clauses (by substituting *accessOp* with Read, Write, and Delete). The generated clauses are further grounded on $?F$ and $?T$, i.e., on all files and their thresholds. The reason for doing so is to correctly enforce the current (for the given state s) threshold T for a particular file. In other words, we need to generate the correct number of $?U_i$ variables for each file and its threshold in s. To represent this *dynamic* part of a clause (that is dynamically adjusted for each state), we enclose it within **[[** and **]]** brackets. We note that the number of variables that need to be generate is given by the $?T$'s value.

The same reasoning applies for the *ownsOp* axioms. We replace *(o|O)wnsOp* with (d|D)elegate and (r|R)evoke, in addition to grounding the clauses on $?F$ and $?T$.

$$\begin{aligned} &\mathit{ownsOp}(?U, ?F) \leftarrow \mathit{file}(?F), \mathit{user}(?U), \\ &\quad \mathit{threshold}(?F, \mathit{OwnsOp}, ?T), \\ &\quad [\![\mathit{says}(?U_1, ?U, \mathit{ownsOp}, ?F), \ldots, \mathit{says}(?U_{?T}, ?U, \\ &\quad \mathit{OwnsOp}, ?F), \mathit{owns}(?U_1, ?F), \ldots, \mathit{owns}(?U_T, ?F), \\ &\quad ?U_1 \neq ?U_2, \ldots, ?U_1 \neq ?U_{?T}, \ldots, ?U_{?T-1} \neq ?U_{?T}]\!] \end{aligned}$$

In case of New_{T}, we ground the clause on $?F$ and $?T$, but omit it here due to space constraints.

Given these axioms and the transition rules, it follows that SOM represents a correct implementation of an enforcement function g given in Definition 2 for all requests, except when a subject is a file's owner as well and the action is a read action. In this case, an owner is always given access. Clearly, we can easily remove this provision from $\mathcal{A}[s]$, but we argue that it is a natural provision to have in a file access control model.

D. Centralized vs. Distributed Enforcement

Given SOM's description, the natural question to consider is how to enforce such a model in a third-party cloud file system that does not endorse shared ownership.

Current state-of-the-art distributed authorization logics—such as SecPAL [1], DKAL [7], Binder [8], KeyNote [2]—that could in principle express SOM's axioms, enforce a policy through a policy decision point (PDP), which evaluates a given set of policies. However, a PDP always has one trusted administrator who has full control over the PDP's policies. This administrator can clearly abuse his powers and modify policies within his PDP and circumvent threshold policies, which defeats the core idea of shared ownership.

We frame this concern as the SOM enforcement problem.

Problem: How can SOM *be enforced without granting one owner unilateral powers?*

III. Commune: Distributed Enforcement of Shared Ownership in an Agnostic Cloud

This section presents Commune, our solution for distributed enforcement of the SOM access control policy in an *agnostic* cloud. As SOM does not specify concrete file access operations, we instantiate Commune with write and read actions. Before introducing our solution, we outline our cloud and attacker model.

A. Cloud and Attacker Model

We focus on a cloud storage platform, $\mathcal{S}$, where a set of users $\mathcal{U}$ have personal accounts onto which they upload files. For example, users might set up their own personal clouds [9], [10], or might create personal accounts in existing public clouds. We assume that $\mathcal{S}$ authenticates users before they get access to the platform. A user $U \in \mathcal{U}$ can unilaterally decide who has access to files stored on his account. In particular, $\mathcal{S}$ allows each user to define access control policies of the type $p : \mathcal{U} \times \{\text{write}, \text{read}\} \rightarrow \{grant, deny\}$. We also assume that $\mathcal{S}$ correctly enforces individual access control policies. This model reflects the functionalities provided by existing cloud platforms, such as Amazon S3.

We focus on both external and internal adversaries. An adversary may try to gain read access to a file even if fewer than t owners have issued the corresponding credentials. We refer to this adversary as a "malicious reader". Alternatively, an adversary, who has been granted write access by fewer than t owners, may try to publish a file F as if F were authored by a user who had been granted write access by t or more owners. We refer to this adversary as a "malicious writer". We also consider sets of users who collude to escalate their access rights.

B. Overview of Commune

Before describing Commune, we make the following observations:

Observation 1: Commune's files cannot be stored on a single user account.

Following the discussion regarding the centralized enforcement, a single user must not be charged with making unilateral grant and deny decisions. Otherwise, that user may abuse his rights and take unilateral access control decisions. A naïve solution where a file is encrypted (e.g., using a key shared among the owners) and the ciphertext is stored on a single account, allows that account holder to, unilaterally deny read access to the ciphertext. If the ciphertext cannot be read, any mechanism to distribute or recover the encryption key is of no help. We argue, therefore, that Commune cannot use a centralized repository owned by a single user because the repository owner can unilaterally grant or deny access to the files stored therein. Our alternative is to use a "shared repository", which is an abstraction built on top of the owners' personal accounts on $\mathcal{S}$.

Observation 2: Commune cannot support in-place writing.

If Commune were to allow in-place writing, then users who are granted write access could overwrite a file with "garbage". This would equate to granting users the right to unilaterally delete the file, thus nullifying our efforts to prevent such scenarios. A standard alternative to in-place writing is to introduce "copy-on-write" mechanisms whereby a new file is created upon each file write operation. To optimize performance, Commune implements versioning and splits files into *units* (i.e., the unit of granularity of versioning) so that writing a new version of an existing file, only requires updating the units that have changed with respect to the previous version.

Observation 3: Commune cannot prevent users from disseminating a file or a key through an out-of-band channel.

Access control solutions cannot prevent a user from distributing content through an out-of-band channel. A user who rightfully reads a file, can leak it to third parties. Alternatively, a rightful reader can share the key with $\mathcal{S}$ and thereby leak the file. Similarly, a malicious writer can write a file and disseminate it through an out-of-band channel. For example, a user can publish files on his account on $\mathcal{S}$ and make them available for others to read. We cannot prevent such behaviour. Commune, however, must at least allow honest readers, who abide to the protocol specification, to distinguish between the content written by malicious writers and the content written by honest writers.

Given these observations, Commune unfolds as follows. At system setup, users define the set of n owners $\mathcal{O}$ and the threshold t (with $t \leq n$).[1] Commune abstracts the storage space of the owners' accounts on $\mathcal{S}$ as the "shared repository". Each owner grants/denies read and write access on his account to users (including other owners) according to his individual access control policy. The distributed enforcement of the SOM access control policy then follows from the enforcement of the individual access policies set by each owner.

To write a file to the shared repository, the writer encodes the file in *tokens* and distributes the tokens to the owners' accounts. A file is written to the shared repository if and only if the writer successfully distributes the file's tokens onto at least t owners' accounts. That is, a user has write access to the shared repository if and only if he has write access to at least t of the owners' accounts. We refer to such a user as an "authorized writer".

To read a file from the shared repository, the reader must fetch the file's tokens from at least t distinct owners' accounts. Therefore, a user has read access to the file if and only if he has read access to the file's tokens by at least t owners. We refer to such a user as an "authorized reader".

To securely enforce shared ownership policies, Commune is designed to fulfil the following properties.

- **P1:** A malicious writer (i.e., a user who has been granted write access by fewer than t owners), must not be able to publish a file F as if F were authored by an authorized writer.

[1] The selection of owners and the threshold t are outside of our scope. In settings like scientific collaboration, these are agreed upon by the partners.

- **P2:** A malicious reader (i.e., a user who has been granted read access to a file F by fewer than t owners), must not be able to recover the file content. This property must also hold in case of *revocation*. Assume that, at the time τ_1, U has read access to F granted by at least t owners. Also assume that, at the time $\tau_2 > \tau_1$, U has his access rights revoked. This happens if, at the time τ_2, some of the owners decide to revoke read access to U so that U is left with fewer than t read grants. We must ensure that, starting from time τ_2, U cannot recover meaningful bits of F. We remark that, as is common for access control systems, we cannot prevent U from storing a local copy of F at the time t_1 and reading it even after his read right has been revoked.

  Commune must also provide *collusion resistance*. That is, coalitions of users—where no single user is an authorized reader—must not be able to pool their credentials to escalate their read access rights.

Property P1 ensures protection against malicious writers who try to disseminate content despite lacking the required credentials. Property P2 guarantees that malicious readers cannot read content written to the shared repository.

Commune fulfils property P1 by design, through the abstraction of the shared repository and the copy-on-write mechanism (see Section III-E). Property P2 is fulfilled through two cryptographic building blocks: Secure File Dispersal (SFD), and Collusion Resistant Secret Sharing (CRSS). SFD ensures that malicious readers cannot acquire any information about a file, even if they previously had access to the file and were later revoked. CRSS builds atop SFD and ensures that coalitions of users where no single user has enough credentials to read the file, cannot pool their credentials in order to escalate their read access rights.

In the following, we describe and analyze SFD (Section III-C) and CRSS (Section III-D). In Section III-E, we detail the integration of both building blocks in Commune.

C. Secure File Dispersal (SFD)

Information dispersal algorithms [11] encode a file in n chunks so that any t chunks (where $t \leq n$) are sufficient to decode it. However, information dispersal algorithms do not provide any security guarantees if the number of available chunks is smaller than t: any party with fewer than t chunks may still recover meaningful information about the original file's content.

Previous work on securing information dispersal algorithms [12] combines erasure codes with All-Or-Nothing Transforms (AONT) [13]. The latter is an efficient block-wise transformation that maps an n-block bitstring in input to an n'-block bitstring in output (with $n' \geq n$). AONTs are designed in such a way that, unless all the n' output blocks are available, it is hard to recover any of the input blocks.

Existing AONTs [13], [14] leverage block ciphers and rely on the secrecy of a cryptographic key that is embedded within the output blocks. Given all AONT output blocks, the key can be recovered; once the key is known, individual blocks can be reverted, independently of other blocks. Current AONTs, therefore, preserve their all-or-nothing property only for *one time:* knowledge of the cryptographic key allows to revert single output blocks and to recover parts of the original data. This is at odds with our security requirements. As argued before, we cannot prevent users from caching a local copy of the file and reading it at later time when their read rights may have been revoked. However, we still want to provide revocation of a user who only stored the encryption key at the time when he had read access to the file.

We therefore introduce a new scheme, called Secure File Dispersal (SFD), that combines information dispersal algorithms with an AONT that preserves its all-or-nothing property even if the adversary has the encryption key.

Definition: An SFD scheme consists of the following algorithms:

$\{c_1, \ldots, c_n\} \leftarrow \text{SFD.Encode}(t, n, \text{F}, K, \lambda)$.
Encodes a file F into n chunks, such that F can be correctly decoded using any t chunks; K denotes a key used in the encoding process and λ is a security parameter.

$\text{F}' \leftarrow \text{SFD.Decode}(K, \mathcal{C}, \lambda)$.
Takes as input a key K, a set of chunks $\mathcal{C}$, and security parameter λ; it outputs a file F'.

Correctness. Given $\{c_1, \ldots, c_n\} \leftarrow \text{SFD.Encode}(t, n, \text{F}, K, \lambda)$ and $\text{F}' \leftarrow \text{SFD.Decode}(K, \mathcal{C}, \lambda)$, we require that if $\mathcal{C} \subseteq \{c_1, \ldots, c_n\}$ and $|\mathcal{C}| \geq t$, then $\text{F}' = \text{F}$.

Security.

We define the advantage of adversary $\mathcal{A}$ as follows:

$$\begin{aligned}\text{Adv}_{SFD}(\mathcal{A}) = \Pr[&f \leftarrow \mathcal{A}(K, \mathcal{C}) | K \leftarrow \{0,1\}^l, l \geq \lambda,\\ &\text{F} = (f_1, \ldots, f_m) \leftarrow \{0,1\}^{m\lambda},\\ &\{c_1, \ldots, c_n\} \leftarrow \text{SFD.Encode}(t, n, \text{F}, K, \lambda),\\ &\mathcal{C} \subset \{c_1, \ldots, c_n\},\ \ |\mathcal{C}| < t,\ \ f \subseteq \text{F},\ \ |f| \geq \lambda].\end{aligned}$$

where $f \subseteq \text{F}$ refers to a substring of F. We say that SFD is secure if, for any p.p.t. adversary, its advantage is negligible in the security parameter, i.e., $\text{Adv}_{SFD}(\mathcal{A}) \leq negl(\lambda)$. Our security definition captures the scenario where, at an earlier time, $\mathcal{A}$ was given enough chunks to decode F and has cached a copy of the key K, while at current time he is only given fewer than t chunks. Even if $\mathcal{A}$ has the key K, we require the probability that $\mathcal{A}$ recovers any λ consecutive bits of F to be negligible in the security parameter.

Instantiation. Our SFD scheme combines information dispersal techniques with AON-FFT, an all-or-nothing transformation inspired by Fast Fourier Transform.

Let $E : \{0,1\}^{4\lambda} \rightarrow \{0,1\}^{2\lambda}$ be a semantically secure block cipher (e.g., $E(\cdot)$ could correspond to 256-bit Rijndael [15], with $\lambda = 128$).[2] AON-FFT takes as input a symmetric key K (of size 2λ) and m input blocks $(f_1, \ldots, f_m)$ (each of size λ). It executes in $\log_2 m$ rounds and, at each round, applies $E(\cdot)$ to pairs of blocks. Each round is fed with the output of the previous round. The original input $(f_1, \ldots, f_m)$ is treated as the output of round 0; the final output of the algorithm is the output of round $\log_2 m$ (cf. Figure 3). The pseudo-code of

[2] The key size is 2λ and the input/output size is also 2λ, totalling 4λ size of input.

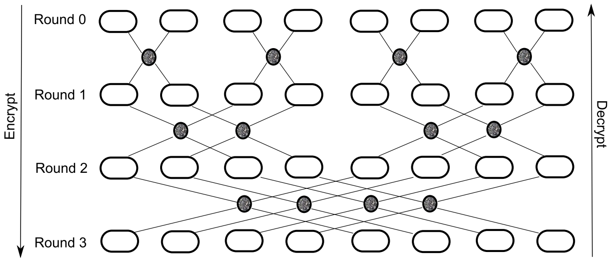


Fig. 3. Sketch of the AON-FFT scheme where the input consists of $m = 8$ input blocks. Solid circles refer to the block cipher $E(\cdot)$, while empty circles depict its input/output blocks.

Algorithm 1 AON-FFT($K, f_1, \dots, f_m$)

```
Parse $f_1, \dots, f_m$ as $f^0_1, \dots, f^0_m$
for $r \leftarrow 1$ to $\log_2 m$ do                         ▷ round counter
    for $i \leftarrow 0$ to $\frac{m}{2^r} - 1$ do
        for $j \leftarrow 1$ to $2^{r-1}$ do
            $f^r_{j+i\cdot 2^r} || f^r_{j+i\cdot 2^r+2^{r-1}} \leftarrow E(K, f^{r-1}_{j+i\cdot 2^r}, f^{r-1}_{j+i\cdot 2^r+2^{r-1}})$
        end for
    end for
end for
return $f^r_1 \dots, f^r_m$ as $\bar{f}_1 \dots, \bar{f}_m$
```

AON-FFT is shown in Algorithm 1. We omit the details of the decryption algorithm since it is specular to encryption.

Given the pseudo-code of AON-FFT, our SFD scheme unfolds as follows:

$\{c_1, \dots, c_n\} \leftarrow$ SFD.Encode$(t, n, \mathrm{F}, K, \lambda)$.
Parse F as $(f_1, \dots, f_m)$ where each f_i has size λ. Run $\bar{f}_1 \dots, \bar{f}_m \leftarrow$ AON-FFT$(K, f_1, \dots, f_m)$. Use the information dispersal encoder to encode $\bar{f}_1 \dots, \bar{f}_m$ in n chunks with reconstruction threshold t.[3]

$\mathrm{F}' \leftarrow$ SFD.Decode$(K, \mathcal{C}, \lambda)$.
Given a set of at least t chunks $\mathcal{C}$ and key K, use the information dispersal decoder to decode blocks $(\bar{f}'_1, \dots, \bar{f}'_m)$. Run $(f'_1 \dots, f'_m) \leftarrow$ AON-FFT$(K, \bar{f}'_1, \dots, \bar{f}'_m)$.

Correctness. If $\{c_1, \dots, c_n\} \leftarrow$ SFD.Encode$(t, n, \mathrm{F}, K, \lambda)$, any subset of at least t chunks $\{c_{i_1}, \dots, c_{i_t}\}$ can be decoded into the whole output of AON-FFT, namely $\bar{f}_1 \dots, \bar{f}_m$. Given K, the output of AON-FFT can be decrypted to recover $\mathrm{F} = (f_1, \dots, f_m)$.

Security. Given the construction of our AON-FFT scheme, it is easy to see that each input block depends on all output blocks and on the encryption key. Furthermore, assuming that $E(\cdot)$ is a semantically secure block cipher, for any p.p.t. algorithm $\mathcal{A}$, we have $\mathrm{Adv}_{SFD}(\mathcal{A}) \leq negl(\lambda)$. More details on the security of AON-FFT can be found in [4].

Note that a construct similar to AON-FFT, was first mentioned by Rivest [13] and later on used as a "proof of storage" in [17]. Nevertheless, the construction proposed therein can use any pseudo-random permutation in the FFT network. Our AON-FFT requires a keyed permutation, hence a block-cipher. Furthermore, the goal of the adversary in [17] is to recover, in a given amount of time, all *output* blocks. In contrast, the goal of our adversary is to recover any *input* block. This entails a different security analysis.

D. Collusion Resistant Secret Sharing (CRSS)

We now introduce our second building block, called Collusion Resistant Secret Sharing (CRSS). Similar to threshold secret-sharing schemes, CRSS allows one party to distribute a secret among a set of designated shareholders, so that any subset of shareholders of size equal to or greater than the threshold can reconstruct the secret. Furthermore, CRSS allows shareholders to issue to other users *delegation* to reconstruct the secret. If a user collects enough (i.e., above the threshold) delegations, he can rightfully reconstruct the secret. However, users cannot pool their delegations to reconstruct the secret, unless one of them has collected enough delegations. In Commune, CRSS is used to secret-share the key K used in SFD, in order to achieve collusion resistance.

CRSS is inspired by decentralized Attribute Based Encryption [18] where shares of a secret are *blinded* with shares of 0, such that, if a user collects enough shares for his identity, the blinding cancels out and the secret can be reconstructed.

Definition: Our definition of CRSS builds on top of a *standard* threshold secret-sharing scheme SS with algorithms SS.Share$(\cdot)$ and SS.Combine$(\cdot)$, to share and reconstruct a secret, respectively. We assume SS to be secure according to the `Game Priv` definition by Rogaway and Bellare [19]. That is, we assume that an adversary has only negligible advantage in identifying which out of two values was (t, n) secret-shared using the SS.Share$(\cdot)$ algorithm, even if the adversary can corrupt up to $t - 1$ shareholders and access their shares.

CRSS defines the following algorithms:

$\{s_1, \dots, s_n\} \leftarrow$ CRSS.Share(s, t, n).
Shares secret s in a set of n shares $\{s_1, \dots, s_n\}$ with reconstruction threshold t.

$d_{i,j} \leftarrow$ CRSS.Delegate(s_i, U_j).
Takes as input a share s_i and an user identity U_j. The output is a *delegation* $d_{i,j}$.

$s' \leftarrow$ CRSS.Combine$(\{d_{i_1,j}, \dots, d_{i_l,j}\})$.
Combines delegations $\{d_{i_1,j}, \dots, d_{i_l,j}\}$ into s'.

Correctness. Given $\{s_1, \dots, s_n\} \leftarrow$ CRSS.Share(s, t, n) and $s' \leftarrow$ CRSS.Combine$(\{d_{i_1,j}, \dots, d_{i_l,j}\})$, we require that if $d_{i_p,j} \leftarrow$ CRSS.Delegate(s_{i_p}, U_j), for $1 \leq p \leq l$ and $l \geq t$, then $s' = s$.

Security.

We model the security of CRSS using an adaptation of the `Game Priv` of [19] and we denote the refined game by `Game Priv`*:

Init. The adversary $\mathcal{A}$ submits two messages x_0, x_1 of equal length. The challenger flips an unbiased coin b and runs $\{s_1, \dots, s_n\} \leftarrow$ CRSS.Share(x_b, t, n).

Find. $\mathcal{A}$ can submit two types of queries. In Type-1 queries, the adversary can corrupt up to $t' \leq t - 1$ shareholders and receives their shares.

[3] SFD can leverage any information dispersal algorithm (e.g., Reed-Solomon codes [16]).

At this time, $\mathcal{A}$ picks t' indexes $i_1, \ldots, i_{t'}$ and receives $\{s_{i_1}, \ldots, s_{i_{t'}}\}$. In Type-2 queries, for any fresh identity U_j, the adversary can ask for up to t'' delegations, as long as $t' + t'' \leq t - 1$. $\mathcal{A}$ submits an identity U_j and t'' indexes $i_1, \ldots, i_{t''}$, and receives delegations $\{d_{i_1,j}, \ldots, d_{i_{t''},j}\}$.

Guess. The adversary outputs his guess b' and wins if $b' = b$.

We define the advantage of the adversary as the probability of its winning minus a half. That is, $\mathrm{Adv}^{\mathtt{Priv}^*}_{CRSS}(\mathcal{A}) = Prob[\mathtt{Priv}^{*\mathcal{A}}] - \frac{1}{2}$. Therefore, we say that CRSS is secure if any p.p.t. algorithm $\mathcal{A}$ has only negligible advantage in winning `Game Priv*`.

The above `Game Priv*` models a scenario where a set of malicious users, including up to t' shareholders, collects up to t'' delegations for each of their identities. If $t' + t'' \geq t$, the malicious shareholders can produce the missing delegations for any of the colluding user identities, so that the secret can be reconstructed by means of CRSS.Combine($\cdot$). Otherwise, colluding users must not be able to retrieve the secret.

Instantiation. Our CRSS scheme is based on the threshold secret-sharing scheme proposed in [20], which is defined as follows:

$g^x, \{x_1, \ldots, x_n\} \leftarrow$ SS.Share$(-, t, n)$.

Pick a cyclic group G of prime order q where the discrete logarithm assumption holds; let $\langle g \rangle = G$. Pick a random $x \in Z_q$ and set the secret to g^x. Pick a random $t-1$-degree polynomial X with coefficients in Z_q, such that $X(0) = x$. Set the i-th share to $x_i = X(i)$.

$s' \leftarrow$ SS.Combine$(\{x_{i_1}, \ldots, x_{i_l}\})$.

Given shares $\{x_{i_1}, \ldots, x_{i_l}\}$, use polynomial interpolation to recover the secret. That is $s' = g^{\sum_{p=1}^{p=l} x_{i_p} \lambda_p}$ where $\lambda_p = \prod_{1 \leq k \leq l}^{k \neq p} \frac{x_{i_k}}{x_{i_k} - x_{i_p}}$.

Note that in the above scheme, the secret is not given as input to the Share algorithm; rather, it is set to g^x for a randomly chosen x. Given the above algorithms, our CRSS scheme unfolds as follows:

$\{s, s_1, \ldots, s_n\} \leftarrow$ CRSS.Share$(-, t, n)$.

Run SS.Share$(-, t, n)$ to obtain $g^x, \{x_1, \ldots, x_n\}$. Pick $H(\cdot) : \{0, 1\}^* \rightarrow G$ to be a cryptographic hash function that maps random strings in G. Pick a random $t-1$-degree polynomial Y with coefficients in Z_q, such that $Y(0) = 0$, and denote $y_i = Y(i)$. The secret is set to $s = g^x$ while each share is set to $s_i = (x_i, y_i)$.

$d_{i,j} \leftarrow$ CRSS.Delegate(s_i, U_j).

Parse $s_i = (x_i, y_i)$ and output $d_{i,j} = g^{x_i} H(U_j)^{y_i}$.

$s' \leftarrow$ CRSS.Combine$(\{d_{i_1,j_1}, \ldots, d_{i_l,j_l}\})$.

Run $s' \leftarrow$ SS.Combine$(\{d_{i_1,j_1}, \ldots, d_{i_l,j_l}\})$.

Correctness. If $l \geq t$, then CRSS.Combine$(\{d_{i_1,j}, \ldots, d_{i_l,j}\})$ outputs

$$
\begin{aligned}
s' &= \prod_{p=1}^{p=l} \left(d_{i_p,j_p}\right)^{\lambda_{i_p}} = \prod_{p=1}^{p=l} \left(g^{x_i} H(U_j)^{y_i}\right)^{\lambda_{i_p}} \\
&= g^{\sum_{p=1}^{p=l} \lambda_{i_p} x_{i_p}} H(U_j)^{\sum_{p=1}^{p=l} \lambda_{i_p} y_{i_p}} \\
&= g^k H(U_j)^0 = g^k = s.
\end{aligned}
$$

Security. The security of CRSS is based on the fact that, in the random oracle model, delegations for different identities cannot be combined to remove the blinding factor from the secret. Assuming that $H(\cdot)$ is modeled as a random oracle and that the discrete logarithm assumption holds in G, we can show that any p.p.t. algorithm $\mathcal{A}$ has only negligible advantage in winning `Game Priv*`. More details on the security of CRSS can be found in [4].

E. Commune: Protocol Specification

Recall that Commune leverages a shared repository, which is an abstraction of the owners' storage space. The shared repository uses a versioning system so that content cannot be overwritten but only new content can be added. In particular, Commune optimizes performance by splitting a file in smaller *units*, and encoding/decoding each unit separately. Therefore, when a new file version is written to the shared repository, the writer only needs to upload the units that have changed from the previous version.

Files written to the repository are encoded in *tokens* and distributed across the owners' accounts. Leveraging the basic ACLs of $\mathcal{S}$, owners define their individual policy on the tokens in their accounts. The distributed enforcement of the SOM policy is implied by the enforcement of each owner's individual policy on his tokens by $\mathcal{S}$. Encoding must guarantee both correctness and security of reading operations. Hence, users who are authorized to read at least t tokens must be able to decode the original file; users who are granted read access on fewer than t tokens must not be able to recover its content. Furthermore, users must not be able to pool their credentials to escalate their access rights.

1) Create a File: File creation requires one user, the file creator, to "bootstrap" the system and write the initial version of the file into the repository. For this reason, we assume that—at the file creation time—the file creator has been granted the right to write new data to each of the owner's accounts on $\mathcal{S}$.

The file creator splits the file F into k fixed-sized units. For each unit F_i, he runs $\{s_i, s_{i1}, \ldots, s_{in}\} \leftarrow$ CRSS.Share$(-, t, n)$ to produce a fresh secret s_i and n of its shares. Secret s_i is used as a symmetric key to encode the unit F_i in n chunks using SFD. That is, the file creator runs $\{c_{i1}, \ldots, c_{in}\} \leftarrow$ SFD.Encode$(t, n, \mathrm{F}_i, s_i, \lambda)$. The token of the unit F_i for the owner O_j is set to (c_{ij}, s_{ij}) (i.e., one chunk outputted by SFD.Encode($\cdot$) and one secret-share outputted by CRSS.Share($\cdot$)). Finally, for each owner O_j, the file creator writes $\{(c_{ij}, s_{ij})\}_{i \in [1,\ldots,k]}$ to O_j's account on $\mathcal{S}$. Each owner, therefore, receives one token for each unit that constitutes F.

2) Grant/Deny Write Rights: An owner O_j grants write rights to a user U_l by granting to U_l the right to write new data (i.e., tokens) to O_j's account. Similarly, O_j denies write rights to U_l by denying U_l the right to write new data to O_j's account.

3) Update a File: Assume U_l wants to write a new version of a file F. For simplicity, assume that the new version differs from the previous one by only one unit F_i (the case where the old and the new versions differ in several units is handled in a similar fashion). At this point, some owners may allow U_l to write tokens to their accounts while others may not. Let $\mathcal{O}^+$ be the subset of owners who grant to U_l write rights to their accounts. Similarly, let $\mathcal{O}^-$ be the subset of owners who deny to U_l write rights to their accounts. U_l can, therefore, only distribute tokens to owners in $\mathcal{O}^+$. This scenario is equivalent to the case where U_l distributes tokens to all owners in $\mathcal{O}$, but the ones in $\mathcal{O}^-$ decide to reject the version produced by U_l and make the received tokens unavailable.

U_l is an authorized writer and his version accepted (i.e., considered as written to the shared repository) if and only if $|\mathcal{O}^+| \geq t$. In this case, there are at least t tokens for the new unit, so it may be decoded by users who collect enough credentials. If $|\mathcal{O}^+| < t$, user U_l is not authorized to write and his version is rejected (i.e., considered as not written to the repository), since there are not enough tokens to decode the unit produced by U_l.

4) Grant/Deny Read Rights: Recall that for each unit F_i, an owner O_j receives the token (c_{ij}, s_{ij}). O_j can grant to U_l read access to that unit by *endorsing* the token for U_l and granting to U_l read access on the endorsed token. Token endorsement requires O_j to run $d_{ij,l} \leftarrow$ CRSS.Delegate(s_{ij}, U_l). The endorsed token $(c_{ij}, d_{ij,l})$ is then made available by O_j for U_l to read. If a file consists of multiple units, O_j must endorse all relative tokens for U_l and grant to U_l read access on all endorsed tokens.

O_j can revoke read rights that were previously granted, by denying to U_l the right to read the previously endorsed tokens.

5) Read a File: If the original file spans several units, U_l must decode each unit separately in order to read the entire file. That is, for each unit, he uses the set of endorsed tokens he can fetch to recover the secret key via CRSS.Combine$(\cdot)$ and then uses the secret key to decode the unit via SFD.Decode$(\cdot)$. Note that for an authorized reader to read version x of file F, he must fetch the latest endorsed tokens created up to (and including) version x, for each unit that comprises the file. Assume user U_l is granted read access to $\{(c_{ij_1}, d_{ij_1,l}), \ldots, (c_{ij_t}, d_{ij_t,l})\}$. To recover F_i that user runs $s_i \leftarrow$ CRSS.Combine$(\{d_{ij_1,l}, \ldots, d_{ij_t,l}\})$ and then $\text{F}_i \leftarrow$ SFD.Decode$(s_i, \{c_{ij_1}, \ldots, c_{ij_t}\}, \lambda)$. U_j proceeds in a similar way to recover all units of F that he has access to.

F. Security Analysis

User authentication rules out attacks from entities that do not have an account on $\mathcal{S}$. That is, if $\mathcal{S}$ correctly enforces user authentication, non-registered users have no means to access the platform or the (encrypted) data. Therefore, in the following we only focus on internal adversaries, i.e., malicious users who have an account at $\mathcal{S}$.

From Sections III-C and III-D, it follows that given t tokens of a file unit F_i (endorsed for a unique user identity), it is possible to recover both the secret key used to encode F_i and its AON-FFT ciphertext, so that the original file can be decrypted. That is, users can read files written by honest writers, if they are granted such right by at least t out of n owners.

Property P1 (cf. Section III-B) is fulfilled as follows. First, Commune uses copy-on-write to prevent writers from overwriting content in the shared repository with garbage. Second, malicious writers (i.e., writers with less than t write permissions) are unable to distribute a file without honest readers detecting it. In other words, a file is considered written if and only if it is correctly encoded in tokens and those tokens are distributed to and endorsed by at least t out of n owners. Any content distributed through other means (e.g., out of band channels) is recognized as malicious by honest readers. We argue that detection of unauthorized files is the only solution for protecting honest readers, because there are no mechanisms to deter malicious writers from disseminating arbitrary content (cf. Observation 3). We also stress that honest readers can easily detect writers that distribute polluted (i.e., non-decodable) tokens. Denial-of-service attacks are, nevertheless, out our scope.

Property P2 is satisfied by combining CRSS and SFD. The former ensures that coalitions of users, where no single user has enough tokens endorsed for his identity, cannot pool their endorsed tokens in order to escalate their access rights. The latter addresses the case where at a time τ_1 a user has access to t or more tokens of a file unit F_i, but at a time $\tau_2 > \tau_1$, his access rights are revoked. That is, at time τ_2, the user has access to fewer than t endorsed tokens. SFD ensures that even if, at time τ_1 the user may have cached the key used to encode F_i, he will not be able to decode parts of F_i at time τ_2. Note that, once a user has access to the file, then he can locally store any plaintext of his choice. Similar to other access control schemes, Commune cannot deter this behavior.

Finally, given the guarantees that Commune makes for write and read actions, it follows that Commune is a (correct) solution for distributed enforcement of the SOM access control policy.

IV. Comrade: Blockchain-Based Shared Ownership

In this section, we present an alternative solution for enforcing shared ownership in the cloud by leveraging functionality from the blockchain. Our solution, dubbed Comrade, enables a distributed blockchain-based enforcement of the SOM access control policy in a cooperative cloud. Unlike Commune, Comrade does not assume an agnostic cloud, and requires the cloud operator to cooperate and to interface with the blockchain. Since SOM does not specify concrete file access operations, we instantiate Comrade with write and read actions. Before introducing our solution, we provide some background on the blockchain and describe the system model.

A. Blockchain and Smart Contracts

The notion of blockchain was originally introduced by the well-known proof-of-work hash-based mechanism that *confirms* cryptocurrency payments in Bitcoin [21]. The PoW-based blockchain ensures that all transactions and their order of execution are available to all blockchain nodes, can be verified by all involved entities and aids the consensus between the parties. Bitcoin's blockchain fueled innovation, and a number of innovative applications have already been devised by exploiting the secure and distributed provisions of the underlying blockchain. Prominent applications include secure timestamping [22], [23], and smart contracts [24].

Smart contracts refer to binding contracts between two or more parties that are executed by all blockchain nodes. Namely, smart contracts implement state machine replication. Smart contracts typically consist of a self-contained code and persistent storage available to all blockchain nodes. For example, Ethereum [24] is a decentralized platform that enables the execution of arbitrary applications (or contracts) on its blockchain. Owing to its support for a Turing-complete language, Ethereum (which currently also relies on PoW-based consensus) offers an easy means for developers to deploy their distributed applications in the form of smart contracts.

To make smart contracts more powerful, techniques have been developed to securely insert real-world facts into blockchains, such as TownCrier [25]. These facts, such as weather information or flight delays, allow contracts to take real-world events into account and to offer new functionalities.

B. Overview of Comrade

The main idea behind Comrade is that a smart contract can instantiate a trusted third party that can evaluate user credentials against owners access policies in a trustworthy manner. This is a basic provision of the blockchain technology that holds as long as the security assumptions on the blockchain hold. (We will argue on those assumption in the security analysis). Hence, in Comrade, a smart contract assists the cloud's PDP ensuring trustworthy handling of policies and credentials. Differently from Commune, however, Comrade needs the cloud to be "shared-ownership aware" and enforce the policies defined by the smart contract. (Recall that we assume the cloud to correctly enforce access policies.)

In more details, cloud accounts in Comrade are not owned by users, but by a smart contract that is running within a blockchain. We refer to such a smart contract by *owner contract* and we rely on it to ensure access control as agreed upon by the file owners. The cloud's PDP makes access control decisions by evaluating a standardized function within the owner contract, as depicted in Figure 4. To grant or deny access rights, the owners submit their votes to the owner contract, which stores them in the blockchain. The PDP's decision then depends on the access control policy, encoded in the owner contract, and data stored inside the blockchain, i.e., owners' votes or securely inserted facts.

To perform an action a on file F in Comrade, user U_l proceeds as follows. U_l issues a standard access request to the cloud storage. The request is authenticated using U_l's private key. The cloud PDP determines the corresponding owner contract for F and evaluates the `hasAccess()` function inside that owner contract: `hasAccess`(F, U_l, a).

`hasAccess()` is evaluated based on the contract's access control policy, the owners' votes and potentially additional blockchain data. The derived access control decision is then enforced by the cloud's Policy Enforcement Point (PEP). Notice that the cloud PDP performs this evaluation by locally executing `hasAccess()` on the current state of the blockchain, i.e., the evaluation triggers no action on the blockchain and requires no fees.

The owner contract also manages the users. Users can join the system by sending a request to the owner contract. For every user, the contract's storage contains the user's public key, used for authentication and data encryption as explained below. The storage also contains every user's accounting balance. Finally, the contract contains procedures for initializing and closing the cloud account.

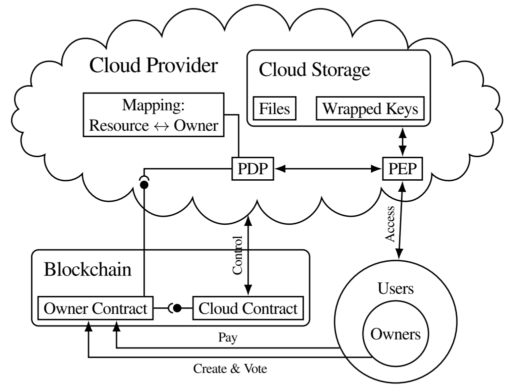


Fig. 4. Overview of Comrade . Access control decisions depend on the evaluation of a smart contract executed within the blockchain.

Recall that the owner contract stores the votes inside its storage. To minimize the overhead associated with such a voting scheme (i.e., storage costs in the blockchain), Comrade employs a hierarchical file structure and groups files into directories. This allows users to issue a directory-specific vote; votes on directories are valid for all contained files and subdirectories (unless a more precise vote exists). We additionally group users into roles by leveraging role-based access control (RBAC) [26]. RBAC allows full flexibility at higher efficiency as owners only need to vote on access rights for the roles.

Similar to Commune, we assume that the cloud provider will enforce access control decisions correctly at all time (although the provider might be interested in learning the contents of files).

Comrade also ensures fair payment by all owners, protect the cloud provider from free-riding, and punishes unfair behaviour. To do so, each user in Comrade makes a policy-defined deposit at the owner contract at system setup. The owner contract tracks each user's balance (e.g., and punishes them for delayed payments). To pay for cloud storage, the owner contract forwards a part of the users' deposits to a deposit inside the cloud contract.

In turn, the cloud contract deducts the operational costs from the deposit and requests the deposit to be refilled before it reaches zero. Once it reaches zero, access to the cloud resources is denied and after some grace period the cloud resources are released.

Similarly, the owner contract requires users to restock their deposit. Otherwise, the owner contract can impose sanctions, e.g., deny certain access rights or ignore votes in case of an owner. Such sanctions and the payment procedures are defined as part of the owner contract which is visible to all owners at contract creation. Notice here that different accounting policies are feasible. For example, the owners can equally split the costs, can ask users to pay a share of the costs or the policy can dictate usage-based cost sharing where more active users pay more.

In contrast to Commune, Comrade requires slight changes to the cloud architecture. Namely, the cloud needs to provide a blockchain interface to manage and pay for used cloud resources. To offer such an interface, a single smart contract per cloud provider is sufficient. We refer to such a contract as a *cloud contract*. The cloud can monitor the state of the cloud contract and perform the requested operations. Such an interface seems realistic as cloud providers currently provide more complex interfaces such as command-line tools or web platforms. The cloud also needs a slight modification in its PDP. Access control requests for cloud resources owned by a smart contract are decided by evaluating a function inside the matching owner contract. We refer to this as a blockchain-aware PDP. Overall, Comrade only requires minor, inexpensive changes in the cloud infrastructure.

We argue that Comrade ensures that the cloud provider cannot be held accountable for collecting and correctly evaluating other owners' policies. For example, incorrect evaluations may incur negative reputation or financial penalties. Instead, all votes are collectively evaluated by the blockchain nodes. Moreover, Comrade allows for the first time the implementation of complex, distributed, event-based access control policies that would considerably enrich the cloud offering.

C. Comrade: Protocol Specification

We now detail the operations of Comrade.

1) Create a File: During file creation, one user—the file creator—writes the initial version of the file into the repository. This requires the file creator to have write permissions for the directory the file is created in.

The file creator also encrypts the file using a randomly chosen *file key* before uploading it. The encrypted file is uploaded as F using a write action. To securely distribute the file key to U_i, the file creator also uploads wrapped keys F_{k,U_i} containing the file key for file F encrypted with the public key of user U_i. By default a file creator uploads wrapped file keys for all owners. Notice that the access control policy for F_{k,U_i} is defined such that a user U_j can access F_{k,U_i} if and only if $U_j = U_i$ and U_j can access F.

2) Grant/Deny Write Rights: An owner O_j grants write rights for a resource F (and the associated wrapped file key) to an entity U_l by submitting a corresponding vote to the blockchain. The vote consists of a blockchain transaction $\mathrm{V}(O_j, U_l, \text{write}, \mathrm{F}, D)$, as shown in Algorithm 2. Here, F can be a file or directory, U_l can be a single user or a specific role and D is grant or deny. Similarly, O_j denies write rights for F to U_l by voting against the access. Notice that access to F also implies access to the associated wrapped file key.

3) Update a File: Assume U_l wants to write a new version of a file F. U_l encrypts F using its file key and issues a write action as described in Section IV-B. In case the owner contract implements a threshold-based access control policy, the request succeeds if there are at least t owner votes in favour.

4) Grant/Deny Read Rights: Analogously to write rights, an owner O_j grants or denies read rights for a resource F to an entity U_l by submitting a corresponding vote to the owner contract. As mentioned earlier, this vote corresponds to a blockchain transaction $\mathrm{V}(O_j, U_l, \text{read}, \mathrm{F})$.

Algorithm 2 Sketch of Comrade Smart Contracts

```
function V(O, U, a, F, D)                                ▷ Vote function
    votes[O, U, a, F] = D            ▷ D = GRANT/DENY, a = Action
end function

function HASACCESS(F, U, a)                              ▷ PDP function
    grant ← 0
    O ← Owners[F]                    ▷ Identifying File-Based Owners
    for i ← 1 to |O| do
        if votes[O_i, U, a, F] == GRANT then
            grant ← grant + 1           ▷ Accumulating Owner Votes
        end if
    end for
    return (grant ≥ threshold[A, F])                      ▷ Policy Check
end function
```

5) Read a File: Assume U_l wants to read a file F. U_l issues a read request for F and F_{k,U_i} as described in Section IV-B. In case the owner contract implements a threshold-based access control policy, the request succeeds if there are at least t owner votes in favour. U_l decrypts F_{k,U_i} using its private key to obtain the file key and finally decrypts F.

D. Security Analysis

Since Comrade heavily rely on the security provisions of the Ethereum blockchain, we start by analyzing the security provisions of Ethereum and then explain how Comrade correctly instantiates the SOM model.

1) Blockchain Security: Recall that all transactions issued by the owners in Comrade are confirmed in the blockchain by the validators/miners. As required for the security of the underlying blockchain, we assume the standard safety conditions particular to the underlying blockchain technology. For instance, since Ethereum relies on Proof-of-Work, we assume that the adversary cannot control the majority of the computing power in the network (see [21] for further details). In practice, we assume that the adversary does not affect the mining process in Ethereum (i.e., does not act as a miner). Notice that malicious miners can decide not to include the transactions issued by the owners in Comrade. By doing so, the adversary can attempt to delay the confirmation of transactions issued by entities in Comrade for a short amount of time. Conforming with the current operation of Ethereum, we assume that the issuers of transactions will re-broadcast their transactions in case they are not included in the subsequent block. This will ensure that these transactions will be eventually confirmed by honest miners in the system—as long as the majority of the computing power harnessed in Ethereum is honest.

Moreover, we assume that the adversary cannot monopolize the connections of owners/nodes in the system. This prevents the adversary from mounting Eclipse attacks [27] and partial Eclipse attacks [28] to abuse consensus realization in Ethereum.

Finally, the adversary can try to exploit any existing vulnerability in the smart contract instantiating Comrade. However, given that the contract features a small codebase, we assume that the contract can be formally verified to prevent any such exploits from occurring in practice.

2) Comrade Security: Similar to Commune, in Comrade user authentication at the server and at the owner contract rules out external adversaries. Further, assuming that the aforementioned security provisions of the blockchain are met, the owner contract essentially instantiates a trusted third party that handles access policies and credentials. Moreover, since an untampered log of the system is available from the blockchain, computation of the contract can be publicly audited.

Each credential issued by an owner is instantiated by a blockchain transaction that is confirmed within the blockchain by the validator/miners. As such, the vote of an owner on an access control decision is eventually taken into account by the smart contract and persisted to the blockchain. Similarly, access requests for a file by owners are processed, saved to the blockchain, and evaluated by the smart contract against the policy define by the owners of that file.

As such, malicious writers (i.e., writers who have been granted write access on a resource by fewer than t owners) are denied access by the cloud PDP according to the owner contract. At the same time, copy-on-write prevents authorized writers from overwriting content in the shared repository with garbage (similarly to what happen in Commune).

Similarly, malicious readers (i.e., readers who have been granted read access on a resource by fewer than t owners) are denied access from the cloud's PDP as it evaluates the `hasAccess()` on the request of the malicious reader. Further, we note that revocation is correctly enforced since $\mathcal{S}$ evaluates access requests on the latest policy defined by the owners. Also, Comrade is collusion-resistant by design, because credentials granted by owners are issued against a given user identity and cannot be pooled to escalate rights.

Finally, we note that cloud providers have to evaluate `hasAccess` on every resource access. Since `hasAccess` is Turing-complete, the cloud provider must protect against resource exhaustion attacks, where clients trigger expensive `hasAccess` functions. Therefore, the cloud provider can define a maximum‘ number of execution steps for an evaluation of `hasAccess` and charge the owner contract according to the number of required execution steps. Notice that this is a similar concept as the notion of *gas* in Ethereum [4]. This ensures fair payments across different tenants and defends against resource exhaustion.

V. Prototype Design & Evaluation

In this section, we describe prototype implementation of Commune and Comrade integrated with Amazon S3 [5] and evaluate their performance.

A. *Commune Implementation*

We leverage Amazon S3 to instantiate $\mathcal{S}$: for each user in $\mathcal{U}$, we create personal accounts in Amazon S3, into which users can upload content and for which users can define arbitrary access control policies. In our implementation, we use Amazon S3 access control features to distribute tokens from the file creator to the set of owners $\mathcal{O} \subseteq \mathcal{U}$. In particular, we assume that each user sets up *(i)* one "temporary" folder where other peers are granted write access, and *(ii)* one "main" folder where endorsed tokens are stored and retrieved. When the file creator wants to distribute a token to owner O_j, he writes the token to O_j's temporary folder. Since no other user apart from O_j has read access to the temporary folder, the new token is protected from unauthorized access. At this point, O_j can endorse the token for another user U_l by storing the token in his main folder, and granting read access on it to U_l.

TABLE I

Transaction Fees in USD for Our Comrade Prototype

	New Owner Contract	New Permission	New File
4 Owners	3.05	0.08	0.00
8 Owners	5.47	0.08	0.00

Our Commune prototype, implemented in Java, is a multi-threaded client-side interface to repositories hosted on Amazon S3. The client runs on a user's machine and uploads/downloads content to/from the repositories. The client's implementation of SFD leverages Rijndael [15] as the underlying block cipher for AON-FFT and systematic Reed-Solomon codes [16] for information dispersal. We chose a symbol size of 16 bytes, and a security parameter $\lambda = 128$ bits.

To optimize performance, our prototype handles file unit operations at a smaller granularity, called *pieces*. During the creation of any file unit, the unit is split into pieces that are processed in parallel. A token for each unit contains one output chunk of SFD for each piece that composes the unit. The piece size w is chosen such that $t\lambda|w$, where λ is the security parameter and t is the required reconstruction threshold. This condition ensures that *(i)* a piece can be encrypted in an integer number of ciphertext blocks of λ bits, *(ii)* an encrypted piece can be divided into an integer number of input chunks for the Reed-Solomon encoder, and *(iii)* the size of each chunk of the Reed-Solomon encoder/decoder is at least λ bits.

B. *Comrade Implementation*

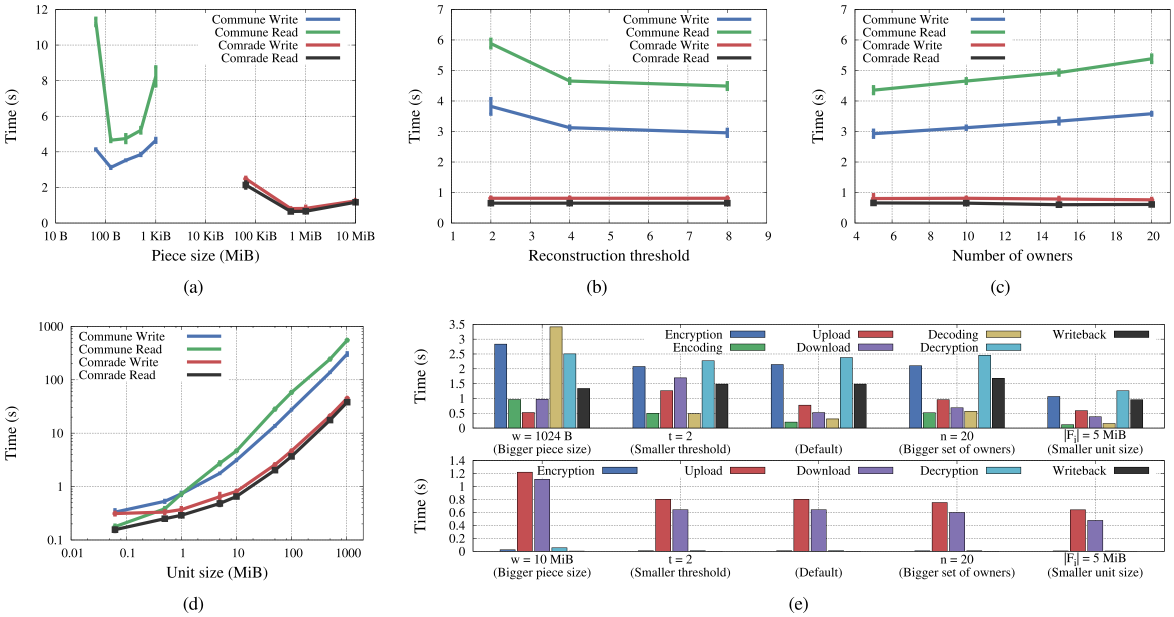
We implement Comrade using solidity-based smart contracts[5] on the Ethereum blockchain and a python-based client, which are all connected to a single Amazon S3 account, owned by the owner contract. Since Amazon does not support blockchain-aware PDPs yet, we implement the PDP in an Amazon EC2 instance. The PDP has access to the S3 account and makes all access control decisions based on the current state of the blockchain. For every file access, our clients contact the PDP.

For the deployment of our contracts, we use the existing, public Ethereum blockchain. The owner contract with the main logic of Comrade works follows. Clients vote on the access control directly through Ethereum transactions. Notice that we run tests on a private Ethereum [24] chain to avoid paying transaction fees. Table I summarizes the fees. The creation of our owner contract in Ethereum costs \$3.05[6] and \$5.47 for 4 and 8 owners respectively. Granting permissions costs \$0.08 while uploading a new file incurs no fees.

[4] http://ethdocs.org/en/latest/contracts-and-transactions/

[5] Full smart contract code is available at https://gist.github.com/ritzdorf/37ab1c3c57b9167c8418156a06c72b44

[6] At time of writing, the Gas price is $2 \cdot 10^{-9}$ ETH where 1 ETH = \$407.15.



(a) (b) (c) (d) (e)

Fig. 5. Latency evaluation of our prototype implementations. Each data point is averaged over 20 measurements; where appropriate, we also provide the corresponding 95% confidence intervals. Figure 5(e) splits up the latency into its different components. (a) Impact of the piece size. (b) Impact of the reconstruction threshold. (c) Impact of the number of owners. (d) Impact of the unit size. (e) Runtime analysis for five different configurations of Commune and Comrade below.

To support authentication with the PDP, every user registers its public key within the owner contract. Since the storage inside the blockchain is expensive, we use compact elliptic curve cryptography (ECC) (since ECC public keys are smaller than RSA keys). To access a file, a client establishes a TLS connection to the PDP using its registered public key inside a client certificate. The PDP identifies the client based on the key and makes the access control decision by locally evaluating a function of the owner contract. Similar to Commune, our Comrade prototype breaks units up into pieces.

In our implementation, we require that the encrypted, shared files and the wrapped keys are stored on Amazon S3. Notice that, unlike Commune, no additional redundancy for stored files is required in Comrade. Therefore, the wrapped keys represent the only storage overhead incurred by Commune. This overhead amounts to 36 bytes per wrapped key; hence the storage overhead per file is 36 bytes times the number of file owners. Notice that this is a negligible storage overhead since most shared files are at least 1 MB in size.

C. Evaluating Single Unit Write/Read

We evaluate the performance of Commune and Comrade for a single file unit write and read, with respect to *(i)* the piece size w (default value $w = 128$ bytes), *(ii)* the reconstruction threshold t (default value $t = 4$), *(iii)* the number of owners n (default value $n = 10$), and *(iv)* the size of the file unit $|F_i|$ (default value $|F_i| = 10$ MiB).

We then change one variable at a time to assess its impact on the system performance. For each configuration, we measure the time required *(i)* to create and upload F_i (denoted by *Write* in our plots), and *(ii)* to retrieve F_i (denoted by *Read*). These latencies are measured from the initiation of the operation until the output is available either in the repositories (for *Write*) or on a local disk (for *Read*). We control for the effect of caching by uploading random binary streams at each repetition.

During *Read*, the Commune client fetches endorsed tokens from t randomly chosen owners. Recall that a (t, n) systematic erasure code outputs t data chunks and $n - t$ parity chunks. Since data chunks need not be decoded, our evaluation accounts for the average-case scenario where the probability that a token contains a data chunk is bounded by $\frac{t}{n}$. Notice that we do not evaluate the time required to grant read rights (i.e., the time required to endorse a token or to submit a blockchain transaction) since it does not depend on any of the considered parameters.

Our results are depicted in Figure 5. For Commune, we additionally monitor the runtime of the intermediate steps for a number of configurations as shown in Figure 5(e).

Our evaluation shows that writing a new unit in Commune (*Write*) is less expensive than reading it (*Read*) while the order is reversed in Comrade. The former effect is due to the overhead of thread synchronization when storing decoded pieces on the local disk while the *Write* performance in Comrade is due to the overhead of uploading wrapped keys for all owners.

1) Impact of the Piece Size: Figure 5(a) shows the impact of the piece size w on the latency. For Commune, a smaller w leads to a smaller number of input blocks to the AON-FFT scheme, which results in better performance since AON-FFT requires $\log_2 m$ rounds of encryption for m input blocks. However, we experience higher latencies for very small values of w, especially in the *Read* operation. This is due to the thread synchronization overhead when writing data to disk. For Comrade, a smaller w increases the overhead because of additional synchronization while a larger w cannot benefit from parallelism. Throughout the rest of the evaluation, we set $w = 128$ B for Commune and $w = 512$ KiB for Comrade since they offer a good performance trade-off as shown in Figure 5(a).

2) Impact of the Reconstruction Threshold: Figure 5(b) shows the latency impact of the reconstruction threshold t. In Comrade, the threshold does not influence latency as shown in Figure 5(b). In Commune, the chunk size of the Reed-Solomon encoder increases as t decreases; this results in larger chunk upload and download times. Figure 5(e) also shows that a smaller value of t results in longer encoding and decoding times. On the one hand, during *Write*, small values of t result in larger encoding overhead since the size of the encoding matrix increases. On the other hand, during *Read*, small values of t decrease the probability of recovering data chunks (w.r.t. the probability of recovering parity chunks), which makes decoding slower (cf. Figure 5(e)).

3) Impact of the Number of Owners: Figure 5(c) shows that latency increases for Commune's *Read* and *Write* as the number of owners grows. The latency increase during *Read* is due to an higher probability of fetching parity codes that take more time to be decoded by the Reed-Solomon decoder. During *Write*, this increase is caused by the creation and distribution of additional tokens from the file creator to the set of owners. The performance of Comrade is virtually unaffected by the number of owners since the upload of additional wrapped keys can be performed in parallel—thereby resulting in negligible additional cost.

4) Impact of the Unit Size: Figure 5(d) shows Commune's and Comrade's latency for different unit sizes. The time required to read/write a unit increases almost linearly with the unit size (Figure 5(d) relies on semi-logarithmic axes). However, the performance of Comrade is a magnitude faster than that of Commune. The time required to read a 10 MB unit is roughly 4.47 seconds for Commune, but only 0.81 seconds for Comrade. As shown in Figure 5(e), this stems from the fact that Commune's latency is dominated by the encryption and decryption as part of AON-FFT, while Comrade does not leverage AON-FFT and therefore witnesses a considerably lower latency.

D. Evaluating Multiple Units Read/Write

We now determine the peak throughput exhibited by our prototype implementations. Here, we increase the number of concurrently accessed units until the throughput is saturated. We then compute the peak throughput as the maximum aggregated amount of data in bits per second that can be transferred between client and Amazon S3. Table II shows that the peak read/write throughput is above 29 Mbps for Commune and above 190 Mbps for Comrade.

TABLE II

Peak Throughput. Each Data Point Is the Average of 20 Measurements

	Peak Throughput (Mbps)	
	Commune	Comrade
Write	43.39	190.26
Read	29.52	225.37

We argue that, while Commune's overhead might be tolerable in low-throughput, high-latency scenarios such as collaborative text editing where users work on content on their local machines (and only periodically synchronize content with the cloud), Comrade is a viable option in a wide variety of application scenarios including those with more frequent cloud interactions.

VI. Discussion

In this section, we discuss further insights into the design of and possible extensions of Commune and Comrade.

1) Transparency to Users: As explained, Commune enables users to coordinate access control to cloud content in a distributed manner. We stress that all the operations in Commune are implemented by the client application described in Section V. Users need not "manually" distribute or fetch tokens. In fact, users are only required to set the list of owners for the files they create and to define the access policy on the files for which they are appointed as owners.

In Comrade, the owners initially create the owner contract. Afterwards, the owner contract acts as an orchestrator for all users. Comrade transparently fetches ciphertexts and the corresponding wrapped keys.

2) Changing threshold t: To maintain consistency in Commune, we do not support the change of threshold t for any file F. If an owner would want to change the threshold, say from t to t', he would have to compute and distribute new tokens to *all* owners in $\mathcal{O}$. Then, *all* owners in $\mathcal{O}$ must replace their old tokens with the newly received ones. Since each owner has full rights on its tokens, there is no mechanism to force all owners to accept these changes, and replace their tokens. This can lead to an inconsistent state in which some tokens correspond to a file version with threshold t, while other tokens correspond to another version with threshold t'. Therefore, Commune does not support changing the threshold.

In contrast, Comrade supports changing threshold t by modifying the owner contract. The owner contract defines the requirements for such a change, e.g., agreement by all owners. Once the requirements for a change are fulfilled, the change takes effect and future evaluations of the owner contract through the cloud PDP use the updated threshold.

3) Adding/Revoking Owners: Our model assumes that the set of owners $\mathcal{O}$ is defined before file creation. Adding an owner in Commune requires that either the original file creator or at least t out of the n owners provide the new owner with his set of tokens. However, revoking ownership rights from an owner, say O_j, may not be feasible since tokens cannot be removed from O_j's storage on $\mathcal{S}$ without his consent. One possible solution would be to re-encode the file and distribute new tokens to owners in $\mathcal{O} \setminus \{O_j\}$. Nevertheless, similar to the case of changing the threshold t, some of the owners in $\mathcal{O} \setminus \{O_j\}$ may decide to discard the new tokens and keep the old ones—leading to an inconsistent state.

In Comrade, owners can be added and revoked through the owner contract. The requirements for adding or revoking owners are mandated by the owner contract which can require e.g., the approval by a majority of owners. Afterwards, the owner list inside the owner contract is updated and new owner votes take effect (or obsolete owner votes are disregarded).

4) Fine-Grained Per-Version Access Control: Commune and Comrade enable owners to perform per-version access control. That is, owner O_j can, for example, grant U_l read access to version x of a file F but deny U_l access to F's version x'. In collaborative scenarios some versions of a given file may contain information only intended for a subset of the users (e.g., due to IPR protection).

Note that, due to versioning, a given unit may span several versions of file F. Nevertheless, we argue that this is transparent to the user who only decides whether to grant/deny access to a given version x, while tokens are handled by the client application.

VII. Related Work

Current state-of-the-art access control systems, such as SecPAL [1], KeyNote [2], and Delegation Logic [3], can in principle express t out of n policies. These languages, however, rely on the presence of a centralized PDP component to evaluate their policies. Furthermore, their PDPs cannot be deployed within a third-party cloud platform. As explained in Section II, these access control systems rely on an administrator to define and manage access control policies. In our setting, this means that a set of owners has to elect one enforcer who has unilateral powers over their files.

Multi-Authority Attribute Based Encryption (MA-ABE) [18], [29] is a powerful tool that may be used to address the problem of shared ownership and allow multiple parties to collaboratively control access to a shared resource. However, most existing proposals for MA-ABE require a bilinear map, hence they incur in expensive operations and rely on novel cryptographic assumptions. In this paper, we rely on CRSS instead. Although CRSS only allows threshold policies, it only requires a cyclic group of prime order and relies on standard assumptions. We argue that for the application at hand, threshold policies are sufficient and therefore our system can benefit from the low complexity of CRSS. Furthermore, MA-ABE, just like CRSS, may only be used to regulate access to an encryption key. Regulating access to large files requires a different approach such as combining CRSS with SFD.

Secret sharing schemes [30] allow a dealer to distribute a secret among a number of shareholders, such that only authorized subsets of shareholders can reconstruct the secret. In threshold secret sharing schemes [20], [31], the dealer defines a threshold t and each set of shareholders of cardinality equal to or greater than t is authorized to reconstruct the secret. Secret sharing guarantees security (i.e., the secret cannot be recovered) against a non-authorized subset of shareholders; however, they incur a high computation/storage cost, which makes them impractical for sharing large files.

Rabin [11] proposed an information dispersal algorithm with smaller overhead than that of [31], however, his proposal does not provide any security guarantees when a small number of shares (fewer than the threshold) are available. Krawczyk [32] combines both Shamir's [31] and Rabin's [11] approaches; in [32] a file is first encrypted using AES and then dispersed using the scheme in [11], while the encryption key is shared using the scheme in [31].

Information dispersal based on erasure codes [16] are effective tools to enhance the reliability of cloud-based storage systems [33]–[36]. Ramp schemes [37] constitute a trade-off between the security guarantees of secret sharing and the efficiency of information dispersal algorithms.

5) All or Nothing Transformations: All-or-nothing transformations were first introduced in [13] and later investigated in [14] and [38]. The majority of AONTs leverage a secret key that is embedded in the output blocks. Once all output blocks are available, the key can be recovered and single blocks can be reverted. Rivest [13] also mentioned a transformation that is inspired by Fast Fourier Transform. Van Dijk *et al.* [17] later on leveraged Rivest's transformation to construct a "proof of encryption" of files in the cloud. In this paper, we extend the use of Rivest's transformation to construct an AONT scheme, that keeps its all-or-nothing property even if the adversary is given the secret key. Resch and Plank [12] combine AONT and information dispersal to provide both fault-tolerance (i.e., decoding requires only t out of n shares) and data secrecy (i.e., confidentiality is guaranteed w.r.t. parties that collect fewer than t shares), in the context of distributed storage systems. In [12], however, an adversary who caches the encryption key can still decode single shares. In [39], Karame *et al.* showed that by first encrypting the data then post-processing it using a linear transform, one can construct an *encryption mode* which provides similar guarantees as all or nothing transforms, and with comparable performance.

VIII. Conclusion

Even though existing cloud platforms are used as shared repositories, they do not support any notion of shared ownership. We consider this a severe limitation because contributing parties cannot jointly decide how their resources are used.

In this paper, we introduced a novel concept of shared ownership and we described it through a formal access control model, called SOM. We then propose two possible instantiations of our proposed shared ownership model. Our first solution, called Commune, relies on secure file dispersal and collusion-resistant secret sharing to ensure that all access grants in the cloud require the support of an agreed threshold of owners. As such, Commune can be used in existing agnostic clouds without modifications to the platforms. Our second solution, dubbed Comrade, leverages the blockchain technology in order to reach consensus on access control decision. Unlike Commune, Comrade requires that the cloud is able to translate access control decisions that achieved consensus in the blockchain into storage access control rules. Comrade, however, shows better performance than Commune.

Given the rise of personal clouds (e.g., [9], [10]), we argue that Commune and Comrade find direct applicability in setting up shared repositories that are distributively managed atop of the various personal clouds owned by users. We therefore hope that our findings motivate further research in this area.

References

[1] M. Y. Becker, C. Fournet, and A. D. Gordon, "SecPAL: Design and semantics of a decentralized authorization language," *J. Comput. Secur.*, vol. 18, no. 4, pp. 619–665, 2010.

[2] M. Blaze, J. Ioannidis, and A. D. Keromytis, "Trust management for IPsec," *ACM Trans. Inf. Syst. Secur.*, vol. 5, no. 2, pp. 95–118, 2002.

[3] N. Li, B. N. Grosof, and J. Feigenbaum, "Delegation logic: A logic-based approach to distributed authorization," in *Proc. TISSEC*, 2003, pp. 128–171.

[4] C. Soriente, G. O. Karame, H. Ritzdorf, S. Marinovic, and S. Čapkun, "Commune: Shared ownership in an agnostic cloud," in *Proc. SACMAT*, 2015, pp. 39–50.

[5] *Amazon Simple Storage Service (S3)* Accessed: 2018. [Online]. Available: http://aws.amazon.com/s3/

[6] S. Ceri, G. Gottlob, and L. Tanca, "What you always wanted to know about Datalog (and never dared to ask)," *IEEE Trans. Knowl. Data Eng.*, vol. 1, no. 1, pp. 146–166, Mar. 1989.

[7] Y. Gurevich and I. Neeman, "DKAL: Distributed-knowledge authorization language," in *Proc. CSF*, Jun. 2008, pp. 149–162.

[8] J. DeTreville, "Binder, a logic-based security language," in *Proc. IEEE Symp. Secur. Privacy*, May 2002, pp. 105–113.

[9] *The Respect Network*. [Online]. Available: https://www.respectnetwork.com/

[10] *WD My Cloud*. [Online]. Available: http://www.wdc.com/en/products/products.aspx?id=1140

[11] M. O. Rabin, "Efficient dispersal of information for security, load balancing, and fault tolerance," in *J. Assoc. Comput. Mach.*, vol. 36, no. 2, pp. 335–348, 1989

[12] J. K. Resch and J. S. Plank, "AONT-RS: Blending security and performance in dispersed storage systems," in *Proc. FAST*, 2011, pp. 1–12.

[13] R. L. Rivest, "All-or-nothing encryption and the package transform," in *Proc. Int. Workshop Fast Softw. Encryption (FSE)*, 1997, pp. 210–218.

[14] V. Boyko, "On the security properties of OAEP as an all-or-nothing transform," in *Proc. CRYPTO*, 1999, pp. 503–518.

[15] J. Daemen and V. Rijmen. (1999). AES Proposal: Rijndael. [Online]. Available: http://csrc.nist.gov/archive/aes/rijndael/Rijndael-ammended.pdf

[16] J. H. van Lint, *Introduction to Coding Theory*. New York, NY, USA: Springer-Verlag, 1982.

[17] M. van Dijk, A. Juels, A. Oprea, R. L. Rivest, E. Stefanov, and N. Triandopoulos, "Hourglass Schemes: How to prove that cloud files are encrypted," in *Proc. CCS*, 2012, pp. 265–280.

[18] A. B. Lewko and B. Waters, "Decentralizing attribute-based encryption," in *Proc. EUROCRYPT*, 2011, pp. 568–588.

[19] P. Rogaway and M. Bellare, "Robust computational secret sharing and a unified account of classical secret-sharing goals," in *Proc. CCS*, 2007, pp. 172–184.

[20] C. Charnes, J. Pieprzyk, and R. Safavi-Naini, "Conditionally secure secret sharing schemes with disenrollment capability," in *Proc. CCS*, 1994, pp. 89–95.

[21] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," Tech. Rep., 2008.

[22] F. Armknecht, J.-M. Bohli, G. O. Karame, Z. Liu, and C. A. Reuter, "Outsourced proofs of retrievability," in *Proc. CCS*, 2014, pp. 831–843.

[23] F. Armknecht, J.-M. Bohli, G. O. Karame, and F. Youssef, "Transparent data deduplication in the cloud," in *Proc. CCS*, 2015, pp. 886–900.

[24] V. Buterin, "A next-generation smart contract and decentralized application platform," White Paper, 2016. [Online]. Available: https://github.com/ethereum/wiki/wiki/White-Paper

[25] F. Zhang, E. Cecchetti, K. Croman, A. Juels, and E. Shi, "Town Crier: An authenticated data feed for smart contracts," in *Proc. CCS*, 2016, pp. 270–282.

[26] D. F. Ferraiolo and D. R. Kuhn, "Role-based access controls," in *Proc. 15th NIST-NSA Nat. Comput. Secur. Conf.*, 1992, pp. 554–563.

[27] E. Heilman, A. Kendler, A. Zohar, and S. Goldberg, "Eclipse attacks on bitcoin's peer-to-peer network," in *Proc. SEC USENIX Assoc.*, 2015, pp. 129–144.

[28] A. Gervais, H. Ritzdorf, G. O. Karame, and S. Čapkun, "Tampering with the delivery of blocks and transactions in bitcoin," in *Proc. CCS*, 2015, pp. 692–705.

[29] Y. Rouselakis and B. Waters, "Efficient statically-secure large-universe multi-authority attribute-based encryption," in *Financial Cryptography and Data Security*. IACR, 2015.

[30] A. Beimel, "Secret-sharing schemes: A survey," in *Proc. 3rd Int. Workshop Coding Cryptol. (IWCC)*, 2011, pp. 11–46.

[31] A. Shamir, "How to share a secret," *Commun. ACM*, vol. 22, no. 11, pp. 612–613, 1979.

[32] H. Krawczyk, "Secret sharing made short," in *Proc. Int. Conf. Adv. Cryptol.*, 1993, pp. 136–146.

[33] H. Xia and A. A. Chien, "RobuSTore: A distributed storage architecture with robust and high performance," in *Proc. SC*, Nov. 2007, pp. 1–11.

[34] M. Abd-El-Malek, G. R. Ganger, G. R. Goodson, M. K. Reiter, and J. J. Wylie, "Fault-Scalable Byzantine Fault-Tolerant Services," in *Proc. SOSP*, 2005, pp. 59–74.

[35] J. Kubiatowicz *et al.*, "Oceanstore: An architecture for global-scale persistent storage," in *Proc. ASPLOS*, 2000, pp. 199–201.

[36] M. K. Aguilera, R. Janakiraman, and L. Xu, "Using erasure codes efficiently for storage in a distributed system," in *Proc. DSN*, 2005, pp. 336–345.

[37] G. R. Blakley and C. Meadows, "Security of ramp schemes," in *Advances in Cryptology*. IACR, 1984, pp. 242–268.

[38] A. Desai, "The security of all-or-nothing encryption: Protecting against exhaustive key search," in *Advances in Cryptology*. IACR, 2000, pp. 359–375.

[39] G. O. Karame, C. Soriente, K. Lichota, and S. Čapkun, "Securing cloud data in the new attacker model," IACR Cryptol. ePrint Arch., Tech. Rep., 2014.

Hubert Ritzdorf received the bachelor's degree from the University of Bonn, Germany, the master's degree from ETH Zürich, Switzerland, and the Ph.D. degree from the System Security Group, Department of Computer Science, ETH Zürich. He is currently the CTO of ChainSecurity.

Claudio Soriente received the Ph.D. degree from the University of California at Irvine, Irvine, CA, USA. He was with the Swiss Federal Institute of Technology, Zürich, Switzerland, and with the Polytechnic University of Madrid, Spain. He was also a Researcher with Telefonica Research. He is currently a Senior Researcher with NEC Laboratories Europe. His research interests include network and wireless security, privacy, and applied cryptography.

Ghassan O. Karame (M'11) received the M.Sc. degree in information networking from Carnegie Mellon University in 2006 and the Ph.D. degree in computer science from ETH Zürich, Switzerland, in 2011. From 2011 and 2012, he was a Post-Doctoral Researcher with the Institute of Information Security, ETH Zürich. He is currently the Manager and a Chief Researcher with the Security Group, NEC Laboratories Europe. He is interested in all aspects of security and privacy with a focus on cloud security, SDN/network security, and bitcoin security. He is a member of the ACM. More information about him can be found in the web page http://ghassankarame.com/.

Srdjan Marinovic received the Ph.D. degree in computer security from Imperial College London. Then, he joined the Information Security Institute, ETH Zürich, as a Senior Researcher. He is currently serving as the CTO for The Wireless Registry Inc., a startup based in Washington, DC.

Damian Gruber received the bachelor's degree from ETH Zürich, Switzerland. Afterward, he completed the Information Security track at ETH Zürich, and received the master's degree in 2016. He is currently an Intern with the Security Group, NEC Laboratories Europe, Germany.

Srdjan Capkun received the Dipl.Ing. degree in electrical engineering/computer science from the University of Split, Croatia, and the Ph.D. degree in communication systems from EPFL in 2004. He was a Post-Doctoral Researcher with the Networked and Embedded Systems Laboratory, UCLA, and an Assistant Professor with the Technical University of Denmark. He is currently a Full Professor with the Department of Computer Science, ETH Zürich, and the Director of the Zürich Information Security and Privacy Center.