METAL
A Metadata-Hiding File-Sharing System

Weikeng Chen
UC Berkeley
weikeng@eecs.berkeley.edu

Raluca Ada Popa
UC Berkeley
raluca@eecs.berkeley.edu

Abstract

File-sharing systems like Dropbox offer insufficient privacy because a compromised server can see the file contents in the clear. Although encryption can hide such contents from the servers, metadata leakage remains significant. The goal of our work is to develop a file-sharing system that hides metadata—including user identities and file access patterns.

Metal is the first file-sharing system that hides such metadata from malicious users and that has a latency of only a few seconds. The core of Metal consists of *a new two-server multi-user oblivious RAM (ORAM)* scheme, which is secure against malicious users, a *metadata-hiding access control* protocol, and a *capability sharing* protocol.

Compared with the state-of-the-art malicious-user file-sharing scheme PIR-MCORAM (Maffei et al.'17), which does not hide user identities, Metal hides the user identities and is $500\times$ faster (in terms of amortized latency) or $10^5\times$ faster (in terms of worst-case latency).

More details of Metal can be found on https://www.oblivious.app/.

Contents

1 Introduction

Storing files on a cloud server and sharing these files with other users (e.g., as in Dropbox) are common activities today. To hide the confidential contents of files from a compromised server, academia and industry developed end-to-end encryption (E2EE) systems [GSMB03; KRS+03; BCQ+11; PSV+14; HAJ+14; LCS+14; WMZV16]; using these, the user encrypts the file contents, so a compromised server only sees the encryption of a file, and only permitted users can decrypt the file. Unfortunately, this approach leaves unprotected a lot of user and file metadata. Figure 1 summarizes metadata leakage in E2EE systems: notably, the user identities and file access patterns.

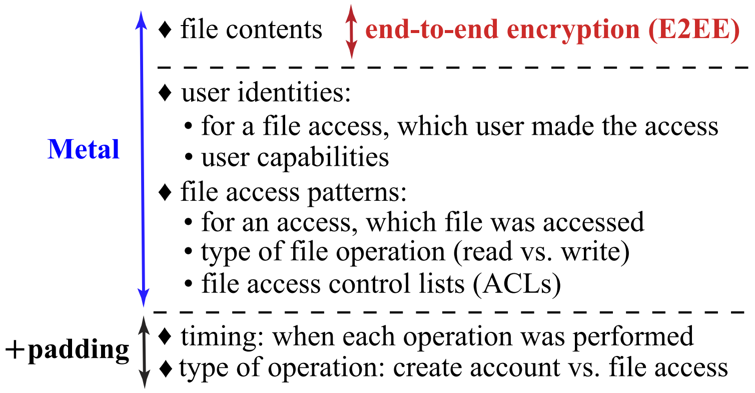
♦ file contents
end-to-end encryption (E2EE)
Metal
♦ user identities:
• for a file access, which user made the access
• user capabilities
♦ file access patterns:
• for an access, which file was accessed
• type of file operation (read vs. write)
• file access control lists (ACLs)
+padding
♦ timing: when each operation was performed
♦ type of operation: create account vs. file access

Figure 1: Scope of data/metadata protected by end-to-end encryption (E2EE) systems and Metal; padding in time and computation hides more metadata.

Such metadata is sensitive, which has become notorious in a related area—communication surveillance. Former NSA General Counsel, Stewart Baker, said *"Metadata absolutely tells you everything about somebody's life. If you have enough metadata, you don't really need content."* [Rus13] Former NSA Director, Michael Hayden, stated: *"We kill people based on metadata."* [Col14] Since knowing whom a user calls is similar in spirit to knowing with whom a user shares a file, leaking metadata in file sharing is also worrisome. To illustrate this issue, consider some privacy concerns that arise when medical data is stored on the cloud:

The sensitive user identities. Consider that patient Alice and her oncologist Bob share Alice's medical profile in an E2EE system. Even with encryption, the server knows that Alice and Bob share some files with each other. With the side information that Bob is an oncologist, likely available from a Google search for Bob's name, the server knows that Alice is seeing an oncologist and thus infers that she suffers from cancer.

The sensitive file access patterns. Even without user identities, file access patterns are sensitive. Consider that some doctors share a folder with disease handouts that consists of many files, one for each disease. The server sees the access frequency of each file and can relate it to disease incidence rates, which can be found online [WHO]. Thus, the server can infer the disease in each file. If the server knows the time when Alice goes to the doctor, the server can infer Alice's disease by seeing which file is accessed by the doctor.

Further, there are many general attacks against anonymous systems leveraging social data [Agg05; BDK07; NS08; WHKK10; NKA14; JLM+15; JLG+15; GMN+16] or access patterns [IKK12; CGPR15; ZKP16; GMN+16; WGL+17; LMP18; GLMP18; GLMP19], some of which might apply to file sharing.

The first attempt to hide such metadata is oblivious RAM (ORAM) [Gol87; Ost90; SDS+13]. However,

Work	**File sharing**	**Hide access patterns**	**Hide users**	**Server complexity**	**Server assumption**
Secret-write PANDA [HOWW19]	No	No	No	Nearly polylog	1-server
AnonRAM-lin [BHKP16]	No	Yes	Yes	Linear	1-server
AnonRAM-poly [BHKP16]	No	Yes	Yes	Polylog (linear worst-case)	2-server
GORAM [MMRS15]	Yes	No	Partial	Polylog	1-server
PIR-MCORAM [MMRS17]	Yes	Partial	No	Linear	1-server
Metal (this paper)	Yes	Yes	Yes	Polylog	2-server

Table 1: Comparison of multi-user ORAM schemes when ***there are an unbounded number of malicious users***. All the listed schemes assume the server(s) to be semi-honest. The server computation complexity here is in respect to the number of files, assuming each file is of a constant size. The comparison will be discussed in more detail in Section 9.

ORAM relies on the trustworthiness of a *single* client or a proxy to maintain the confidentiality of the entire data storage. A recent line of *multi*-user ORAM schemes [MMRS15; BHKP16; MMRS17; HOWW19], shown in Table 1, is more relevant to our setting. These schemes attempt to retain some oblivious guarantees even when some users are compromised; for example, file accesses of an honest user remain hidden across all the files accessible only by honest users. Unfortunately, there are very few such works; the schemes that support file sharing, PIR-MCORAM [MMRS17] and GORAM [MMRS15], leak either user identities or file access patterns, as depicted in Table 1.

This paper presents Metal, the first cryptographic file-sharing system that hides both user identities and file access patterns both from the server and from malicious users. Figure 1 lists the various types of metadata that Metal protects. The scheme with the closest security guarantees, PIR-MCORAM [MMRS17], has a very high overhead. Although Metal is not a lightweight system either, it makes a big leap toward reaching practicality—Metal's access latency is $\geq 500\times$ (for amortized latency) or $\geq 10^5\times$ (for worst-case latency) shorter than that of PIR-MCORAM, and in absolute value, only a few seconds.

PIR-MCORAM's very high overhead is largely due to an unfortunate lower bound that challenges this research area: Maffei et al. [MMRS17] showed that, to hide access patterns, a single-server file-sharing system must basically scan all the files; hence, PIR-MCORAM scans every file in the system for each file access. To avoid this impossibility result, AnonRAM-poly [BHKP16] adopts a *two-server model*, where at least one server is honest. This model is also adopted by much prior work in related settings for similar reasons [MZ17; WYG+17; CB17; KGK+18]. Metal adopts this two-server model as well.

Unfortunately, even in the two-server model, efficiency remains a troubling challenge. Putting aside the fact that worst-case accesses in AnonRAM-poly are still linear, a significant inefficiency in AnonRAM-poly is that each user's access requires the user to generate a heavy zero-knowledge proof (to prove to the servers that this user did not maliciously deviate from the protocol). Generating such a proof is already $20\times$ times slower than the overall access time of Metal (as described in Section 7.6). Further, AnonRAM-poly does not support file sharing; extending AnonRAM-poly to file sharing is challenging because its design makes it difficult to hide the access patterns across files with different sharing permissions. Finally, given the complexity, the authors of AnonRAM-poly have not implemented AnonRAM-poly.

With Metal, we propose a radically different design than AnonRAM-poly, which centers around file sharing and obviates the need for zero-knowledge proofs while resisting malicious users. In Section 7, we evaluate Metal extensively and show that its access time is within a few seconds for a file store of 2^{20} 64 KB files. We now overview Metal's techniques.

1.1 Overview of Metal's techniques

As a file-sharing system, Metal provides users with the ability to access files and to share permissions to files. When a user makes a request to Metal's servers, Metal checks if the user has the required permission, then the user can fetch or share a file. To understand how Metal performs these operations securely, we now overview Metal's techniques, organized by the challenges they address.

Challenge: Single-user nature of ORAM. ORAM [Gol87; Ost90; SCSL11; SDS+13] can hide access patterns, but it supports only a single client. To share an ORAM with many users, prior work proposes trusting a proxy or trusting all users [WST12; SS13b; BNP+15; SZE+16; CS19], which does not guarantee security in the presence of malicious users.

Primitive Metal: Inspired by ORAM, we start with a primitive construction of Metal (Section 5.2), which we describe as follows. In Primitive Metal, the two servers interact and run secure two-party computation (*S2PC*) [Yao86; GMW87; BMR90], as we illustrate in Figure 2. The reader should intuitively think that what happens inside S2PC is "**safe**" (albeit expensive as we will see), and what happens outside is "unsafe". Now, the servers can run a *global single-user ORAM client* inside their S2PC, which ensures that neither server sees the state of this global ORAM client, as well as other components for access control and capability sharing. To store and share files, the users communicate with the global ORAM client in the S2PC, which accesses files stored in the servers' ORAM storage on a user's behalf.

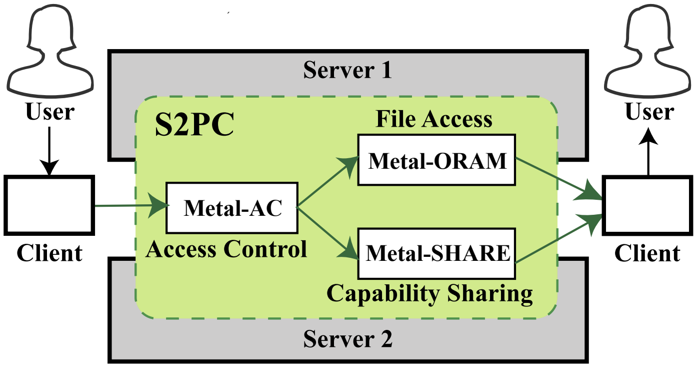

Figure 2: Metal's system architecture (described in Section 2.1).

This primitive scheme enables users to share files. For the servers to implement a service in this way, the S2PC protocol needs to be *reactive* [GKK+12; LO13; NR16]: the servers *repeatedly* take input into the S2PC, update some internal state, and provide output.

Challenge: Inefficiency of Primitive Metal. Though Primitive Metal has the desired security guarantees, it is highly inefficient. Our evaluation in Section 7.7 shows that the client ORAM access in S2PC requires a huge amount of server-server communication: $\geq$ 1 GB for each file access. It also requires $\geq$ 75 s to access one file in a store of 2^{20} 64 KB files.

Metal-ORAM: In Primitive Metal, the communication is high because the trusted global ORAM client, which runs inside S2PC, takes the file contents as input. We address this problem using our technique called ***synchronized inside-outside ORAM trees (Section 5.3)***, as follows. Metal maintains two ORAM trees on the servers: one containing the file contents called DataORAM and another one containing files' indices and locations called IndexORAM. DataORAM stays *outside* S2PC because it is large, and IndexORAM stays *inside* S2PC because it is small. Metal maintains the two ORAMs *synchronized*: after locating a file's identifier in the IndexORAM, one can find the file contents at the *same* location in DataORAM.

To keep the two ORAMs synchronized, the S2PC must apply the maintenance operations of an ORAM to IndexORAM and DataORAM *in the same way*; these operations include path selection and stash eviction. However, it is unclear how to capture these operations inside S2PC and how to apply them securely to DataORAM, which is outside S2PC.

For this problem, we develop our ***tracking and permutation generation technique (Section 5.6)***, which works as follows. During the ORAM access in IndexORAM, our circuit inside S2PC *tracks* the transformations applied to IndexORAM and converts them into a permutation. It turns out that the transformations are not naturally a permutation, but by "resurrecting" missing blocks in a certain way, Metal succeeds to create a permutation. Then, we use a custom S2PC protocol to apply the permutation securely and efficiently to DataORAM.

Altogether, in Metal, the general S2PC no longer touches the file data but works with the position maps and block locations, which reduces the overhead by $\approx 20\times$. We call this scheme Metal-ORAM, and we expect these techniques to be useful for other secure multi-party computation (SMPC) protocols.

Challenge: Performing oblivious access control in S2PC. A natural solution for file access control is to obliviously verify, inside S2PC, that the user's name appears in the file's access control list. However, since a file could involve thousands of users, checking the access control list in S2PC is expensive.

Metal-AC: Metal designs ***capability-based anonymous access control***, which we call Metal-AC (AC refers to access control); the unit of our access control, the *capability*, is inspired by the classical systems concept of a capability [DH66]. The key differences in Metal are that, given a capability, the servers cannot tell which file (or user) the capability is for, and that the capability is implemented cryptographically, checked inside S2PC by the two servers. By doing so, Metal-AC avoids the heavy handling of access control lists.

Challenge: Establishing anonymous identities. To preserve user anonymity, users must hide their real-world identities (e.g., email addresses) when sharing files. Simply choosing a pseudonym is insufficient because the servers or the malicious users can link the activities that involve this pseudonym together. Even if a user creates multiple accounts, the sharing of files between these accounts can link them together.

Metal-SHARE: In Metal, users share files via ***anonyms***, each of which is a secret identity exclusively shared between a pair of users. Different from traditional pseudonyms, a user's many anonyms are ***unlinkable*** to one another, so they will not reveal a user's identity even when put together. Metal's anonyms also permit one-sided anonymity; e.g., an anonymous whistleblower can send a file to a specific journalist.

We call this scheme Metal-SHARE. This scheme is efficient: even if a user creates millions of anonyms, the user's effort to receive a new file does not increase with the number of anonyms—Metal's client accumulates all file capabilities shared with a user even when they are under different anonyms.

When designing Metal with these strong privacy guarantees in mind, a number of other challenges popped up. For example, some naive solutions enable the servers to see how many files a user has received. Padding to the maximum number of files for each user is very expensive. Instead, Metal instantiates ***capability broadcast (Section 6.2)*** on the servers, which hides the per-user numbers of received files.

We describe Metal's security guarantees (Section 2.3) and provide security proof sketches in the paper.

2 Overview

We now describe Metal's system architecture, threat model, and security guarantees.

2.1 System architecture

Figure 2 shows Metal's system architecture, which consists of two servers and many users:

- The two servers run a secure two-party computation (S2PC) procedure (green part in Figure 2). This procedure is a *reactive* S2PC protocol: it continuously receives input, updates its internal state, and produces output.
- Each user runs a Metal client on the user's device. The user invokes the client's user-facing API functions (shown in Table 2) and receives results from the Metal client.
- The Metal client sends requests to the servers. The servers convert the requests to inputs to the S2PC procedure and run the S2PC. The servers then send the output from the S2PC procedure to the client (on the right of Figure 2).

Components. Metal consists of three components: Metal-AC for access control, Metal-ORAM for file access, and Metal-SHARE for capability sharing.

As Figure 2 shows, the client's request arrives at the first component, Metal-AC, which checks whether the user has the required permission. If so, the request is dispatched to Metal-ORAM for accessing a file or to Metal-SHARE for sharing.

API functions. The Metal client provides the user with some API functions (shown in Table 2). The client translates user API calls to requests to the servers, processes the servers' responses, and returns the results to the user. In addition, the client stores and manages the user's secret keys and capabilities in Metal.

Syntax of user-facing API functions	**Description**
CreateAccount$() \to U, \{F_{U,1}, F_{U,2}, ..., F_{U,\ell_{\mathsf{file}}}\}$	A user creates a new account U and creates ℓ_{file} empty files on the servers (Section 4).
ReadFile$(U, F) \to$ `fileContent`	A user with account U reads the file identified by F from the servers (Section 5.3).
WriteFile$(U, F,$ `newFileContent`$)$	A user with account U writes to the file identified by F on the servers (Section 5.3).
NewAnonym$(U) \to A_{U,i}$	A user with account U generates a new anonym $A_{U,i}$ with anonym index i (Section 6.1).
SendCapability$(V, F_V, A_{U,i},$ `permission`$)$	Another user with account V and file F_V sends a capability to access F_V (permission is *read*, *write*, or *read+write*) to the user who owns anonym $A_{U,i}$ (Section 6).
ReceiveCapability$(U) \to (F_V,\ A_{U,i},$ `permission`$)$	A user with account U receives a capability to file F_V from another user V, sent through U's anonym $A_{U,i}$ (Section 6).

Table 2: Metal client's user-facing API functions.

We now provide an example about how two users Alice and Bob use Metal's API to store and share files. First, Alice and Bob each create an account using the CreateAccount function. Alice can then invoke ReadFile or WriteFile to read or write her files. Now, suppose that she wants to share a file with Bob.

To receive the file from Alice, Bob uses NewAnonym to generate a new anonym A_{Bob} and sends it to Alice via some out-of-band communication (as discussed in Section 2.2). After Alice receives this anonym, she grants Bob read access to one of her files by calling the SendCapability function, which produces such a capability and sends it to Bob.

Bob then uses the ReceiveCapability function to receive the capability for this file from the servers, in which Bob knows the file is sent through A_{Bob}. Since Bob can have many anonyms and Bob only gives A_{Bob} to Alice, Bob knows that the file is from Alice, assuming that her client is not compromised.

2.2 Threat model

Metal uses the following threat model: the attacker can compromise any set of users in a malicious way and one of the two servers in a semi-honest way, while the other server is not compromised. We assume that each user establishes secure connections with each server (such as TLS).

Metal makes two assumptions on communication:

- ***Anonymity network.*** Achieving anonymity requires users to hide their IP addresses. Metal assumes that each user uses an anonymity tool to contact the servers. Many such tools exist, providing varying degrees of anonymity, such as Tor [DMS04], secure messaging [CBM15; HLZZ15; AS16; TGL+17; KCDF17; PHE+17; ACLS18; KLD20], and a trusted VPN proxy.
- ***Out-of-band communication.*** Before sharing a file, a user must first know the recipient's identity (an anonym in Metal); otherwise, the user does not even know who should receive the file. Exchanging the anonym requires some out-of-band communication between the two users, which is similar to a Bitcoin user's telling another user its wallet address. The users can meet in person or use secure messaging. Metal strives to minimize the use of such out-of-band communication: every two users only need to use this channel to exchange their Metal anonyms *once*, and the subsequent file-sharing activities will be performed within Metal.

2.3 Security guarantees

We now describe Metal's security guarantees informally. We consider a set of malicious users $\mathsf{MalUsers}$ who collude with one another and with one of the servers, and consider an honest user U who can access file F. The malicious users can interact with the honest users, including sharing files with them. For file access, Metal provides the following guarantees:

(a) **Anonymity:** Neither the servers nor anyone in $\mathsf{MalUsers}$ can distinguish the honest user U from other honest users.

(b) **File secrecy and integrity:** If user U has never granted anyone in $\mathsf{MalUsers}$ read or write access to F, $\mathsf{MalUsers}$ learn nothing about F or cannot modify F, respectively.

(c) **Read obliviousness:** Neither $\mathsf{MalUsers}$ nor the servers know which file was read by user U, even if $\mathsf{MalUsers}$ have read/write capability to all files. That is, $\mathsf{MalUsers}$ cannot distinguish a read operation from a completely different read, by another honest user, to another file.

(d) **Write obliviousness:** If U never gave anyone in $\mathsf{MalUsers}$ the read capability to F, neither the servers nor anyone in $\mathsf{MalUsers}$ realizes that file F has changed. If someone in $\mathsf{MalUsers}$ has read capability, they legitimately learn that the file is changed, but they do not learn who changed it if more than one honest user has write permission to F.

(e) **Read/write indistinguishability:** Neither the servers nor anyone in $\mathsf{MalUsers}$ knows whether an honest user's file access request is read or write, if none of $\mathsf{MalUsers}$ has read capability to that accessed file.

For file sharing, consider another user V who wants to share file F with U, and U owns two anonyms $A_{U,i}$ and $A_{U,j}$ where $i \neq j$.

(f) **Capability sharing secrecy:** If user V sends a file F's capability to U via $A_{U,i}$, Metal does not reveal to the servers or other users (besides U and V) the following: U, V, F, $A_{U,i}$, or that U and V have access to F.

(g) **Anonym unlinkability:** Neither servers nor anyone in $\mathsf{MalUsers}$ can link $A_{U,i}$ and $A_{U,j}$ unless U reveals this linkage to compromised users.

(h) **Anonym authenticity:** If user U gives anonym $A_{U,i}$ to someone in $\mathsf{MalUsers}$, users in $\mathsf{MalUsers}$ cannot send files to the other anonym $A_{U,j}$ that $\mathsf{MalUsers}$ do not know.

Formalism and proofs roadmap. Metal achieves the guarantees above based on common cryptographic assumptions. In Appendix A, we provide a simulation-based security definition and proof for Metal-ORAM. In Section 4 and Section 6.1, we provide proof sketches for Metal-AC and Metal-SHARE; understanding security for them is easier than for Metal-ORAM, so we delegate their proofs to an extended paper.

Non-guarantees. Metal does not hide when the user calls the API (timing) or which function the user is calling (e.g., sharing a permission vs. reading a file). These two leakages can be hidden by padding in time and in computation, which are easy to add to Metal (at an extra cost), as we discuss in Section 8. Metal does not protect against denial-of-service attacks by a server, and Metal does not protect against either server being malicious (beyond semi-honest), which we leave to future work.

3 The layout of S2PC in Metal

In this section we present the layout of the secure two-party computation (S2PC) that the two servers run in Metal. Metal's S2PC takes a specific form, within which we will plug in Metal's techniques. To instantiate the secure computation, Metal uses Yao's protocol [Yao86; GMW87; BMR90; KS08; BHKR13; ZRE15] in a reactive manner.

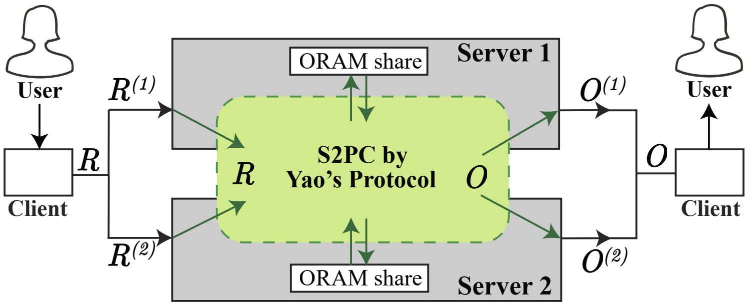

Figure 3: Metal's two servers run Yao's protocol to take user input and access ORAM storage.

Client sending a request. As Figure 3 illustrates, a client sends its request R (e.g., which file to access) to the two servers *secret-shared* (e.g., using XOR secret-sharing) into $R^{(1)}$ and $R^{(2)}$. In this way, no server sees the request in the clear. Server i will receive $R^{(i)}$. Inside S2PC, the servers combine the two shares $R^{(1)}$ and $R^{(2)}$ to create R.

Metal's servers also secret-share the ORAM store such that neither server knows the data stored in the ORAM. But, if they want to access some parts of the ORAM, they take their local shares of those parts as input and reconstruct those parts inside the S2PC. The two servers then update the ORAM store by outputting the updated shares from the S2PC.

Yao's protocol. Our S2PC is based on Yao's garbled circuits protocol. We present Yao's protocol as a black box here and in the form relevant to our S2PC. Yao's protocol enables two parties (here, the two Metal servers) to jointly compute a function over their own secret inputs without leaking the secret inputs to each other. Concretely, suppose that Server 1 has secret input $x^{(1)}$ and Server 2 has secret input $x^{(2)}$, they can compute a function $f(x^{(1)}, x^{(2)}) \to (y^{(1)}, y^{(2)})$ such that Server i learns only its own input $x^{(i)}$ and function output $y^{(i)}$, and nothing else about the other party's input or function output. To supply a random tape for the function, each server independently samples a share of the random tape, takes it as input to S2PC, and reconstructs the random tape by XORing the two shares inside the S2PC. By doing so, one of the two servers does not know the random tape.

In Metal, $x^{(i)}$ will consist of $R^{(i)}$, the ORAM share stored by Server i, and some other state. The function f processes the user's request by checking the capability and running ORAM client operations or file-sharing operations. The result of f is y, which consists of the response O to the client (as Figure 3 shows), an update to the ORAM, and other changes to the servers' state. The S2PC outputs O to the client in secret shares $O^{(1)}$ and $O^{(2)}$, where each server has one share.

Client receiving a response. The servers send the two shares to the client, who can put them together and obtain output O.

Metal's S2PC uses Yao's protocol in a *stateful and reactive* manner like some works in S2PC [GKK+12;

LO13; WCS15; NR16; ZWR+16; DS17]. That is, it does not compute just one function f within S2PC, but instead it runs a sequence of functions $\{f_1, f_2, ...\}$ continuously—this sequence of functions can keep state, take new inputs, reveal some outputs midway, and continue processing in this manner for many steps. This reactive property captures the fact that the servers offer a service, not only a one-time computation. The stateful nature is needed to maintain IndexORAM state.

4 Metal-AC: Anonymous access control

Metal's first component, Metal-AC, checks whether the user has permission to complete the request. Since it is the simplest of our three components, we present it first as a warm-up.

One natural design for Metal-AC is to store access control lists (ACLs) on the servers. However, materializing ACLs is expensive—to access ACLs obliviously, each file's ACL must first be padded to the size linear to the number of users, then be accessed by ORAM.

Instead, in Metal, each client on a user's machine stores the user's *capabilities*, which represent a user's permission to read or write a file and are reminiscent of operating systems' capabilities [DH66]. A user needs to present a capability (in secret shares) to the servers before accessing or sharing a file.

Metal uses authenticated encryption, which provides confidentiality and unforgeability, to implement capabilities. The two servers verify a capability by jointly decrypting the capability inside the S2PC. In Metal, a capability is a ciphertext of the access description under a key that is secret-shared between the two servers. For example, a capability to read and write file F has a description "File ID_F: R+W":

$$C_F^{\text{R+W}} := \mathsf{AuthEnc}(\mathsf{capability\_key},\ \text{“File } \mathsf{ID}_F\text{: R+W”})\ .$$

Each server stores a share of the capability key, which is used for all users' capabilities. The servers grant and verify the capabilities inside the S2PC through secret shares, and thus one server cannot see the capabilities or the capability key.

Granting a capability. The servers, in S2PC, grant a capability to a user in the following two situations:

- When the user **creates an account** (by calling the CreateAccount function), the servers, in S2PC, reserve a continuous range of ℓ_{file} file identifiers for this user, who obtains a *multi-file capability* for reading and writing any of these ℓ_{file} files. Later, the user operates on these ℓ_{file} files. The user can use this capability to share the files.
- When the user **receives a capability of a file that another user shares with this user** (by calling the ReceiveCapability function), the user obtains a capability for this file, which is generated by the servers during the other user's invocation of SendCapability (described in Section 6). The user can use this capability to access the file but not to share the file.

To grant a capability, the S2PC between the servers decides an access description (e.g., file F with permission P_F) and proceeds as follows: the S2PC reconstructs the capability key, computes the capability, and returns the capability in the form of secret shares to the user's client, as described in Figure 4.

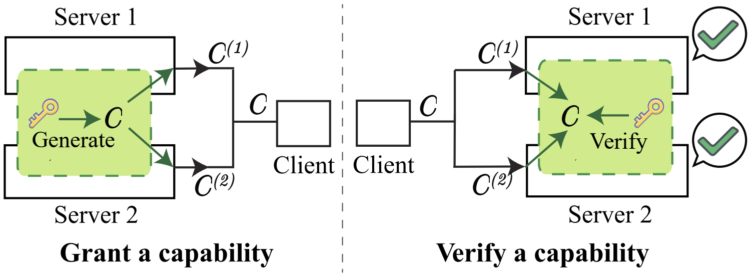

Figure 4: Metal-AC grants or verifies a capability C using the capability key, which is secret-shared between the two Metal servers.

Verifying a capability. Since users can be malicious, each user needs to present a capability to the servers before accessing some file. To start with, the user's client splits the capability into two secret shares and provides one to each server. Since each time the client uses fresh randomness for secret sharing, the servers do not know if the same capability is used again. Then, as Figure 4 shows, the S2PC uses the capability key to decrypt the capability.

If the access description is valid for the operation the user wants to perform, Metal-AC invokes Metal-ORAM or Metal-SHARE as in Figure 2 and provides the access description to the corresponding component inside S2PC.

Security proof sketch. Metal-AC uses authenticated encryption to hide the information inside the capability from the user, which avoids the leakage of the file owner due to file identifiers' being reserved in owner-specific continuous ranges during the account creation, and to prevent a malicious user from forging a capability. Metal-AC uses S2PC to distribute the access to the capability key, so that one server cannot grant a capability or see what is inside the capability. By using secret shares to exchange the capability between the client and the S2PC, anyone of the servers does not even see the capability.

In relation to the security guarantees we described in Section 2.3, Metal-AC ensures anonymity since none of the two servers can see what the capability is, and the user does not have the capability key; later in Metal-ORAM (Section 5.3), we can see that Metal-AC helps us achieve file secrecy and file integrity by allowing only those users with the valid capability to access that file. Metal-AC does not leak the capability as well as the access description inside the capability, which helps achieve obliviousness and read/write indistinguishability.

5 Metal-ORAM: Efficient two-server multi-user ORAM for file storage

In this section we describe how the two Metal servers store and obliviously access user files using Metal-ORAM. We first provide some background about ORAM as well as the construction of Primitive Metal and its limitation. Then, we describe Metal's synchronized ORAM trees as well as the tracking and permutation generation techniques, which overcome this limitation. We prove the security of Metal-ORAM in Appendix A.

5.1 Background on ORAM

Metal-ORAM wants to use ORAM for this scenario: the two servers running a S2PC procedure store an array of files D in the S2PC state, and they want to access the x-th file $D[x]$ inside the S2PC, without any server knowing the secret location x. ORAM for S2PC [GKK+12; LO13; WCS15; ZWR+16; DS17] is a cryptographic primitive that enables such oblivious data access in S2PC.

We identified Circuit ORAM [WCS15] to be appropriate for our setting: the ORAM client has competitive performance, which is polylogarithmic to the number of files even in the worst case, while other schemes such as SqrtORAM [ZWR+16] and Floram [DS17] have a linear worst-case complexity. Circuit ORAM has the benefit that the user waiting time remains acceptable even in the worst case.

We now provide necessary background about Circuit ORAM for the reader to understand how Metal uses it. Circuit ORAM stores such a file array D in a binary tree. To store N files, Circuit ORAM uses a tree with height $h = \lceil \log_2 N \rceil$, as Figure 5 shows. Each tree node can store three fixed-size *blocks*. In addition to tree nodes, a stash temporarily stores some blocks that have not been added to the tree, up to the stash size bound.

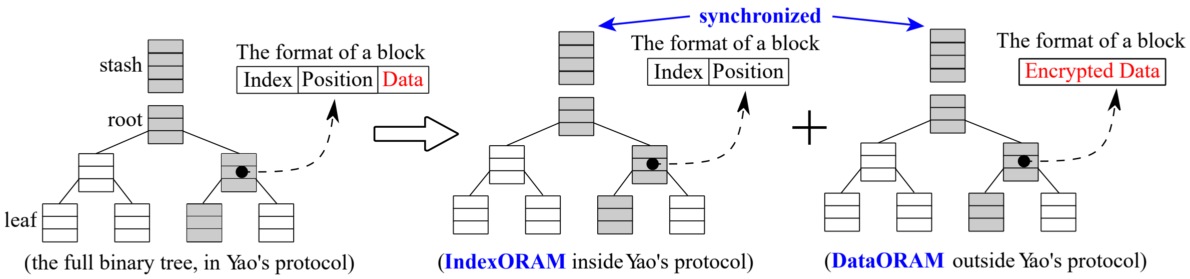

Figure 5: Metal-ORAM moves data out of Yao's protocol (Section 5.3). The data is too large to be processed efficiently in Yao's protocol.

Each block either is empty or stores the data of a file $D[x]$. Such a block consists of the index x, the data $D[x]$, and its position—the root-to-leaf tree path where this block resides. If a block is currently buffered in the stash, the block stores which tree path the block will be evicted onto.

To locate a block in this tree, Circuit ORAM keeps a position map, which maps each index $x \in \{1, 2, ..., N\}$ to the path on which the block resides if the block is not buffered in the stash. The position of $D[x]$ in the position map should match the position field in the block that contains $D[x]$.

Reading. To read a file, the two servers in the S2PC first look up the file's position in the position map, then look up the block by a linear search of both the stash and the path corresponding to this position. The two servers can then read the data in the block in S2PC. After reading the file, the two servers assign a new random position to this block, put the block into the stash, and update the position map accordingly.

Writing. To write a file, the two servers follow the similar steps, but when they put the block back into the stash, the two servers replace the data with the new data from the user. Note that accesses to the position map also need to be oblivious. Metal uses the standard recursive technique [SCSL11; SSS12; SDS+13] to store the position map in ORAM.

Stash eviction. After each read and write, Circuit ORAM needs to perform a stash eviction, in which some blocks buffered in the stash are evicted into the tree to ensure that the stash does not overflow. For each eviction, Circuit ORAM chooses two paths of the tree [GGH+13] and rearranges the blocks in the stash and the blocks on these two paths. This rearrangement is the heaviest step and involves a lot of technical details less relevant to describe here but included in [WCS15].

5.2 Primitive Metal

We now have enough background to describe Metal's primitive scheme, which serves as the foundation for our subsequent improvements. Recall that Metal's primitive scheme already provides the desired security guarantees, though it is slow. First, Circuit ORAM immediately gives us a way to achieve *read/write obliviousness* and *read/write indistinguishability*. Recall that the two servers now can run a function f that requests file data $D[x]$ as input from ORAM; none of the servers knows the index x or the data $D[x]$. The two servers also do not know whether the function f reads or writes the data because we pad the computation in function f; this padding overhead is small because reading and writing are similar in ORAM. Second, we obtain *anonymity* and *file secrecy/integrity* with the help of Metal-AC (Section 4).

We now outline this primitive scheme. In this scheme, all users' files are arranged in a file array D stored in the ORAM inside the S2PC and are padded to have the same size (e.g., 64 KB). A user who wants to read or write a file $D[x]$ first presents a capability to pass the check in Metal-AC. If the user passes the capability check, the two servers access the ORAM on the user's behalf. The two servers return the file data back to the user via secret shares, as described in Section 3. Note that if the user writes to a file rather than reads, the user receives dummy data in the response, as a result of padding.

Yet, Primitive Metal is slow: our experiments (Section 7.7) show that it takes ≥ 75 s to read a file in a store of 2^{20} 64 KB files. In addition, storing data in S2PC is quite expensive because every bit in S2PC is represented as a pair of garbled circuit labels, resulting in a storage overhead of at least $256\times$.

The bottleneck of Primitive Metal. The primitive scheme is slow due to the data-intensive operations inside Yao's protocol. Recall that Yao's protocol builds on garbled circuits; processing a large amount of data leads to many garbled gates being generated, transmitted, and evaluated, resulting in heavy computation and communication. In particular, the primitive scheme is (1) reading all the blocks in the stash and on an ORAM path and (2) rearranging all the blocks in the stash and on two ORAM paths during stash eviction.

Metal-ORAM avoids this bottleneck by moving all the file data out of Yao's protocol and processing such data with more efficient, customized protocols.

5.3 Moving data out of Yao's protocol: Metal's synchronized inside-outside ORAM trees

To avoid the primitive's limitation, Metal-ORAM splits the ORAM binary tree into two synchronized ORAM stores, IndexORAM and DataORAM, as illustrated in Figure 5. IndexORAM only contains small metadata with no file data, which is the only data structure that will be accessed ***inside*** Yao's protocol. DataORAM stores the file data ***outside*** Yao's protocol, not accessed by Yao's protocol.

IndexORAM. We use the recursive technique [SCSL11; SSS12; SDS+13] to store the position map inside recursively larger ORAM trees; hence, IndexORAM is a *set of trees* of increasing sizes. This set of trees enables looking up the position in the last tree for a given file x. However, to simplify matters for clarity, we illustrate only the last tree in Figure 5, and we will refer to a single IndexORAM tree in the rest of the protocol description, with the understanding that Metal-ORAM is handling the logistics of the other smaller trees as well.

DataORAM. DataORAM—as Figure 5 shows—resembles IndexORAM's last tree but only stores file data. DataORAM stores the data in the form of ElGamal ciphertexts [ElG84; Ber06; Ham15; Ristretto] under a *global* public key; each server has a share of the corresponding private key. Using the properties of ElGamal, the two servers can rerandomize the ciphertexts without knowing the private keys and can work together to decrypt ciphertexts as needed, which we will leverage in the construction of our protocols. In Metal, the DataORAM tree is stored on Server 1's disk.

Synchronization. Though we split the tree into two structures, we ensure that these two trees are ***synchronized*** in that the data of a file is at the *same* location in DataORAM as its index/position is in IndexORAM.

We now describe how to read and write a file with these synchronized inside-outside ORAM trees.

Reading. To read file $D[x]$, the two servers first find the file index in IndexORAM and retrieve the position p of the file using Primitive Metal's approach. After doing so, the S2PC procedure determines which block on the path stores the file, i.e., the i-th block on the path p is the block for $D[x]$. Due to the synchrony between IndexORAM and DataORAM, as Figure 5 shows, the encrypted data of $D[x]$ can be found also in the i-th block of the same path p in DataORAM.

However, we cannot simply have the two servers fetch the i-th block in DataORAM: while the servers can see the path p due to ORAM's guarantees, they should not see i. The location i is related to the block history [RAC16], and revealing i to the servers breaks obliviousness. Therefore, Metal-ORAM combines threshold decryption and our *secret-shared doubly oblivious transfer* protocol (Section 5.4) in a way that the user receives the decryption of the i-th block on path p in DataORAM, i.e., the file data $D[x]$, but neither server learns i or $D[x]$.

After reading a file, the two servers need to perform the ORAM management routines: they put the index block into the stash of IndexORAM and put the data block into the stash of DataORAM. These blocks will later be evicted into the tree. We will describe the details of the reading protocol in Section 5.4.

Writing. To write a file $D[x]$, the two servers run the protocol in a similar manner, but we want to ensure that (1) the user with write permission does not see the file contents (since such a user might not have the read permission) and (2) the user-provided data is inserted into DataORAM.

Thus, the writing protocol makes the following changes: First, instead of reading the i-th block in the array, the protocol reads a dummy block, which contains empty file data; therefore, a user with only write capability does not see any file data in this operation. Second, when the two servers insert a data block back into the DataORAM's stash, the two servers instead write the user-provided block into the stash. The user-provided block is created in the following manner: the user secret-shares the file contents between the servers, each server encrypts one share, and the servers combine the two encrypted shares.

To make reading and writing indistinguishable, we merge their protocols as one protocol such that the servers are running the same protocol for reading or writing, with little overhead, as we will show in Section 5.4 and Section 5.5. This merged protocol does not reveal whether it is reading or writing to one of the servers, and the protocol still preserves file secrecy and integrity: a user with read capability cannot modify the file, and a user with write capability does not see the file data.

Stash eviction. The last aspect we need to take care of is stash eviction, which is needed after every read or write. The stash eviction rearranges some blocks in the tree. The challenge is that if Metal-ORAM only evicts the stash in IndexORAM, the synchrony between IndexORAM and DataORAM breaks.

Metal-ORAM remedies the synchrony by "somehow" capturing the rearrangement that happens to IndexORAM and also applying it to DataORAM. We cannot simply reveal the rearrangement to the servers since doing so breaks the ORAM obliviousness. Instead, Metal-ORAM provides a technique for *secure tracking and permutation generation*, described in Section 5.6, to convert Circuit ORAM's rearrangement into a permutation. Then, Metal-ORAM employs a distributed permutation protocol to apply the rearrangement to DataORAM, such that the two ORAM trees are re-synchronized.

5.4 Fetching blocks in DataORAM

We now describe how to fetch the data block in DataORAM without revealing the location i to the two servers. Circuit ORAM allows the two servers to learn which path the block is assigned to, so the two servers' task is to fetch the data block from among the $|\mathsf{stash}|$ blocks in the stash and the $(3 \times h)$ blocks on the path. Let $\vec{m}$ be the array of $N = (|\mathsf{stash}| + 3 \times h)$ blocks that these blocks form. The S2PC knows the location i; it secret-shares i between the two servers, such that Server 1 knows $i^{(1)}$, and Server 2 knows $i^{(2)}$. Below, we describe our *secret-shared doubly oblivious transfer* (SS-DOT) protocol, at the end of which Server 2 receives the i-th (encrypted) block in the array, without any server learning what i is. The fetched block is encrypted under ElGamal, and the decryption key is secret-shared between the two servers, so the two servers can run an existing threshold decryption protocol [BHKP16] and return the file contents to the user in a secret-shared form.

We then add read/write indistinguishability to this fetching operation. Recall that if a user only has write capability, the user should not see the file's data. To ensure such file secrecy as well as to make read/write indistinguishable, the two servers add a dummy block that does not contain any file data at the end of the array. The two servers now search from an array of $(|\mathsf{stash}| + 3 \times h + 1)$ blocks. If the user writes to a file $D[x]$, the S2PC secret-shares $i = (|\mathsf{stash}| + 3 \times h + 1)$ instead, such that the two servers fetch the dummy block, and the user sees only dummy data. This dummy block is unused when the user is reading a file; it merely stays in the array for padding.

Secret-shared doubly oblivious transfer. To fetch the i-th block, Metal uses the following customized protocol. Recall that each server has a share of i: $i^{(1)}$ and $i^{(2)}$, respectively. Server 1 has an array of file data blocks $\vec{m} = \{m_1, m_2, \ldots, m_N\}$. In our protocol, $N = (|\mathsf{stash}| + 3 \times h + 1)$, and Server 1 needs to rerandomize the blocks read from DataORAM, using the functionality of ElGamal encryption, before executing the SS-DOT protocol.[1] This protocol has Server 2 obtain the i-th block without either server learning i.

Oblivious transfer (OT) [Rab81; EGL85] does not suffice for our task because in OT one server knows the index. Doubly oblivious transfer [MRCK19] does not suffice either because it does not support two-party secret sharing and focuses on one-out-of-two transfer instead of one-out-of-N.

There are many ways to implement this simple functionality in S2PC, so we do not claim much novelty for this procedure. Yet what is important for us is to find a way that is efficient for our setting because this

[1] A trick to implement this rerandomization efficiently is to observe that Server 2 only sees one of these N blocks, and thus one can rerandomize these N blocks using the same randomness, which saves a lot of computation.

operation runs for every file access. We develop a simple and efficient procedure as follows:

1. The two servers $\mathcal{S}_1$ and $\mathcal{S}_2$, inside S2PC, reconstruct i from its shares $i^{(1)}$ and $i^{(2)}$ and generate N keys $\{k_1, \dots, k_N\}$ such that $\mathcal{S}_1$ receives as output all these keys, and $\mathcal{S}_2$ receives only k_i.
2. For each $j \in \{1, ..., N\}$, $\mathcal{S}_1$ uses k_j to symmetrically encrypt 0 and m_j to obtain ciphertexts z_j and c_j, respectively, with authenticated encryption. $\mathcal{S}_1$ shuffles all the (z_j, c_j) pairs and sends them to $\mathcal{S}_2$.
3. $\mathcal{S}_2$ uses k_i to decrypt the first ciphertext of each pair: only one, say z_j, will decrypt to 0. It then decrypts the corresponding c_j, obtaining m_i. Note that j is independent of i because of Server 1's shuffle.

This procedure has the advantages that the computation in S2PC is independent of the length of m_i and that the messages m_i are symmetrically encrypted, which has small ciphertext expansion and efficient encryption/decryption.

Security proof sketch. The security of SS-DOT, i.e., obliviousness for both parties, is a direct result of the security of S2PC and authenticated encryption.

5.5 Putting a block into DataORAM's stash

The next step is to put a block into the DataORAM's stash, which will be later evicted to the tree (Section 5.6). Recall that if the user is reading a file, Metal-ORAM should put back the file's current data block, and if the user is writing to a file, Metal-ORAM should insert the user-provided data block. This distinction is crucial for file integrity because we want to avoid a malicious user who only has the read capability to tamper with the file by changing the block.

Metal-ORAM implements this operation by a permutation. Suppose that we place in an array the following: the blocks in the stash, the block read during the fetching (Section 5.4), and the user-provided block in an array. The array therefore has $(|\mathsf{stash}| + 2)$ blocks. Suppose that the S2PC finds that the k-th block of the stash is vacant. If the user is reading the file, S2PC can generate a permutation σ_{read} that exchanges the k-th block with the $(|\mathsf{stash}| + 1)$-th block. If the user is writing the file, S2PC generates σ_{write} that exchanges the k-th block with the $(|\mathsf{stash}| + 2)$-th block instead. By doing so, the correct block is inserted into the stash (i.e., the first $|\mathsf{stash}|$ blocks of the permuted array). The servers then discard the last two blocks.

The challenge is to obliviously perform this permutation: neither server should learn k because leaking k breaks the ORAM obliviousness, and neither server should know which permutation, σ_{read} or σ_{write}, is performed because we want read/write indistinguishability.

Metal-ORAM *distributes this permutation* in a way that hides the permutation. Inside the S2PC, Metal-ORAM secret-shares the permutation into two permutations $\sigma^{(1)}$ and $\sigma^{(2)}$ between the two servers where the composition of $\sigma^{(1)}$ and $\sigma^{(2)}$ equals σ_{read} or σ_{write}. The two servers rerandomize the blocks and apply the permutations in turn; the result is the same as when applying σ_{read} or σ_{write} directly. Formally,

1. The two servers $\mathcal{S}_1$ and $\mathcal{S}_2$, inside S2PC, sample a random permutation $\sigma^{(1)}$ and compute $\sigma^{(2)} = \sigma \circ (\sigma^{(1)})^{-1}$ where $\circ$ denotes composition of permutations and $(\sigma)^{-1}$ denotes the inversion such that $(\sigma)^{-1} \circ \sigma$ is the identity permutation.
2. $\mathcal{S}_1$ rerandomizes the ciphertexts of the $(|\mathsf{stash}| + 2)$ blocks above, applies the permutation $\sigma^{(1)}$, and sends the permuted blocks to $\mathcal{S}_2$.
3. $\mathcal{S}_2$ receives the blocks from $\mathcal{S}_1$, rerandomizes the ciphertexts of the blocks, applies the permutation $\sigma^{(2)}$, and sends them back to $\mathcal{S}_1$.
4. $\mathcal{S}_1$ receives the blocks from $\mathcal{S}_2$ and stores the blocks in the corresponding locations in DataORAM.

This method has been used in a similar manner in SqrtORAM [ZWR+16] to obliviously reorganize the data blocks.

5.6 Resynchronizing after eviction by tracking and permutation generation

After each access to the ORAM, Metal needs to evict the stash. We can run the Circuit ORAM's stash eviction algorithm inside S2PC to update IndexORAM, which rearranges the index blocks, but it breaks the synchrony with DataORAM: a file's index block in IndexORAM is now at a new location, but the data block in DataORAM is still at the previous location.

Metal-ORAM's solution is to extract how blocks in IndexORAM move during the stash eviction and apply the same movement to DataORAM. The challenge is to implement this efficiently. A prior scheme, Onion ORAM [DDF+16], uses private information retrieval for this purpose, but it requires a large number of data block operations that is quadratic to the number of blocks being moved, which is heavy.

Metal-ORAM instead develops an algorithm to *track* the changes to IndexORAM inside the S2PC and to *convert them* into a permutation. Then, Metal-ORAM uses the distributed permutation in Section 5.5 to rearrange the blocks in DataORAM.[2]

We demonstrate the tracking and permutation generation process in Figure 6 and provide the algorithm in Figure 7. We now explain the algorithm at a high level.

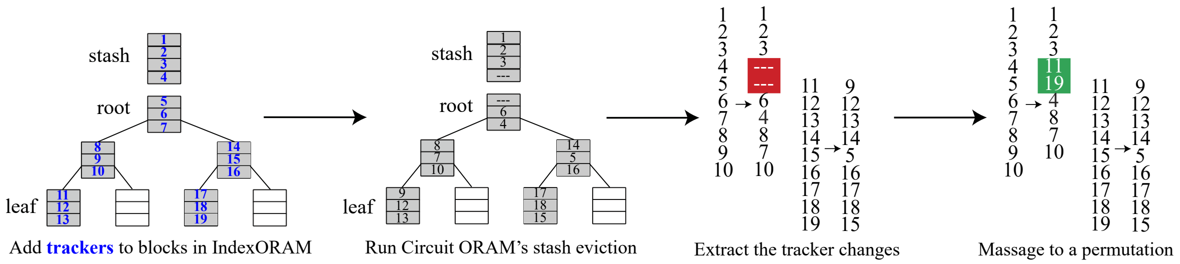
stash
root
leaf
stash
root
leaf
Add **trackers** to blocks in IndexORAM
Run Circuit ORAM's stash eviction
Extract the tracker changes
Massage to a permutation

Figure 6: Metal's tracking and permutation generation (Section 5.6).

Tracking. Before the stash eviction, as Figure 6 shows, Metal-ORAM attaches some "trackers" to the index blocks inside S2PC. Next, it performs the stash eviction per the ORAM algorithm and then observes how the trackers moved. In Circuit ORAM's stash eviction, some trackers disappear because the blocks that they were attached to were deleted (indicated by '---' in Figure 6). Thus, the list of trackers directly pulled from IndexORAM after the eviction is incomplete (i.e., with empty slots), as shown in Figure 6's red area. We give the details of tracking in Figure 7.

Permutation generation. These missing trackers prevent us from creating a permutation directly. Hence, Metal-ORAM brings back those numbers to the empty slots, as Figure 6's green area highlights. The resultant list of trackers becomes a permutation subsuming the changes in IndexORAM. Metal-ORAM feeds this permutation into the distributed permutation described in Section 5.5 to apply the permutation to DataORAM. Thus, the IndexORAM and DataORAM become resynchronized, as desired, shown in Figure 7.

We have described how stash eviction works in our synchronized inside-outside ORAM trees. In our implementation, Metal-ORAM combines the permutation in Section 5.5 with the resynchronizing permutation

[2] The permutation does not explicitly remove the previous version's data block from DataORAM: since the index of the previous version has been deleted in IndexORAM, that data block becomes inaccessible (treated as dummy) in our construction. The previous version's data block may be indirectly discarded later, as a result of a permutation process shown in Section 5.5.

Before Circuit ORAM's stash eviction:

- Recall that Circuit ORAM evicts blocks onto two paths. The algorithm appends a number from 1 to $(|\mathsf{stash}| + 6 \times h - 3)$ (called *trackers*) to each block on the two paths in IndexORAM inside S2PC using the following order, as Figure 6 shows:
  1. the blocks on the first path, from stash to leaf
  2. the *unnumbered* blocks on the second path, from stash to leaf

After Circuit ORAM's stash eviction:

- The algorithm "peels off" the tracker numbers from the two paths in order and constructs an array. Some numbers will be missing (indicated by "---" in Figure 6).
- The algorithm uses linear scanning to find numbers in $\{1, 2, ..., |\mathsf{stash}| + 6 \times h - 3\}$ that have not appeared in the array (e.g., 11 and 19 as in Figure 6).
- The algorithm uses linear scanning to find the "---" slots in the array and fills into the slots the unused numbers. (The order is unimportant, but our algorithm starts with the smaller ones.)

Figure 7: Algorithms for tracking and permutation generation.

inside the S2PC, so every file access only uses one distributed permutation.

6 Metal-SHARE: Unlinkable capability sharing

We have achieved oblivious file storage, but we have not yet shown how a user shares files with other users. In this section we describe Metal-SHARE, which contributes the functionality of file sharing without introducing metadata leakage. We first describe two central notions of Metal-SHARE, anonyms and capability broadcast list, and then describe the sharing protocol.

Anonyms. Anonyms are anonymous identities that a user can leverage in file sharing. An anonym is similar to an email address for receiving emails or a Bitcoin wallet address for receiving Bitcoin; but, our usage of anonyms has the additional benefit that it hides the anonym owner's identity such that two anonyms of the same user cannot be linked to each other. Every user in Metal can locally create many anonyms without interactions with the servers. A user then gives his/her anonyms to others in order to receive file capabilities from them. For example, a user U who wants to receive (many) files from user V in the future can provide V with one of the anonyms, $A_{U,i}$ (where i is the anonym index), as Figure 8 shows.

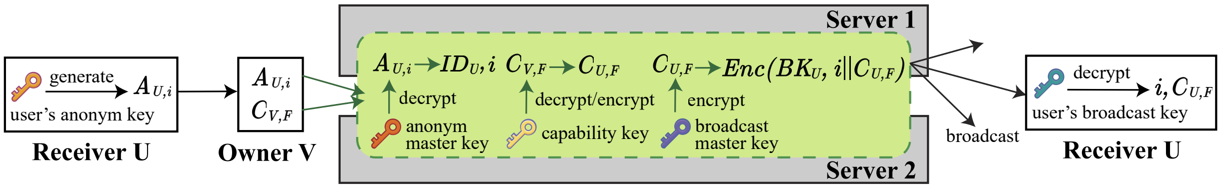

Figure 8: Sending and receiving a file's capability in Metal-SHARE.

Capability broadcast. When a user V sends a capability to U, V puts the capability on the servers so that the receiver U can later retrieve it from the servers. This allows U and V to share files even if they will never be online at the same time.

To avoid leaking metadata through network patterns of a user, Metal-SHARE has Server 1 publish a list of encrypted capabilities so that each user can download this list. A user can only decrypt those ciphertexts destined to the user. This is similar to a blockchain where all nodes download the blocks, but only some of the blocks contain transactions relevant to the node. In contrast to a blockchain, though, in Metal-SHARE, the blocks are much smaller than in a regular blockchain, and users only need to download and organize the capabilities periodically.

To implement this broadcast, Server 1 keeps a capability broadcast list that contains the encrypted capabilities for users to download. The two servers will encrypt, inside S2PC, each capability under the recipient's *broadcast key*, which the user receives during the account creation from the two servers' S2PC. When a user's client wants to download the list, Server 1 shuffles the capabilities in the list before sending them to the client.

After user V obtains the capability, V can—in the future—use this capability to access the file, without interacting with U. If U wants to revoke the permission, U must discard the old file, create a new file, and share the new file with other users who are supposed to retain the permission.

To prevent the broadcast list from growing monotonically, Metal-SHARE has each encrypted capability in the list to be deleted after a fixed interval (e.g., three days).

Example. We now illustrate how to share a capability. Suppose that user V owns file F and wants to send the read capability of file F to another user U, as Figure 8 shows. The procedure is:

1. **Get the receiver's anonym.** User V obtains one of U's anonyms from U, say $A_{U,i}$, as we discussed in Section 2.2. This anonym can be used for all future file-sharing activities from V to U.
2. **Send a capability.** User V who owns file F requests the servers to grant a capability for reading F to the anonym $A_{U,i}$. The servers check V's capability $C_{V,F}$, create a read capability $C_{U,F}$ for user U, and encrypt $C_{U,F}$ together with i under U's broadcast key, as Figure 8 shows. The ciphertext of $C_{U,F}$ is appended to the capability broadcast list.
3. **Receive a capability.** User U's client periodically downloads the new capability ciphertexts from Server 1's broadcast list and uses U's broadcast key to decrypt each ciphertext. In this manner, it finds capabilities that are destined to U, one of which will be capability $C_{U,F}$ with the anonym index i and the type of permission. U can learn which anonym was used by the sender based on i. If $A_{U,i}$ was only provided to V, U knows that this file is from V.

6.1 Unlinkable anonyms

We now focus on the left part of Figure 8 and discuss how the user U generates a new anonym $A_{U,i}$, how the sender V uses this anonym, and how the S2PC processes the anonym.

Generating the anonym. User U has an *anonym key* AK_U that it received from the servers (via secret shares) during account creation. Using this key, U can generate anonym $A_{U,i}$ for any anonym index i. Informally, the anonym is an encryption of the user ID and the anonym index i.

Sending a capability to this anonym. Another user V who owns file F receives the anonym $A_{U,i}$, as Figure 8 shows. User V has the capability $C_{V,F}$ with full permission and wants to grant the read capability to U. To do that, V calls the server API with the anonym and V's capability, asking the servers to create a qualified capability (in this example, read-only) for U (request sent in secret shares as in Section 3).

Opening the anonym inside the two servers' S2PC. The two servers secret-share *the anonym master key* (AMK), which can decrypt everyone's anonyms. Thus, the two servers can open the anonym inside S2PC, as Figure 8 shows, and continue with the sharing protocol (Section 6.2).

Construction. Metal-SHARE implements anonyms using a special-purpose scheme that builds on Paillier encryption [Pai99], additive secret sharing, and message authentication code (MAC). Our construction is in Figure 9, and we now describe the intuition behind the construction to show the insights.

First, anonyms need to achieve *anonym authenticity*, as defined in Section 2.3. If user U gave anonym $A_{U,i}$ to V, V should not be able to create another anonym $A_{U,i'}$ under a different anonym index $i' \neq i$, assuming that U has never leaked the anonym $A_{U,i'}$ to anyone in the set of colluding malicious users. Our solution is to give user U an anonym key that is derived from the servers' anonym master key, as the Setup and $\mathsf{UserKeyGen}$ algorithms in Figure 9 show. U then needs to append a message authentication code over the pair (ID_U, i), as the $\mathsf{AnonymGen}$ algorithm shows, so that another malicious user cannot forge an anonym for user U.

Second, anonyms must provide *anonym unlinkability*. Hence, we cannot expose ID_U, i, or the MAC to another user because such information may deanonymize U. The natural solution is to use public-key encryption to encrypt $(\mathsf{ID}_U, i, \mathsf{mac})$ in such a way that it can only be recovered in the servers' S2PC. For efficiency, we must move the public-key operations out of S2PC: the two servers will do threshold decryption outside S2PC, and inside S2PC they will merge the decryption results efficiently. There are a few public-key encryption schemes that can make this merging step efficient: Paillier encryption [Pai99], Goldwasser-Micali encryption [GM82], and Brakerski-Gentry-Vaikuntanathan encryption with $\mathbb{Z}_2$ slots [BGV12; SV10; SV14;

```
$\mathsf{Anonym.Setup}(1^\lambda)$:
Run by the servers during the setup to generate the Paillier keys and the anonym master key.
• For $j \in \{1,2\}$, $\mathcal{S}_j$ runs:
  $(\mathsf{sk}_j, \mathsf{pk}_j) \leftarrow \mathsf{Paillier.KeyGen}(1^\lambda)$,
  and publishes $\mathsf{pk}_j$.
• For $j \in \{1,2\}$, $\mathcal{S}_j$ samples a secret share of anonym master key $\mathsf{AMK}^{(j)} \leftarrow\$ \{0,1\}^\lambda$ and stores $\mathsf{AMK}^{(j)}$.
```

```
$\mathsf{Anonym.UserKeyGen}(\mathsf{ID}_U, \mathsf{AMK}^{(1)}, \mathsf{AMK}^{(2)})$
Run by the servers and user $U$ (identified by $\mathsf{ID}_U$) during the account creation to grant the anonym key $AK_U$ to user $U$.
• $\mathcal{S}_1$ and $\mathcal{S}_2$ run a S2PC that takes $\mathsf{ID}_U, \mathsf{AMK}^{(j)}$ as input ($j \in \{1,2\}$) and computes:
  – $\mathsf{AMK} := \mathsf{AMK}^{(1)} \oplus \mathsf{AMK}^{(2)}$.
  – $AK_U := \mathsf{PRF}_{\mathsf{AMK}}(\mathsf{ID}_U)$.
• $\mathcal{S}_1$ and $\mathcal{S}_2$ use the protocol in Section 3 to share $AK_U$ with the user.
• The user stores the anonym key $AK_U$.
```

```
$\mathsf{Anonym.AnonymGen}(\mathsf{ID}_U, i, AK_U, \mathsf{pk}_1, \mathsf{pk}_2)$
Run by the receiver user $U$ to create an anonym with index $i$, using the anonym key $AK_U$.
• $s := \mathsf{ID}_U \parallel i \parallel \mathsf{MAC}_{AK_U}(\mathsf{ID}_U \parallel i)$.
• $U$ additively secret-shares $s$:
  – $s^{(1)} \leftarrow\$ \{0,1,...,2^{|s|+\rho}\}$.
  – $s^{(2)} := s^{(1)} + s$.
• $c^{(j)} \leftarrow \mathsf{Paillier.Enc}_{\mathsf{pk}_j}(s^{(j)})$ for $j \in \{1,2\}$.
• Outputs the anonym $A_{U,i} := (c^{(1)}, c^{(2)})$.
```

```
$\mathsf{Anonym.AnonymRerand}(A_{U,i}, \mathsf{pk}_1, \mathsf{pk}_2)$
Run by the file owner and sender $V$ to rerandomize the receiver's anonym $A_{U,i}$ before sending the anonym (in secret shares) to the servers.
• Lets $A_{U,i} = (c^{(1)}, c^{(2)})$.
• $V$ rerandomizes the anonym:
  – $r \leftarrow\$ \{0,1,...,2^{|s|+2\rho}\}$.
  – $c^{(j)}_{\mathsf{new}} \leftarrow \mathsf{Paillier.AddPlain}_{\mathsf{pk}_j}(c^{(j)}, r)$ for $j \in \{1,2\}$.
• Outputs the anonym $A^{\mathsf{rerand}}_{U,i} := (c^{(1)}_{\mathsf{new}}, c^{(2)}_{\mathsf{new}})$.
```

```
$\mathsf{Anonym.AnonymDecrypt}(A^{\mathsf{rerand}}_{U,i}, \mathsf{sk}_1, \mathsf{sk}_2)$
Run by the servers upon receiving the (rerandomized) anonym from the file owner $V$ to decrypt the anonym in preparation for the capability broadcast.
• $V$ sends $A^{\mathsf{rerand}}_{U,i} = (c^{(1)}_{\mathsf{new}}, c^{(2)}_{\mathsf{new}})$ to the two servers.
• $\mathcal{S}_j$ runs $s^{(j)}_{\mathsf{new}} := \mathsf{Paillier.Dec}_{\mathsf{sk}_j}(c^{(j)}_{\mathsf{new}})$ ($j \in \{1,2\}$).
• $\mathcal{S}_1$ and $\mathcal{S}_2$ run a S2PC that takes $(s^{(j)}_{\mathsf{new}}, \mathsf{AMK}^{(j)})$ as input ($j \in \{1,2\}$), as follows:
  – $\mathsf{AMK} := \mathsf{AMK}^{(1)} \oplus \mathsf{AMK}^{(2)}$.
  – S2PC reconstructs $s$:
    * $s := s^{(2)}_{\mathsf{new}} - s^{(1)}_{\mathsf{new}}$.
    * Lets $s$ be $\mathsf{ID}_U \parallel i \parallel \mathsf{mac}$.
  – $AK_U := \mathsf{PRF}_{\mathsf{AMK}}(\mathsf{ID}_U)$.
  – If $\mathsf{mac} = \mathsf{MAC}_{AK_U}(\mathsf{ID}_U \parallel i)$, $\mathsf{valid} = 1$. Otherwise, $\mathsf{valid} = 0$.
  – Stores $\mathsf{valid}, \mathsf{ID}_U, i$ in the S2PC's state for the use of capability broadcast (Section 6.2).
```

Figure 9: Algorithms of the customized encryption that instantiates anonyms; the use of Paillier encryption follows the common Paillier encryption syntax. Without loss of generality, we assume that the message size of the Paillier encryption set up by security parameter λ is larger than $|s| + 2\rho + 1$ bits where ρ is the statistical security parameter (80 bits in our implementation).

HElib]. We choose Paillier because the size of a ciphertext is small. The AnonymGen and AnonymDecrypt algorithms in Figure 9 show how Metal-SHARE combines additively homomorphic secret sharing and Paillier encryption to instantiate anonyms.

Note that the sender V also needs to refresh the ciphertext such that the two servers do not realize that the refreshed ciphertext is from the same anonym during the threshold decryption. This step avoids the linkage among multiple uses of $A_{U,i}$. The sender V rerandomizes the encrypted secret shares using the additive homomorphism in Paillier encryption, as AnonymRerand in Figure 9 shows.

In this rerandomization step, we adapt a trick from [BPTG15] to rerandomize the anonym. As follows, ρ denotes the statistical security parameter. Before the rerandomization, the distribution of each of $s^{(1)}, s^{(2)}$ is statistically indistinguishable from a uniform distribution in $\{0, 1, ..., 2^{|s|+\rho}\}$ as a result of secret sharing. When we rerandomize the two shares by homomorphically adding $r \leftarrow\$ \{0, 1, ..., 2^{|s|+2\rho}\}$, the distribution of each of the new $s^{(1)}, s^{(2)}$ is now statistically indistinguishable from a uniform distribution in $\{0, 1, ..., 2^{|s|+2\rho}\}$ and is *statistically independent* of the original value of $s^{(1)}$ or $s^{(2)}$, which gives us the following guarantee: even if a user calls SendCapability many times using the same anonym, one of the two servers, knowing the previous rerandomized anonyms, cannot link these anonyms together.

Security proof sketch. Anonym authenticity can be proved by reducing to the unforgeability of MAC. Anonym unlinkability can be reduced to the security properties of Paillier encryption and additive secret sharing. Since our secret sharing has a customized design, we discuss its security as follows:

- **Sharing.** Recall from Figure 9 that the secret s is shared into $s^{(1)} \leftarrow_{\$} \{0, 1, ..., 2^{|s|+\rho}\}$ and $s^{(2)} = s^{(1)} + s$ where ρ is the statistical security parameter. The distribution of each share is statistically indistinguishable from a uniform distribution in $\{0, 1, ..., 2^{|s|+\rho}\}$. Since the two shares are then encrypted under separate Paillier keys, if an attacker only has one of the private keys, the attacker sees only one share, not s.
- **Rerandomizing.** As discussed above, the rerandomization algorithm homomorphically adds r to both shares, and the distribution of each new share (inside the Paillier encryption) is statistically indistinguishable from a uniform distribution in $\{0, 1, ..., 2^{|s|+2\rho}\}$ even with the knowledge of the original share. One of the servers does not learn s or the shares in the original anonym.

6.2 Capability derivation and broadcast

We now focus on the right part of Figure 8. Recall that the two servers secret-share the capability key, as described in Section 4. The two servers can decrypt the capability and recreate a capability with qualified permission, such as read-only.

Then, the S2PC encrypts the new capability $C_{U,F}$, along with the anonym index i, using the broadcast master key in such a way that only user U's broadcast key can decrypt it (the user receives this key during the account creation). The ciphertext is revealed to Server 1, who then appends this ciphertext to the broadcast list. User U downloads the new ciphertexts during the past intervals since U was last online. U uses U's broadcast key to try decrypting each ciphertext, among which U can find $C_{U,F}$, the anonym index i, and the type of permission. With this new capability, U can read the file F using Metal-ORAM.

Discussion on hiding the number of incoming files. The broadcast in Metal-SHARE has an overhead linear to the number of file-sharing operations in the whole system, which is not ideal. The benefit of this broadcast is that it hides the number of incoming files and avoids leaking users' use patterns.

One alternative is to use private information retrieval (PIR) like Pung [AS16; ACLS18]. However, each invocation to Pung can only retrieve a fixed number of data entries. If a user has comparably much more files than other users, this user has to run Pung's protocol multiple times, from which the attacker can still learn this user's use patterns. Another solution is to have users send capabilities to one another via encrypted emails (e.g., PGP [OpenPGP], Autocrypt [Autocrypt], and ClaimChain [KLI+18]), but it does not hide the sharing patterns (the sending of emails).

One seemingly working solution to avoid the linear broadcast is to set a fixed bound N for the number of capabilities that a user can download during an interval T, and a user downloads exactly N capabilities every interval T. In case of insufficient capabilities, the user pads the number to the bound N. If a user cannot retrieve all the capabilities (more than N), the user retrieves the rest of them the next time. Unfortunately, this solution leaks use patterns, as we now discuss.

Consider the following scenario: When users receive file capabilities, they may subsequently perform a few noticeable operations, e.g., adding a new line to the file. If a user has too many incoming capabilities and many of them are deferred to be downloaded the next time in this approach, other users may notice this user's delayed responses to some files and learn that more than N files are sent during an interval. Inevitably, avoiding this leakage requires each user to retrieve all his/her capability ciphertexts as in Metal-SHARE.

Making broadcast efficient. Though capability receiving has to be linear, Metal-SHARE improves the efficiency and makes it practical for the client. In Metal-SHARE, the capabilities are encrypted symmetrically under the user's broadcast key (derived from the broadcast master key, which is secret-shared between the servers). As a result, the encryption, transmission, and decryption costs become small. Concretely, if the broadcast list has 10^4 capabilities, a user only needs to download 1 MB and can decrypt all of them in 10 ms.

7 Performance

In this section we discuss Metal's asymptotic efficiency and concrete efficiency, compare Metal with PIR-MCORAM and AnonRAM-poly, and compare Metal with Primitive Metal to show how Metal's techniques improve the performance.

7.1 Asymptotic efficiency

Since each function (described in Table 2) of Metal may be called independently by the users' clients, we describe each function separately. We first describe the notation we will use.

Notation. We consider that the system supports N_{user} users, $N_{\mathsf{file}} \geq N_{\mathsf{user}}$ files in total, file size D, and N_{anonym} anonyms per user, as well as a broadcast list with N_{list} entries. As follows, we use $O_\lambda(\cdot)$ to express the complexity that hides a polynomial of the security parameter λ, while $N_{\mathsf{user}}, N_{\mathsf{file}}, N_{\mathsf{anonym}}, N_{\mathsf{list}}$ are polynomially bounded by λ. Like [WCS15], we parameterize Circuit ORAM with $\frac{1}{N^{\omega(1)}}$ failure probability (in Metal, 2^{-80}).

Cost for CreateAccount, NewAnonym, and SendCapability. Given that $N_{\mathsf{user}}, N_{\mathsf{file}}, N_{\mathsf{anonym}}$ are polynomially bounded by the security parameter λ, we write their asymptotic efficiency in a way that only depends on λ for simplicity:

- The client's and each server's compute time is $O_\lambda(1)$.
- The server-server communication is $O_\lambda(1)$.
- The client-server communication is $O_\lambda(1)$.

Cost for ReadFile and WriteFile. ReadFile and WriteFile have the computational complexity and communication complexity that are linear to the file size D and polylogarithmic to the number of files N_{file}. Concretely,

- Each server's compute time is $O_\lambda((D + \log^2 N_{\mathsf{file}}) \log N_{\mathsf{file}}) \cdot \omega(1)$.
- The client's compute time is $O_\lambda(D)$.
- The server-server communication (including the communication for SS-DOT and distributed permutation) is $O_\lambda((D + \log^2 N_{\mathsf{file}}) \log N_{\mathsf{file}}) \cdot \omega(1)$.
- The client-server communication is $O_\lambda(D)$.

Cost for ReceiveCapability. ReceiveCapability mainly returns the user the requested part of the broadcast list. At the end of each epoch, Server 1 shuffles the encrypted capabilities, incurring an additional cost. Given that $N_{\mathsf{user}}, N_{\mathsf{file}}, N_{\mathsf{anonym}}$ are polynomially bounded by the security parameter λ, we can write the asymptotic efficiency as follows:

- Server 1's compute time is $O_\lambda(N_{\mathsf{list}} \log N_{\mathsf{list}})$ when shuffling.
- The client's compute time is $O_\lambda(N_{\mathsf{list}})$.
- The client-server communication is $O_\lambda(N_{\mathsf{list}})$.

7.2 Implementation

We implemented Metal in C/C++. We use Obliv-C [ZE15] for Yao's protocol with the improved half-gate garbling scheme for multi-instance security, as described in [GKW+19; GKWY20], which provides 125-bit security. We use Absentminded Crypto Kit [Doe18] for ORAM and OpenSSL for TLS.

API functions	**Time (s)** **without Tor**	**Time (s)** **with Tor**
CreateAccount	0.417 ± 0.001	4.1 ± 0.2
ReadFile / WriteFile	3.62 ± 0.01	8.0 ± 0.2
• client preprocessing	0.056 ± 0.006	
• server accessing IndexORAM	0.134 ± 0.001	
• server encryption	0.007 ± 0.001	
• server fetching	0.126 ± 0.001	
• server eviction	1.526 ± 0.002	
• server threshold decryption	0.038 ± 0.001	
• client postprocessing	0.009 ± 0.001	
NewAnonym	0.03 ± 0.01	
SendCapability	1.13 ± 0.01	4.56 ± 0.19
ReceiveCapability	1.759 ± 0.002	6.2 ± 0.4

Table 3: The latency of user-facing API functions, measured with and without Tor, for a store of 2^{20} 64 KB files. The result is the average of one hundred measurements, with the confidence interval under two-sided Student's t distribution with 90% confidence.

For Metal-AC's authenticated encryption, we use the EAX mode [BRW04], which we deemed to be the most efficient mode for our setting after an extensive search.

The rerandomizable encryption in Metal-ORAM is implemented using ElGamal encryption over Curve25519-Ristretto group [Ber06; Ham15; Ristretto] with a constant-time encoding and a few optimizations. This scheme is about 80× faster than standard ElGamal encryption over a Schnorr group [ElG84] and 200× faster than standard Paillier encryption [Pai99] for our setting.

7.3 Evaluation Setup

Machine configuration. We used two `r4.2xlarge` machines on Amazon EC2 as the servers, one in Northern California, one in Oregon, each with eight CPUs and 61 GB memory. We situated them in different regions to simulate the real-world scenario that the servers are in different trust domains. The user ran in a `t2.xlarge` machine in Canada with four CPUs and 16 GB memory. We allocated the user in Canada to simulate that the user is from a remote location. In our experiment, we measured the latency from this machine. Metal-ORAM's DataORAM is stored on Server 1's Amazon gp2 volume.

Network latency. We measured the round-trip time (RTT) and bandwidth. The inter-server RTT was 19 ms, and the client-server RTT was 70 ms. Measured under AWS's guidelines [AWSBench], the inter-server bandwidth per connection was ≈ 290 MB/s, and the client-server bandwidth was ≈ 17 MB/s.

7.4 Metal's performance

To measure the latency of each operation in Metal, we use a setup with 2^{20} 64 KB files (in total 64 GB of data). To measure the latency of receiving a capability from the servers, we have a user download 10^4 capabilities from Server 1's broadcast list.

We measured the latency of these operations with and without Tor in Table 3. As we remark in Section 2.2, Metal can use other anonymity networks beside Tor. We evaluated on Tor [DMS04] because it is a popular tool. The results without Tor more cleanly show the overhead specific to Metal.

We show the end-to-end benchmark result in Table 3, together with a breakdown of the cryptographic operations in our file access API. From the table, we can see that the latency for each file access is a few seconds. The latencies for creating an account and sending/receiving capabilities are also small.

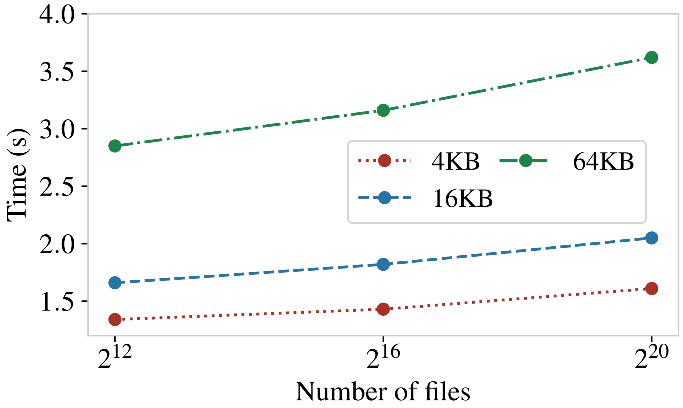

Figure 10: File access latency vs. the number of files / file size. The result is the average of one hundred measurements.

We show the measurements of how the latency of a file access depends on the file size and the number of files. Figure 10 has an exponential x-axis for number of files and a linear y-axis for time. We can see the latency increases linearly to the file size and grows logarithmically to the number of files.

For large-scale measurement, we measured the setup with 2^{20} 1 MB files (where each file is padded to 1 MB). It takes 28.8 ± 0.1 s to access a file. Though Metal works with fixed-sized files, larger files can be segmented into smaller files of fixed size, and clients fetch file segments instead of files as the user needs them. As a result, the cost to access the file then depends on the number of segments.

Metal-ORAM does not support parallel accesses since Circuit ORAM is not parallel. Thus, a user who wants to access a file has to wait for previous accesses to be complete. This restriction has the benefit of strong consistency. But, one who wants to make Metal-ORAM more parallelizable can distribute the computation (described in Section 8) or extend our techniques to parallel ORAM (e.g., Circuit OPRAM [CS17]).

Network I/O and the size of garbled circuits. We measured the inter-server network I/O and the size of garbled circuits (the number of AND gates) in Table 5 for different ORAM implementations. We can see that the network I/O grows almost logarithmically to the number of files. The circuit size, which represents the amount of computation in Yao's protocol, does not grow with the file size.

7.5 Comparison with PIR-MCORAM

As we discussed in Section 9, only PIR-MCORAM [MMRS17] simultaneously provides file sharing and some access patterns protection in the presence of malicious users. Unlike Metal, PIR-MCORAM leaks user identities, but it has the advantage that it only uses a single server. However, this single-server setting results in a latency that is at least linear to the number of files, which becomes slow. Metal's latency, on the other hand, is sublinear to the number of files.

Since PIR-MCORAM [MMRS17] is not open-source, we could not perform an end-to-end evaluation of it. Nevertheless, the evaluation in PIR-MCORAM's paper [MMRS17] provides measurements and discusses the linear behavior of the results (the main cost is in the zero-knowledge proofs). Hence, we can extrapolate the results for amortized and worst-case time from PIR-MCORAM. We did not compare with TAO-MCORAM [SZE+16; MMRS17], another system in the same paper, since TAO-MCORAM uses a trusted proxy, which is not a fair comparison with Metal.

File size	**Amortized time (s)**		**Worst-case time (s)**	
	$\mathbf{2^{16}}$ files	$\mathbf{2^{20}}$ files	$\mathbf{2^{16}}$ files	$\mathbf{2^{20}}$ files
4 KB	≈ 15	≈ 135	≈ 3000	≈ 47600
16 KB	≈ 39	≈ 519	≈ 10900	≈ 190200
64 KB	≈ 135	≈ 2055	≈ 47600	≈ 760700

Table 4: Latencies extrapolated from PIR-MCORAM [MMRS17].

Table 4 shows the results. We can see that when the file size and the number of files are large, PIR-MCORAM has a high latency, especially the worst-case time. For 2^{20} 64 KB files, its amortized time for file access is $\geq 500\times$ Metal's latency (of 3.62 s). Also, PIR-MCORAM leaks user identities, which Metal hides.

7.6 Comparison with AnonRAM-poly

AnonRAM-poly [BHKP16] is another anonymous storage system that also uses the two-server model but does not support file sharing among users. AnonRAM-poly [BHKP16] is not implemented. To estimate a lower bound of its latency we implemented the zero-knowledge proofs for file uploading used in AnonRAM-poly—which the users generate and the servers verify for every access. We implemented them using the disjunctive Schnorr's protocol [Sch89; CDS94; JJ99], and we evaluated its performance under the same evaluation setup as in Section 7.3. Even with multi-threading, such zero-knowledge proofs take ≥ 80 s per file access for a store of 2^{20} 64 KB files. In addition, AnonRAM-poly has other expensive components, such as the zero-knowledge proofs in oblivious PRF and oblivious sorting between the two servers. AnonRAM-poly is therefore at least $\geq 20\times$ slower than Metal.

7.7 Comparison with Primitive Metal

We described Metal's primitive construction in Section 5.2, which directly comes from Yao's protocol and does not use Metal-ORAM techniques to move file data out of Yao's protocol. It provides the desired functionality and security but not efficiency. To demonstrate the poor performance of Primitive Metal, we measured the latency of a single ORAM access of the strawman using three state-of-the-art ORAM schemes [WCS15; ZWR+16; DS17] with the implementation in Absentminded Crypto Kit [Doe18]. Note that these implementations only support in-memory storage, which makes them prone to running out of memory but enjoy faster I/O since Metal stores data on the gp2 storage. Table 5 shows the measurements, which we now discuss alongside with how Metal improves it, on three dimensions:

- **Storage overhead reduction.** Table 5 shows that Primitive Metal soon runs out of memory because the implementation has a storage size blowup of $128\times$. Instead, Metal stores the data outside S2PC and therefore has a smaller blowup.

# Files	Amortized time (s)			Network I/O (MB)			# $\times 10^6$ AND gates			Worst-case time (s)		
File size	2^{12}	2^{16}	2^{20}	2^{12}	2^{16}	2^{20}	2^{12}	2^{16}	2^{20}	2^{12}	2^{16}	2^{20}
(This paper) Metal, as discussed in Section 7.4. This is the end-to-end benchmark including time for Metal-AC.												
4 KB	1.34	1.43	1.61	25.7	39.0	55.2	1.43	2.19	3.16	$\star$	$\star$	$\star$
16 KB	1.66	1.82	2.05	32.0	46.7	64.4	1.43	2.19	3.16	$\star$	$\star$	$\star$
64 KB	2.85	3.16	3.62	60.7	81.4	105	1.43	2.19	3.16	$\star$	$\star$	$\star$
Primitive Metal using Circuit ORAM [WCS15], as discussed in Section 7.7, considering only ORAM access time.												
4 KB	3.56	4.46	†	430	508	†	13.5	16.0	†	$\star$	$\star$	$\star$
16 KB	13.6	16.2	†	1690	1976	†	53.1	62.2	†	$\star$	$\star$	$\star$
64 KB	55.5	66.8	†	6717	7844	†	212	247	†	$\star$	$\star$	$\star$
Primitive Metal using SqORAM [ZWR+16], as discussed in Section 7.7, considering only ORAM access time.												
4 KB	3.10	14.7	†	319	1477	†	6.57	30.2	†	437	9810	†
16 KB	†	†	†	†	†	†	†	†	†	†	†	†
64 KB	†	†	†	†	†	†	†	†	†	†	†	†
Primitive Metal using Floram [DS17], as discussed in Section 7.7, considering only ORAM access time.												
4 KB	3.62	4.55	9.76	100	129	290	2.77	2.77	2.77	4.04	9.33	88.0
16 KB	6.67	9.80	31.2	399	514	1152	11.0	11.0	11.0	7.54	30.1	342
64 KB	19.5	32.3	†	1592	2048	†	44.1	44.1	†	24.0	110	†

Table 5: Metal-ORAM's file access latencies compared with Primitive Metal ($\star =$ same as amortized, $\dagger =$ out-of-memory). Network I/O is measured using `iftop`. All results are the average of (at least) one hundred ORAM write operations.

- **Latency reduction.** By extrapolating Table 5's result, we estimate the file access latency for Primitive Metal is ≥ 75 s for 2^{20} 64 KB storage (for Circuit ORAM [WCS15]). Metal uses tree-based ORAM, which has a polylogarithmic worst-case complexity and significantly less computation. In particular, Metal avoids the linear worst-case time as in SqrtORAM [ZWR+16] and Floram [DS17].
- **Network I/O reduction.** Metal reduces the amortized network I/O because it no longer does data-intensive computation in Yao's protocol (as shown in the circuit size in Table 5) and only transfers a few file data blocks per file access. If we extrapolate Table 5's result, the amortized network I/O of Primitive Metal is ≥ 1 GB accessing a 64 KB file in 2^{20} files. In comparison, the network I/O for Metal is about 105 MB, as Table 5 shows.

8 Extensions

In this section we discuss certain extensions to Metal.

Parallel accesses. We can leverage a system technique called quorum consensus [ABD90]. For example, we can improve the read performance by having K pairs of Metal servers with the same file data but independent ORAM store. They can load-balance the user's read requests and are very likely to improve the throughput by $K\times$. The write performance will decrease because a user needs to submit the write request to all K pairs of the servers. In systems where the write requests happen very infrequently, such a design can be helpful in reducing the average latency. Note that this direction to improve the performance often has to sacrifice the read/write indistinguishability.

Padding to hide timing and type of operation. The leakage of timing and type of operation can be hidden by padding in time and computation. To do so, we first modify Metal server API functions to support a *dummy mode* that does not make any actual change but exhibits the same execution patterns. We will not discuss how to implement this dummy mode, but it will rely mostly on general techniques. Then, we ask each user's client to routinely call each server API function; when a client is expected to call a server API function but has nothing to do, the client simply invokes the function in the dummy mode. Nevertheless, such padding is very expensive (e.g., the broadcast list will be lengthy).

9 Related Work

We organize the related work in the following categories:

(1) E2EE storage systems. A line of storage systems uses end-to-end encryption. Academic works include DepSky [BCQ+11], M-Aegis [LCS+14], Mylar [PSV+14], Plutus [KRS+03], ShadowCrypt [HAJ+14], Sieve [WMZV16], and SiRiUS [GSMB03]. In industry, there are Keybase [KBFS], PreVeil [PreVeil], and Tresorit [Tresorit]. These systems are practical, but they leak user identities and file access patterns.

(2) Anonymous storage systems. There has been a line of works on anonymous storage systems. Earlier academic works include Eternity [And96], Publius [WRC00], Freenet [CSWH01], and Free Haven [DFM01]. Some peer-to-peer file-sharing systems have been deployed in the real world, including Napster, Gnutella, and Mojo Nation [MojoNation].

(3) Single-user oblivious storage systems. Oblivious storage systems are designed to conceal file access patterns and provide strong privacy. Single-user oblivious storage systems focus on the setting where there is only one user [Gol87; Ost90; PR10; GM11; SSS12; MLS+13; DSS14; RFK+15] or a group of *trusted* users that can be treated as one user's multiple clients [WST12; SS13b; BNP+15; SZE+16; BMN17; CS19]. A number of works add the support of asynchronous access [WST12; SS13b; BNP+15; SZE+16; CS19] and improve the security against malicious servers [BMN17]. Multi-cloud ORAM [SS13a; LO13] uses multiple non-colluding servers to achieve a high throughput, but it does not support malicious users using the same ORAM store.

(4) Multi-user oblivious storage systems. Multi-user oblivious storage systems are more challenging since every single user is not fully trusted. There are only a few works in this direction [MMRS15; ZZQ16; BHKP16; MMRS17; KPK17; HOWW19]. We discuss them as follows.

Secret-write PANDA [HOWW19] is a multi-user oblivious storage that does not support data sharing, and thus differs from Metal in terms of functionality. Another disadvantage is that it needs to bound the number of malicious users—which is difficult for systems with open membership—and the performance degrades linearly to this bound. If the number of malicious users is not bounded sufficiently at the setup of the scheme, the scheme cannot provide the desired privacy guarantees. In addition, secret-write PANDA runs expensive cryptography, and the performance is likely very impractical.

AnonRAM-lin and AnonRAM-poly [BHKP16] enable many mutually distrusting users to use the same ORAM storage anonymously, but these users cannot share files. AnonRAM-lin has a linear overhead, the performance of which can be prohibitive. And though AnonRAM-poly has a poly-logarithmic overhead, extending it with file sharing is hard: it reveals at which level the data block is in the Goldreich-Ostrovsky ORAM (GO-ORAM) [Gol87; Ost90; GO96], which involves file access history. This is not a problem in AnonRAM-poly because users do not share files. But, if we add file sharing, a group of users sharing the same file will now learn information about one another's access patterns. Fixing this problem requires substantial changes. Moreover, AnonRAM-poly has a linear worst-case overhead, which is undesirable for practical systems [SCSL11].

GORAM [MMRS15] is a multi-user oblivious storage system with anonymity and obliviousness against servers. Its limitation is that GORAM does not provide obliviousness against malicious users, which makes GORAM harder to be used for open systems like Dropbox [Dropbox] where any user can sign up. In addition, GORAM does not hide the owner of a file.

PIR-MCORAM [MMRS17] is a multi-user oblivious file-sharing system that uses a single server and hides a very large class of metadata, including file access patterns (except whether the operation is reading or writing), but it reveals the user identities to the server when the user writes to a file. Metal improves over PIR-MCORAM by avoiding the linear complexity and hides the user identities in both reading and writing. Compared with Metal, PIR-MCORAM has the benefit of using only one single server.

There are also some multi-user ORAM schemes that focus on multiple semi-honest users' sharing files [ZZQ16; KPK17].

(5) RAM-model secure computation. Primitive Metal builds on top of RAM-model secure computation (RAM-SC) [GKK+12; LO13; WHC+14; WCS15; ZWR+16; DS17]. With Primitive Metal being limited in functionality and performance, Metal represents a comprehensive solution for file storage.

(6) Miscellaneous. Secure messaging [CBM15; HLZZ15; AS16; TGL+17; KCDF17; PHE+17; KLD20] also strives to hide metadata in user communication. Nevertheless, it does not store data persistently and usually requires users to stay online, which is difficult in practice. Metadata-hiding storage can also be constructed using hardware enclaves [MPC+18; AKSL18; SGF18], but it requires additional hardware assumptions.

Acknowledgment

We thank our shepherds Thomas Schneider and Amos Treiber, who provide clear and detailed revision guidelines to this paper. In particular, we want to thank them for pointing out the necessity to use the new half-gate garbling scheme (as in [GKW+19]) in Obliv-C for better computational security.

We thank the anonymous reviewers for their valuable feedback on this manuscript or their earlier versions. The manuscript has been improved significantly across the submissions.

We thank Pratyush Mishra, Sam Kumar, Wenting Zheng, Rishabh Poddar, and many other colleagues at UC Berkeley for feedback on the early drafts of this paper.

We thank Samee Zahur for the help with the Obliv-C platform, and we thank Chenkai Weng for implementing the new half-gate garbling scheme, as in EMP-toolkit [EMP].

This work has been supported by the NSF CISE Expeditions Award CCF-1730628, Jim Gray Fellowship, J. K. Zee Fellowship, as well as gifts from the Sloan Foundation, Bakar, Okawa, and Hellman Fellows Fund, Alibaba, Amazon Web Services, Ant Financial, Capital One, Ericsson, Facebook, Futurewei, Google, Intel, Microsoft, Nvidia, Scotiabank, Splunk, and VMware. Our benchmark received support from the AWS Cloud Credits for Research program.

Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation and other funding parties.

References

[ABD90] H. Attiya, A. Bar-Noy, and D. Dolev, “Sharing memory robustly in message-passing systems,” in *PODC’90*, 1990.

[ACLS18] S. Angel, H. Chen, K. Laine, and S. Setty, “PIR with compressed queries and amortized query processing,” in *S&P’18*, 2018.

[Agg05] C. C. Aggarwal, “On k-anonymity and the curse of dimensionality,” in *VLDB’05*, 2005.

[AKSL18] A. Ahmad, K. Kim, M. I. Sarfaraz, and B. Lee, “OBLIVIATE: A data oblivious file system for Intel SGX,” in *NDSS’18*, 2018.

[And96] R. Anderson, “The Eternity service,” in *PRAGOCRYPT’96*, 1996.

[AS16] S. Angel and S. Setty, “Unobservable communication over fully untrusted infrastructure,” in *OSDI’16*, 2016.

[Autocrypt] *Autocrypt: Convenient end-to-end encryption for e-mail*, https://autocrypt.org/.

[AWSBench] *Benchmark network throughput between Amazon EC2 Linux instances in the same VPC*, https://aws.amazon.com/premiumsupport/knowledge-center/network-throughput-benchmark-linux-ec2/.

[BCQ+11] A. Bessani, M. Correia, B. Quaresma, F. André, and P. Sousa, “DepSky: Dependable and secure storage in a cloud-of-clouds,” in *EuroSys’11*, 2011.

[BDK07] L. Backstrom, C. Dwork, and J. Kleinberg, “Wherefore art thou R3579x?: Anonymized social networks, hidden patterns, and structural steganography,” in *WWW’07*, 2007.

[Ber06] D. J. Bernstein, “Curve25519: New Diffie-Hellman speed records,” in *PKC’06*, 2006.

[BGV12] Z. Brakerski, C. Gentry, and V. Vaikuntanathan, “(Leveled) fully homomorphic encryption without bootstrapping,” in *ITCS’12*, 2012.

[BHKP16] M. Backes, A. Herzberg, A. Kate, and I. Pryvalov, “Anonymous RAM,” in *ESORICS’16*, 2016.

[BHKR13] M. Bellare, V. T. Hoang, S. Keelveedhi, and P. Rogaway, “Efficient garbling from a fixed-key blockcipher,” in *S&P’13*, 2013.

[BMN17] E.-O. Blass, T. Mayberry, and G. Noubir, “Multi-client oblivious RAM secure against malicious servers,” in *ACNS’17*, 2017.

[BMR90] D. Beaver, S. Micali, and P. Rogaway, “The round complexity of secure protocols,” in *STOC’90*, 1990.

[BNP+15] V. Bindschaedler, M. Naveed, X. Pan, X. Wang, and Y. Huang, “Practicing oblivious access on cloud storage: The gap, the fallacy, and the new way forward,” in *CCS’15*, 2015.

[BPTG15] R. Bost, R. A. Popa, S. Tu, and S. Goldwasser, “Machine learning classification over encrypted data,” in *NDSS’15*, 2015.

[BRW04] M. Bellare, P. Rogaway, and D. Wagner, “The EAX mode of operation,” in *FSE’04*, 2004.

[BS16] R. Bost and O. Sanders, “Trick or tweak: On the (in)security of OTR’s tweaks,” in *ASIACRYPT’16*, 2016.

[CB17] H. Corrigan-Gibbs and D. Boneh, “Prio: Private, robust, and scalable computation of aggregate statistics,” in *NSDI’17*, 2017.

[CBM15] H. Corrigan-Gibbs, D. Boneh, and D. Mazières, “Riposte: An anonymous messaging system handling millions of users,” in *S&P’15*, 2015.

[CDS94] R. Cramer, I. Damgård, and B. Schoenmakers, “Proofs of partial knowledge and simplified design of witness hiding protocols,” in *CRYPTO’94*, 1994.

[CGPR15] D. Cash, P. Grubbs, J. Perry, and T. Ristenpart, “Leakage-abuse attacks against searchable encryption,” in *CCS’15*, 2015.

[Col14] D. Cole, “We kill people based on metadata,” in *The New York Review of Books—NYR Daily May 10, 2014*, 2014.

[CS17] T.-H. H. Chan and E. Shi, “Circuit OPRAM: Unifying statistically and computationally secure ORAMs and OPRAMs,” in *TCC’17*, 2017.

[CS19] A. Chakraborti and R. Sion, “ConcurORAM: High-throughput stateless parallel multi-client ORAM,” in *NDSS’19*, 2019.

[CSWH01] I. Clarke, O. Sandberg, B. Wiley, and T. W. Hong, “Freenet: A distributed anonymous information storage and retrieval system,” in *International Workshop on Designing Privacy Enhancing Technologies 2001*, 2001.

[CTT94] R. Card, T. Ts’o, and S. Tweedie, “Design and implementation of the second extended filesystem,” in *Dutch International Symposium on Linux’94*, 1994.

[DDF+16] S. Devadas, M. van Dijk, C. W. Fletcher, L. Ren, E. Shi, and D. Wichs, “Onion ORAM: A constant bandwidth blowup oblivious RAM,” in *TCC’16*, 2016.

[DFM01] R. Dingledine, M. J. Freedman, and D. Molnar, “The Free Haven Project: Distributed anonymous storage service,” in *PET’01*, 2001.

[DH66] J. B. Dennis and E. C. Van Horn, “Programming semantics for multiprogrammed computations,” in *CACM’66*, 1966.

[DMS04] R. Dingledine, N. Mathewson, and P. Syverson, “Tor: The second-generation onion router,” in *SEC’04*, 2004.

[Doe18] J. Doerner, *Absentminded Crypto Kit*, https://bitbucket.org/jackdoerner/absentminded-crypto-kit, 2018.

[Dropbox] *Dropbox*, https://www.dropbox.com/.

[DS17] J. Doerner and A. shelat, “Scaling ORAM for secure computation,” in *CCS’17*, 2017.

[DSS14] J. Dautrich, E. Stefanov, and E. Shi, “Burst ORAM: Minimizing ORAM response times for bursty access patterns,” in *SEC’14*, 2014.

[EGL85] S. Even, O. Goldreich, and A. Lempel, “A randomized protocol for signing contracts,” in *CACM’85*, 1985.

[ElG84] T. ElGamal, “A public key cryptosystem and a signature scheme based on discrete logarithms,” in *CRYPTO’84*, 1984.

[EMP] *Efficient multi-party computation toolkit*. [Online]. Available: https://github.com/emp-toolkit.

[GGH+13] C. Gentry, K. A. Goldman, S. Halevi, C. Julta, M. Raykova, and D. Wichs, “Optimizing ORAM and using it efficiently for secure computation,” in *PETS’13*, 2013.

[GKK+12] S. D. Gordon, J. Katz, V. Kolesnikov, F. Krell, T. Malkin, M. Raykova, and Y. Vahlis, “Secure two-party computation in sublinear (amortized) time,” in *CCS’12*, 2012.

[GKW+19] C. Guo, J. Katz, X. Wang, C. Weng, and Y. Yu, “Better concrete security for half-gates garbling (in the multi-instance setting),” in *IACR ePrint 2019/1168*, 2019.

[GKWY20] C. Guo, J. Katz, X. Wang, and Y. Yu, “Efficient and secure multiparty computation from fixed-key block ciphers,” in *S&P’20*, 2020.

[GLMP18] P. Grubbs, M.-S. Lacharité, B. Minaud, and K. G. Paterson, “Pump up the volume: Practical database reconstruction from volume leakage on range queries,” in *CCS’18*, 2018.

[GLMP19] ——, “Learning to reconstruct: Statistical learning theory and encrypted database attacks,” in *S&P’19*, 2019.

[GM11] M. T. Goodrich and M. Mitzenmacher, “Privacy-preserving access of outsourced data via oblivious RAM simulation,” in *ICALP’11*, 2011.

[GM82] S. Goldwasser and S. Micali, “Probabilistic encryption & how to play mental poker keeping secret all partial information,” in *STOC’82*, 1982.

[GMN+16] P. Grubbs, R. McPherson, M. Naveed, T. Ristenpart, and V. Shmatikov, “Breaking web applications built on top of encrypted data,” in *CCS’16*, 2016.

[GMW87] O. Goldreich, S. Micali, and A. Wigderson, “How to play ANY mental game,” in *STOC’87*, 1987.

[GO96] O. Goldreich and R. Ostrovsky, “Software protection and simulation on oblivious RAMs,” in *JACM’96*, 1996.

[Gol87] O. Goldreich, “Towards a theory of software protection and simulation by oblivious RAMs,” in *STOC’87*, 1987.

[GSMB03] E.-J. Goh, H. Shacham, N. Modadugu, and D. Boneh, “SiRiUS: Securing remote untrusted storage,” in *NDSS’03*, 2003.

[HAJ+14] W. He, D. Akhawe, S. Jain, E. Shi, and D. Song, “ShadowCrypt: Encrypted web applications for everyone,” in *CCS’14*, 2014.

[Ham15] M. Hamburg, “Decaf: Eliminating cofactors through point compression,” in *CRYPTO’15*, 2015.

[HElib] *HElib: An implementation of homomorphic encryption*, https://github.com/homenc/HElib.

[HKM15] V. T. Hoang, J. Katz, and A. J. Malozemoff, “Automated analysis and synthesis of authenticated encryption schemes,” in *CCS’15*, 2015.

[HLZZ15] J. van den Hooff, D. Lazar, M. Zaharia, and N. Zeldovich, “Vuvuzela: Scalable private messaging resistant to traffic analysis,” in *SOSP’15*, 2015.

[HOWW19] A. Hamlin, R. Ostrovsky, M. Weiss, and D. Wichs, “Private anonymous data access,” in *EUROCRYPT’19*, 2019.

[IKK12] M. Islam, M. Kuzu, and M. Kantarcioglu, “Access pattern disclosure on searchable encryption: Ramification, attack and mitigation,” in *NDSS’12*, 2012.

[JJ99] M. Jakobsson and A. Juels, “Millimix: Mixing in small batches,” in *DIMACS TR’99*, 1999.

[JLG+15] S. Ji, W. Li, N. Z. Gong, P. Mittal, and R. Beyah, “On your social network de-anonymizablity: Quantification and large scale evaluation with seed knowledge,” in *NDSS’15*, 2015.

[JLM+15] S. Ji, W. Li, P. Mittal, X. Hu, and R. Beyah, “SecGraph: A uniform and open-source evaluation system for graph data anonymization and de-anonymization,” in *SEC’15*, 2015.

[KBFS] *Keybase filesystem (KBFS)*, https://github.com/keybase/kbfs.

[KCDF17] A. Kwon, H. Corrigan-Gibbs, S. Devadas, and B. Ford, “Atom: Horizontally scaling strong anonymity,” in *SOSP’17*, 2017.

[KGK+18] N. Kilbertus, A. Gascón, M. Kusner, M. Veale, K. Gummadi, and A. Weller, “Blind Justice: Fairness with encrypted sensitive attributes,” in *ICML’18*, 2018.

[KLD20] A. Kwon, D. Lu, and S. Devadas, “XRD: Scalable messaging system with cryptographic privacy,” in *NSDI’20*, 2020.

[KLI+18] B. Kulynych, W. Lueks, M. Isaakidis, G. Danezis, and C. Troncoso, “ClaimChain: Improving the security and privacy of in-band key distribution for messaging,” in *WPES’18*, 2018.

[KPK17] N. P. Karvelas, A. Peter, and S. Katzenbeisser, “Using oblivious RAM in genomic studies,” in *Data Privacy Management, Cryptocurrencies and Blockchain Technology’17*, 2017.

[KRS+03] M. Kallahalla, E. Riedel, R. Swaminathan, Q. Wang, and K. Fu, “Plutus: Scalable secure file sharing on untrusted storage,” in *FAST’03*, 2003.

[KS08] V. Kolesnikov and T. Schneider, “Improved garbled circuit: Free XOR gates and applications,” in *ICALP’08*, 2008.

[LCS+14] B. Lau, S. Chung, C. Song, Y. Jang, W. Lee, and A. Boldyreva, "Mimesis Aegis: A mimicry privacy shield–a system's approach to data privacy on public cloud," in *SEC'14*, 2014.

[LMP18] M.-S. Lacharité, B. Minaud, and K. G. Paterson, "Improved reconstruction attacks on encrypted data using range query leakage," in *S&P'18*, 2018.

[LO13] S. Lu and R. Ostrovsky, "Distributed oblivious RAM for secure two-party computation," in *TCC'13*, 2013.

[Min14] K. Minematsu, "Parallelizable rate-1 authenticated encryption from pseudorandom functions," in *EUROCRYPT'14*, 2014.

[MLS+13] M. Maas, E. Love, E. Stefanov, M. Tiwari, E. Shi, K. Asanovic, J. Kubiatowicz, and D. Song, "PHANTOM: Practical oblivious computation in a secure processor," in *CCS'13*, 2013.

[MMRS15] M. Maffei, G. Malavolta, M. Reinert, and D. Schröder, "Privacy and access control for outsourced personal records," in *S&P'15*, 2015.

[MMRS17] ——, "Maliciously secure multi-client ORAM," in *ACNS'17*, 2017.

[MojoNation] *Mojo Nation*, https://sourceforge.net/projects/mojonation/.

[MPC+18] P. Mishra, R. Poddar, J. Chen, A. Chiesa, and R. A. Popa, "Oblix: An efficient oblivious search index," in *S&P'18*, 2018.

[MRCK19] E. V. Mangipudi, K. Rao, J. Clark, and A. Kate, "Towards automatically penalizing multimedia breaches," in *EuroS&PW'19*, 2019.

[MV04] D. A. McGrew and J. Viega, "The security and performance of the Galois/counter mode (GCM) of operation," in *INDOCRYPT'04*, 2004.

[MZ17] P. Mohassel and Y. Zhang, "SecureML: A system for scalable privacy-preserving machine learning," in *S&P'17*, 2017.

[NKA14] S. Nilizadeh, A. Kapadia, and Y.-Y. Ahn, "Community-enhanced de-anonymization of online social networks," in *CCS'14*, 2014.

[NR16] J. B. Nielsen and S. Ranellucci, "Reactive garbling: Foundation, instantiation, application," in *ASIACRYPT'16*, 2016.

[NS08] A. Narayanan and V. Shmatikov, "Robust de-anonymization of large sparse datasets," in *S&P'08*, 2008.

[OpenPGP] *OpenPGP*, https://www.openpgp.org/.

[Ost90] R. Ostrovsky, "Efficient computation on oblivious RAMs," in *STOC'90*, 1990.

[Pai99] P. Paillier, "Public-key cryptosystems based on composite degree residuosity classes," in *EUROCRYPT'99*, 1999.

[PHE+17] A. M. Piotrowska, J. Hayes, T. Elahi, S. Meiser, and G. Danezis, "The Loopix anonymity system," in *SEC'17*, 2017.

[Poi] D. Poirier, *The second extended file system: Internal layout*, https://www.nongnu.org/ext2-doc/ext2.html.

[PR10] B. Pinkas and T. Reinman, "Oblivious RAM revisited," in *CRYPTO'10*, 2010.

[PreVeil] *PreVeil: End-to-end encryption for secure communication*, https://www.preveil.com/.

[PSV+14] R. A. Popa, E. Stark, S. Valdez, J. Helfer, N. Zeldovich, and H. Balakrishnan, "Building web applications on top of encrypted data using Mylar," in *NSDI'14*, 2014.

[Rab81] M. O. Rabin, "How to exchange secrets with oblivious transfer," in *Technical Report TR-81, Aiken Computation Lab, Harvard University'81*, 1981.

[RAC16] D. S. Roche, A. Aviv, and S. G. Choi, "A practical oblivious map data structure with secure deletion and history independence," in *S&P'16*, 2016.

[RBBK01] P. Rogaway, M. Bellare, J. Black, and T. Krovetz, “OCB: A block-cipher mode of operation for efficient authenticated encryption,” in *CCS’01*, 2001.

[RFK+15] L. Ren, C. Fletcher, A. Kwon, E. Stefanov, E. Shi, M. van Dijk, and S. Devadas, “Constants count: Practical improvements to oblivious RAM,” in *SEC’15*, 2015.

[Ristretto] *The Ristretto group*, https://ristretto.group/ristretto.html.

[Rus13] A. Rusbridger, “The Snowden leaks and the public,” in *The New York Review of Books—NYR Daily November 21, 2013*, 2013.

[Sch89] C.-P. Schnorr, “Efficient identification and signatures for smart cards,” in *CRYPTO’89*, 1989.

[SCSL11] E. Shi, T. H. H. Chan, E. Stefanov, and M. Li, “Oblivious RAM with $O((logN)^3)$ worst-case cost,” in *ASIACRYPT’11*, 2011.

[SDS+13] E. Stefanov, M. van Dijk, E. Shi, C. Fletcher, L. Ren, X. Yu, and S. Devadas, “Path ORAM: An extremely simple oblivious RAM protocol,” in *CCS’13*, 2013.

[SGF18] S. Sasy, S. Gorbunov, and C. W. Fletcher, “ZeroTrace: Oblivious memory primitives from Intel SGX,” in *NDSS’18*, 2018.

[SS13a] E. Stefanov and E. Shi, “Multi-cloud oblivious storage,” in *CCS’13*, 2013.

[SS13b] ——, “ObliviStore: High performance oblivious cloud storage,” in *S&P’13*, 2013.

[SSS12] E. Stefanov, E. Shi, and D. Song, “Towards practical oblivious RAM,” in *NDSS’12*, 2012.

[SV10] N. P. Smart and F. Vercauteren, “Fully homomorphic encryption with relatively small key and ciphertext sizes,” in *PKC’10*, 2010.

[SV14] ——, “Fully homomorphic SIMD operations,” in *Designs, Codes and Cryptography’14*, 2014.

[SZE+16] C. Sahin, V. Zakhary, A. El Abbadi, H. Lin, and S. Tessaro, “TaoStore: Overcoming asynchronicity in oblivious data storage,” in *S&P’16*, 2016.

[TGL+17] N. Tyagi, Y. Gilad, D. Leung, M. Zaharia, and N. Zeldovich, “Stadium: A distributed metadata-private messaging system,” in *SOSP’17*, 2017.

[Tresorit] *Tresorit: Secure file sharing & content collaboration with encryption*, https://tresorit.com/.

[WAP+19] L. Wang, G. Asharov, R. Pass, T. Ristenpart, and A. shelat, “Blind certificate authorities,” in *S&P’19*, 2019.

[WCS15] X. Wang, H. Chan, and E. Shi, “Circuit ORAM: On tightness of the Goldreich-Ostrovsky lower bound,” in *CCS’15*, 2015.

[WGL+17] L. Wang, P. Grubbs, J. Lu, V. Bindschaedler, D. Cash, and T. Ristenpart, “Side-channel attacks on shared search indexes,” in *S&P’17*, 2017.

[WHC+14] X. Wang, Y. Huang, T.-H. H. Chan, A. shelat, and E. Shi, “SCORAM: Oblivious RAM for secure computation,” in *CCS’14*, 2014.

[WHKK10] G. Wondracek, T. Holz, E. Kirda, and C. Kruegel, “A practical attack to de-anonymize social network users,” in *S&P’10*, 2010.

[WHO] *World health organization (WHO): Health statistics and information systems*, https://www.who.int/healthinfo/en/.

[WMZV16] F. Wang, J. Mickens, N. Zeldovich, and V. Vaikuntanathan, “Sieve: Cryptographically enforced access control for user data in untrusted clouds,” in *NSDI’16*, 2016.

[WRC00] M. Waldman, A. D. Rubin, and L. F. Cranor, “Publius: A robust, tamper-evident, censorship-resistant web publishing system,” in *SEC’00*, 2000.

[WST12] P. Williams, R. Sion, and A. Tomescu, “PrivateFS: A parallel oblivious file system,” in *CCS ’12*, 2012.

[WYG+17] F. Wang, C. Yun, S. Goldwasser, V. Vaikuntanathan, and M. Zaharia, "Splinter: Practical private queries on public data," in *NSDI'17*, 2017.

[Yao86] A. C.-C. Yao, "How to generate and exchange secrets," in *FOCS'86*, 1986.

[ZE15] S. Zahur and D. Evans, "Obliv-C: A language for extensible data-oblivious computation," in *IACR ePrint 2015/1153*, 2015.

[ZKP16] Y. Zhang, J. Katz, and C. Papamanthou, "All your queries are [*sic*] belong to us: The power of file-injection attacks on searchable encryption," in *SEC'16*, 2016.

[ZRE15] S. Zahur, M. Rosulek, and D. Evans, "Two halves make a whole: Reducing data transfer in garbled circuits using half gates," in *EUROCRYPT'15*, 2015.

[ZWR+16] S. Zahur, X. Wang, M. Raykova, A. Gascón, J. Doerner, D. Evans, and J. Katz, "Revisiting square-root ORAM: Efficient random access in multi-party computation," in *S&P'16*, 2016.

[ZZQ16] J. Zhang, W. Zhang, and D. Qiao, "MU-ORAM: Dealing with stealthy privacy attacks in multi-user data outsourcing services," in *IACR ePrint 2016/073*, 2016.

A Security proof of Metal-ORAM

In Section 4 and Section 6.1 we provided the security proof sketches for Metal-AC and Metal-SHARE, which we believe are sufficient to reconstruct a formal proof. In this section we prove the security of Metal-ORAM in the following real-ideal paradigm:

– In the **real world** two servers run the Metal-ORAM protocol. An adversary A sees the view of one of the servers and can control a set of users in a malicious way.
– In the **ideal world** an ideal functionality $\mathcal{F}_{\text{Metal-ORAM}}$ realizes Metal-ORAM with the desired security guarantees. The simulator Sim forges A's view like in the real world.

Metal-ORAM is secure if A's output in the real world is computationally indistinguishable from Sim's in the ideal world.

A.1 Ideal functionality

The ideal functionality $\mathcal{F}_{\text{Metal-ORAM}}$ stores the file data in array $\mathsf{FileData}$, where $\mathsf{FileData}[\mathsf{ID}_F]$ stores the file identified by ID_F. $\mathcal{F}_{\text{Metal-ORAM}}$ has the following interface:

Configure. Ask Sim which server to compromise, denoted by $\mathsf{CompromisedSrvID} \in \{1, 2\}$.

Read. On receiving $(C_F^{P_F}, \mathsf{NewFileData}, \mathtt{READ})$ from a user in two shares, check if $C_F^{P_F}$ is valid (using Metal-AC), check $\mathsf{NewFileData}$'s format, and if both checks pass,

– obtain ID_F from Metal-AC, find $\mathsf{FileData}[\mathsf{ID}_F]$, and secret-share $\mathsf{FileData}[\mathsf{ID}_F]$ into $\mathsf{TargetData}^{(1)}$ and $\mathsf{TargetData}^{(2)}$; send $\mathsf{TargetData}^{(1)}$ and $\mathsf{TargetData}^{(2)}$ to the user.
– send Sim $(\mathtt{NEW\_REQUEST}, \mathsf{TargetData}^{(j)}, \mathsf{NewFileData}^{(j)}, C_F^{P_F,(j)})$ where $j = \mathsf{CompromisedSrvID}$. (This message reflects the communication between the user and the compromised server.)

If $C_F^{P_F}$ is invalid or $\mathsf{NewFileData}$ is malformed, send the user and Sim $\mathtt{INVALID\_CAPABILITY}$ or $\mathtt{INVALID\_FORMAT}$, respectively. For invalid formats, send which shares are malformed.

Write. On receiving $(C_F^{P_F}, \mathsf{NewFileData}, \mathtt{WRITE})$, check $C_F^{P_F}$ and $\mathsf{NewFileData}$. If both checks pass,

– obtain ID_F and change $\mathsf{FileData}[\mathsf{ID}_F]$ to $\mathsf{NewFileData}$.
– use the dummy file data as $\mathsf{TargetData}$, secret-share it into $\mathsf{TargetData}^{(1)}$ and $\mathsf{TargetData}^{(2)}$, and send them to the user.
– send Sim $(\mathtt{NEW\_REQUEST}, \mathsf{TargetData}^{(j)}, \mathsf{NewFileData}^{(j)}, C_F^{P_F,(j)})$ where $j = \mathsf{CompromisedSrvID}$.

Otherwise, send to the user and Sim $\mathtt{INVALID\_CAPABILITY}$ or $\mathtt{INVALID\_FORMAT}$, as described above.

A.2 Simulator

The simulator Sim learns from $\mathcal{F}_{\text{Metal-ORAM}}$ certain information about a request and knows the system setup, such as the number of files supported by the servers. Sim works as follows:

Initialize. Run A and control A's execution and network. Let A choose $\mathsf{CompromisedSrvID}$ and forward it to $\mathcal{F}_{\text{Metal-ORAM}}$. Sample two ElGamal key pairs, one for each server. Send both public keys and the compromised server's private key to A. Merge the public keys into one global public key. If $\mathsf{CompromisedSrvID} = 1$, instantiate DataORAM.

Initiate a file access request. When A wants to send a request to the servers, forward the request to $\mathcal{F}_{\text{Metal-ORAM}}$.

Forge the compromised server's view. On receiving a message from $\mathcal{F}_{\text{Metal-ORAM}}$, simulate the compromised server's view and provide this view to A. In what follows, we describe the case when A compromises Server 1, and we omit the case for Server 2, which is similar.

If the request is valid, parse the message from $\mathcal{F}_{\text{Metal-ORAM}}$ as $(\texttt{NEW\_REQUEST}, \mathsf{TargetData}^{(1)}, \mathsf{NewFileData}^{(1)}, C_F^{P_F,(1)})$ and run as follows:

- invoke the simulator for Yao's protocol for the capability check.
- simulate the view in which Server 1 checked the format of $\mathsf{NewFileData}^{(1)}$ and exchanged the results of the format check with Server 2.
- invoke the simulator for Yao's protocol for ORAM access to a random path in IndexORAM and for sharing a fake block location (denoted by i); this step corresponds to reading the file index in Section 5.3.
- simulate the SS-DOT to act as if Server 2 would obtain the i-th block on that path (or a dummy block), as follows:
  - invoke the simulator of Yao's protocol for the first step of SS-DOT, which, within S2PC, reconstructs i, samples $N = |\mathsf{stash}| + 3 \times h + 1$ keys, and outputs these N keys to Server 1 and the i-th key to Server 2.
  - simulate the view where Server 1 read blocks from the (fake) storage, encrypted and rerandomized the blocks as the protocol specifies, and sent the encrypted blocks to Server 2.
- simulate the threshold decryption as follows:
  - encrypt $\mathsf{TargetData}^{(1)}$ with the global public key (the ciphertext is denoted by $\mathsf{FakeReadData}$).
  - simulate the view where Server 1 engaged in the threshold decryption of $\mathsf{FakeReadData}$ with Server 2, obtained $\mathsf{TargetData}^{(1)}$, and sent $\mathsf{TargetData}^{(1)}$ to the user.
- simulate the joint encryption of the user-provided new file data (in secret shares), as follows:
  - encrypt $\mathsf{NewFileData}^{(1)}$ provided by the user.
  - simulate the view in which Server 1 received a random encrypted data block from Server 2 and homomorphically added the ciphertext together; the resultant ciphertext is denoted by $\mathsf{FakeNewData}$.
- simulate the distributed permutation as follows:
  - sample a permutation $\sigma^{(1)}$ of the numbers $\{1, 2, ..., |\mathsf{stash}| + 6 \times h - 1\}$.
  - invoke the simulator for Yao's protocol for the ORAM eviction in IndexORAM and the permutation generation inside S2PC, where Server 1 received $\sigma^{(1)}$.
  - simulate the view where Server 1 constructed an array of the size $|\mathsf{stash}|+6\times h-1$, which began with data blocks from the two paths selected by the reverse lexicographic order and followed by $\mathsf{FakeReadData}$ and $\mathsf{FakeNewData}$.
  - simulate the view where Server 1 rerandomized and permuted the array according to $\sigma^{(1)}$ and sent the array to Server 2.
  - sample an array of $|\mathsf{stash}| + 6 \times h - 1$ encrypted dummy data blocks, denoted by $\mathsf{FakePermutedArray}$.
  - simulate the view where Server 1 received from Server 2 $\mathsf{FakePermutedArray}$ and stored it in the storage.
- provide Server 1's view to A.

If the request is invalid because $C_F^{P_F,(1)}$ is invalid, proceed as follows:

- invoke the simulator for Yao's protocol for the capability check, which fails.
- simulate the view where Server 1 responded to the user that the request was invalid.
- provide Server 1's view to A.

If the request is invalid because the data format is incorrect (though $C_F^{P_F,(1)}$ is valid), proceed as follows:

- invoke the simulator for Yao's protocol for the capability check, which passes.
- simulate the view where Server 1 performed the format check of $\mathsf{NewFileData}^{(1)}$.
- simulate the view where Server 1 exchanged the format check results with Server 2 and responded to the user that the request was invalid.
- provide Server 1's view to A.

Continue running A and output whatever A outputs.

A.3 Proof of indistinguishability

We use the following hybrids (denoted by $\mathsf{H}_\cdot$) to show that the view that Sim forges is computationally indistinguishable (denoted by $\approx$) from the compromised server's view in the real world. As follows, we focus on the case where Server 1 is compromised, and particularly, the situation when Sim receives a `NEW_REQUEST` message from $\mathcal{F}_{\text{Metal-ORAM}}$.

Consider q requests, where q is polynomially bounded by the security parameter. Our proof will replace the simulated view of each of the q requests one by one, starting from the first request, with the view in the real execution for the same q requests. We use $\mathsf{H}_{t,i}$ to denote the i-th sub-hybrid of the t-th hybrid, in which t requests have been handled. We start with $\mathsf{H}_{0,0}$, which is the same as the simulated view in the ideal world. For each $t \in \{0, 1, ..., q-1\}$, we define:

- $\boldsymbol{\mathsf{H}_{t,0}}$ is $\mathsf{H}_{0,0}$ (if $t = 0$) or $\mathsf{H}_{t-1,7}$ (if $t \neq 0$). In the following hybrids, we focus on the handling of the $(t+1)$-th request.
- $\boldsymbol{\mathsf{H}_{t,1}}$ replaces the simulated view of capability check with the real execution's view. Security of S2PC implies $\mathsf{H}_{t,1} \approx \mathsf{H}_{t,0}$.
- $\boldsymbol{\mathsf{H}_{t,2}}$ replaces the simulated view of the ORAM access to IndexORAM with the real execution's view. Both views have the same distribution of ORAM access patterns. Security of S2PC implies $\mathsf{H}_{t,2} \approx \mathsf{H}_{t,1}$.
- $\boldsymbol{\mathsf{H}_{t,3}}$ replaces the simulated view of the first step of SS-DOT with the real execution's view where both views generate the same N random keys. Security of S2PC implies $\mathsf{H}_{t,3} \approx \mathsf{H}_{t,2}$.
- $\boldsymbol{\mathsf{H}_{t,4}}$ replaces the simulated view for threshold decryption with the real execution's view. In the simulation, Sim encrypts $\mathsf{TargetData}^{(1)}$, pretending to be from Server 2, and has Server 1 decrypt this ciphertext, in which the view of Server 1 (receiving and decrypting) has the same distribution as in the real execution. Thus, we have $\mathsf{H}_{t,4} \approx \mathsf{H}_{t,3}$.
- $\boldsymbol{\mathsf{H}_{t,5}}$ replaces the simulated view for joint encryption with the real execution's view. The difference between the two views is that, in $\mathsf{H}_{t,4}$, Server 1 receives a random data block, while in $\mathsf{H}_{t,5}$, Server 1 receives the ciphertext of $\mathsf{NewFileData}^{(2)}$, encrypted by Server 2. Using semantic security of ElGamal encryption, we have $\mathsf{H}_{t,5} \approx \mathsf{H}_{t,4}$.
- $\boldsymbol{\mathsf{H}_{t,6}}$ replaces the simulated view for ORAM eviction in IndexORAM and for permutation generation with the real execution's view that generates the same share of permutation $\sigma^{(1)}$ for Server 1. Security of S2PC implies $\mathsf{H}_{t,6} \approx \mathsf{H}_{t,5}$.
- $\boldsymbol{\mathsf{H}_{t,7}}$ replaces the simulated view for the rest of the distributed permutation (the parts after S2PC) with the real execution's view, which writes the data blocks into the storage (and updates the rest of the transcript accordingly). The main difference is that Server 1 receives a random data block array instead of the one permuted by $\sigma^{(2)}$. Because these data blocks are encrypted under randomness unknown to Server 1, Server 1 cannot distinguish these two arrays in different views. The rest is the same, and thus $\mathsf{H}_{t,7} \approx \mathsf{H}_{t,6}$.

The last hybrid, $\mathsf{H}_{q-1,7}$, has the same distribution as the real world's view. The hybrid arguments show that the simulated view is computationally indistinguishable from the real world's view. Therefore, we have the following theorem:

Theorem 1. *(informal) Assuming standard cryptographic assumptions and in the random oracle model, Metal-ORAM's protocol provides the claimed security guarantees.*

B Deamortization of ORAM initialization cost

Metal uses Circuit ORAM [WCS15] for its low amortized and worst-case ORAM complexity. One issue with Circuit ORAM is its high initialization cost, as discussed by Zahur et al. [ZWR+16]. The experiment in Zahur et al.'s paper shows that the initialization can take about 10^5 seconds for a store of 2^{20} four-byte data blocks. Though initialization is a one-time effort, it is useful to deamortize such a heavy cost. Metal-ORAM deamortizes the cost to initialize IndexORAM and DataORAM, as follows.

Deamortizing IndexORAM's initialization. The main challenge in deamortizing IndexORAM is to initialize the IndexORAM's position map in an oblivious and efficient manner. Recall that IndexORAM is a standard Circuit ORAM store, which uses the recursive technique to store the position map, as follows. To store the position map for N blocks, the recursive technique first packs every C blocks' position records into a block (we call it a *level-1* record block) and then stores these level-1 record blocks in an ORAM data structure, which consists of N/C level-1 record blocks. Next, the recursive technique creates the position map for these N/C record blocks, which can be stored in an ORAM data structure of N/C^2 blocks; we call them *level-2* record blocks. By repeating this manner for $\ell = O(\log N)$ times, we can have a constant number of level-ℓ record blocks. The creation of these ℓ ORAM data structures dominates the initialization cost in Circuit ORAM.

Metal-ORAM deamortizes this cost by having IndexORAM work with an *incomplete* position map. Our changes are as follows. Metal-ORAM first changes the record block's structure by appending a new bit $\mathsf{IsInit} \in \{0, 1\}$ at the beginning of each record block. If a record block's $\mathsf{IsInit} = 0$, the record block is considered as uninitialized. To start with, all these record blocks in IndexORAM's position map are filled with all zeros (i.e., IsInit is also 0), which is inexpensive in our secure computation platform Obliv-C [ZE15] since instantiating data structures in S2PC with zeros does not require network communication. These record blocks therefore all have $\mathsf{IsInit} = 0$ and are uninitialized. That is to say, after our initialization, all data blocks $D[x]$ have not been assigned a random position.

After such an initialization of IndexORAM, when Circuit ORAM algorithm accesses IndexORAM and runs into an uninitialized record block, it initializes this record block by assigning new random positions for the C next-level blocks in this record block and setting this block's IsInit to be 1. Note that the next-level blocks do not exist in those positions at this moment, but we pad the computation inside S2PC (by accessing a random path in the next level) such that the two servers do not know whether the record blocks being accessed are initialized or not.

This deamortizing technique reduces the initialization time, but the cost is in fact added into all future ORAM accesses (though, which is not the bottleneck in Metal-ORAM). We acknowledge that the total amount of work in the long term with our IndexORAM's initialization is larger than the full initialization described in [WCS15; ZWR+16]; our initialization mainly enables the two servers to set up in a much shorter time.

Metal-ORAM's initialization of IndexORAM takes 3.69 s under our experiment setup (Section 7.3).

Deamortizing DataORAM's initialization. Recall that DataORAM is a binary tree in which each node stores encrypted data blocks. To store 2^{20} 64 kB files, DataORAM takes ≈ 405 GB. The traditional way to initialize DataORAM is to create a ciphertext that encrypts dummy data and to populate the whole DataORAM with this ciphertext; this approach eventually takes ≈ 33 minutes under our experiment setup (Section 7.3) due to disk I/O (to an Amazon SSD gp2 store), which is heavy.

Metal-ORAM deamortizes this initialization cost by using sparse files [Poi; CTT94], a widely supported file type in which, initially, a file only has metadata, with no space assigned to the file. Then, when a program wants to read N bytes on an unassigned part, the operating system (OS) returns N bytes of zeros as the data, while the part stays unassigned. Only when a program writes data to an unassigned part of the file, the OS assigns actual disk space for that part and stores the new data on the disk. As a result, a sparse file can be created instantly (even if the file is supposed to be very large). When Metal-ORAM reads a ciphertext from an unassigned part (which is all-zero), Metal-ORAM instead uses a dummy ciphertext, so the rest of the protocol remains unchanged.

Metal-ORAM's creation of DataORAM takes ≤ 1 ms under our setup by using Linux's `truncate` command.

C Efficient authenticated encryption in garbled circuits

Metal-AC (Section 4) uses authenticated encryption (AE) for capabilities. To our knowledge, an extensive search for efficient AE in Yao's protocol has never appeared in the literature. We offer as follows.

The go-to solutions for authenticated encryption in modern CPU today are the GCM mode [MV04] or the OCB mode [RBBK01]. However, GCM requires complicated polynomial evaluation, which is slow in Yao's (also noted by Wang *et al.* [WAP+19]), and OCB requires inverse cipher, which is also slow.

Thus, our goals is to search for an efficient AE scheme with three properties: (1) minimal calls to the cipher; (2) no call to the inverse cipher; and (3) no complicated Galois field computation. We find two types of AE schemes that can meet such properties:

- The EAX mode [BRW04] by Bellare, Rogaway, and Wagner, which calls the cipher $1 + 2 \times |M|$ times, where $|M|$ is the message size in blocks. For $|M| = 1$, it takes only 3 calls.
- The OTR mode [Min14; BS16] by Minematsu and its unpatented counterparts discovered by Hoang, Katz, and Malozemoff through automated synthesis [HKM15], which call the cipher $2 + 2 \times \lceil \frac{|M|}{2} \rceil$ times, no inverse cipher needed.

Metal's capability description can fit in one AES block; for this case, EAX makes fewer calls to the block cipher, so AES-EAX is most suitable for AE in Metal-AC.