Titanium: A Metadata-Hiding File-Sharing System with Malicious Security

Weikeng Chen
DZK/UC Berkeley
weikeng@dzk.org

Thang Hoang
Virginia Tech
thanghoang@vt.edu

Jorge Guajardo
Robert Bosch RTC
jorge.guajardomerchan@us.bosch.com

Attila A. Yavuz
University of South Florida
attilaayavuz@usf.edu

***Abstract*—End-to-end encrypted file-sharing systems enable users to share files without revealing the file contents to the storage servers. However, the servers still learn metadata, including user identities and access patterns. Prior work tried to remove such leakage but relied on strong assumptions. Metal (NDSS '20) is not secure against malicious servers. MCORAM (ASIACRYPT '20) provides confidentiality against malicious servers, but not integrity.**

Titanium is a metadata-hiding file-sharing system that offers confidentiality and integrity against malicious users and servers. Compared with MCORAM, which offers confidentiality against malicious servers, Titanium also offers integrity. Experiments show that Titanium is 5× to 200× faster or more than MCORAM.

I. Introduction

Many companies offer cloud storage with end-to-end encryption, such as BoxCryptor [1], Icedrive [2], Keybase Filesystem [3], MEGA [4], pCloud [5], PreVeil [6], Sync [7], and Tresorit [8]. The enthusiasm in end-to-end encryption stems from the public's concerns about how personal data is misused [9] and how hackers have stolen databases of large enterprises [10].

However, end-to-end encryption is not the end, because cloud servers still see metadata. Metadata such as whom the user shares files with is similar to communication privacy—the Stanford MetaPhone study [11] found that phone call metadata is *"densely interconnected, susceptible to re-identification, and enables highly sensitive inferences"*. A former NSA General Counsel said, *"Metadata absolutely tells you everything about somebody's life"* [12].

Extracting secrets from access patterns has been an important area of security research, with much success. There are works that deanonymize users using social connections [13–21], compromise encrypted databases with access patterns [22–28], and break secure hardware with memory access patterns [29, 30]. These attacks might also be applied to end-to-end encrypted file-sharing systems.

To understand the impact of metadata leakage, consider an application (Fig. 1) and how it would benefit from metadata protection. A whistleblower, Alice, wants to report a company's scandal to a journalist. If they communicate via the end-to-end encrypted storage, the servers know that Alice shares files with a journalist and when the files are accessed. If the servers collude with the company, the company may find out the whistleblower.

End-to-end encrypted storage	Metadata-hiding file-sharing systems
Alice and Journalist have accounts	Users remain anonymous
Alice and Journalist share file F • Alice has write permission • Journalist has read permission	F's access control list is unknown
Alice wrote to F on May 26 Journalist read F on May 27	Someone accessed some file on May 26 Someone accessed some file on May 27
⚠ On May 28, the scandal was reported	

Fig. 1: Comparison of security guarantees between end-to-end encrypted storage and metadata-hiding file-sharing systems.

Moreover, a hacker or a malicious employee of the cloud may already know the whistleblower's identity.

Alice and the journalist need a storage system that hides metadata from the servers. This system must have anonymity, so the server does not learn whom it is talking to. It must hide access patterns, so the server cannot infer the user behaviors.

Does such a metadata-hiding file-sharing system exist? In the last decade, researchers have been trying to design practical metadata-hiding storage [31–35]. This is challenging because there is almost nothing to trust: both users and servers can be malicious. We need malicious security.[1]

A. The need for malicious security

The first step toward malicious security is to handle malicious users. For anonymity, there must be many users, and we cannot assume that none of them are malicious. Over the years, security against malicious users has been achieved, as shown in Tab. I.

In contrast, there is no solution to guarantee security against malicious servers. Several constructions [31–33, 35] all assume *semi-honest* servers. A recent work [34] is the closest to this goal, but it does not offer integrity against malicious servers.

Malicious security should be the standard for distributed applications, rather than semi-honest security. This is because malicious attacks can even look *indistinguishable* from honest executions, so the attackers will *never* be caught for behaving maliciously. The possibility of undetectable malicious attacks is concerning to users who need metadata privacy to protect themselves, such as Alice and the journalist.

In §II, we present several malicious attacks on semi-honest constructions. Though these attacks fall beyond the scope of

[1] *Malicious security* ensures security against adversaries who can behave arbitrarily to compromise privacy and integrity of the system. This is in contrast to *semi-honest security*, where adversaries will follow the protocol faithfully.

Network and Distributed Systems Security (NDSS) Symposium 2022
27 February - 3 March 2022, San Diego, CA, USA
ISBN 1-891562-74-6
https://dx.doi.org/10.14722/ndss.2022.24161
www.ndss-symposium.org

TABLE I: Comparison of cryptographic metadata-hiding file storage systems.

Scheme	Security				Functionality		Server overhead	
	Anonymity	Malicious users	Malicious servers	Tolerate t out of N corrupted servers	Multi-owner	File sharing	Computation	Server-server communication
GORAM [31]	No	No	No	$(1,1)$	No	Yes	polylog	N/A
PIR-MCORAM [32]	No	Yes	No		No	Yes	linear	
AnonRAM-lin [33]	Yes	Yes	No		Yes	No	linear	
FHE-MCORAM [34]	Yes	Yes	Partial†		No	Yes	linear	
AnonRAM-polylog [33]	Yes	Yes	No	$(1,2)$	Yes	No	polylog	polylog
Metal [35]	Yes	Yes	No	$(1,2)$	Yes	Yes	polylog	polylog
DPF-MCORAM [34]	Yes	Partial†	Partial†	$(\sqrt{N}-1,N)$ ‡	Yes	Yes	linear	N/A
Titanium (this paper)	Yes	Yes	Yes	$(N-1,N)$	Yes	Yes	polylog	polylog

† FHE-MCORAM does not offer integrity against malicious servers, and DPF-MCORAM does not offer integrity against malicious servers or users.
‡ DPF-MCORAM supports only $N = 4, 9, 16$, i.e., security against 1-out-of-4, 2-out-of-9, or 3-out-of-16 corrupted servers, which is indeed weaker than those under $(N-1, N)$, such as 1-out-of-2, 2-out-of-3, and 3-out-of-4, respectively.

semi-honest assumptions, our goal is to show why malicious security is necessary. Particularly, one of the attacks, ciphertext-substitution attack (in §II-A), can decrypt the entire storage in Metal [35], and the attack is indistinguishable from normal execution, so the attacker will never be caught.

Moreover, integrity, which is not ensured in [34], is critical. Malicious users should not be able to write to other users' files. Malicious servers should not be able to serve incorrect or outdated files without being caught. Achieving integrity in the presence of a malicious adversary is challenging, and sometimes impossible. For example, in any single-server construction, it is impossible to have integrity [36–41] against the malicious server because the server can always present different versions of the files to different users, which we discuss in §II-B.

B. Toward efficient file access

As one may expect, hiding access patterns will incur significant overhead, and we cannot expect it to catch up with existing cloud storage like Dropbox [42]. However, we want it to be at least practical enough so that users who need metadata privacy can use it at a reasonable cost.

The dominating overhead is server computation. As Tab. I shows, some prior constructions have a linear overhead, where they perform *linear passes on the entire storage* to hide access patterns, while others have a smaller overhead.

Linear passing is expensive. For example, in a file-sharing system with ten million files, when the user writes to a file of 16KB, it needs to write at least 150 GB to the disk. Even if the disk is a solid-state drive, it would take twelve minutes.[2] Moreover, solid-state drives naturally cannot sustain such massive write accesses [43, 44]. A common solid-state drive with a lifetime write limit of 150 TB can only sustain 1000 such writes, and then be disposed as e-waste.

For efficiency, it is necessary to avoid linear passes. This requirement rules out the single-server construction due to a lower bound by Maffei, Malavolta, Reinert, and Schröder [32]: in a single-server file-sharing system, hiding file access patterns must incur a linear overhead. This lower bound holds even for the semi-honest server, meaning that the server can simply look at the trace of disk accesses and infer file access patterns unless linear passes over the entire storage are used.

For this reason, our construction, named Titanium, distributes trust across multiple servers like [33–35]. Titanium can tolerate up to $N-1$ out of the N servers being maliciously corrupted. This is the best we can achieve without being subject to the lower bound. Titanium avoids linear passes by using a sublinear oblivious access algorithm on the multi-server model, Circuit ORAM [45], and making improvements that reduce its eviction overhead by up to a half.

[2]Measured on an AWS `gp2` solid-state drive block storage.

C. Titanium's goals and techniques

Titanium offers confidentiality and integrity guarantees against malicious servers and users, as well as a sufficient level of efficiency for sensitive file sharing scenarios. To understand how Titanium achieves these properties, we give a high-level overview of Titanium's techniques, organized by the goals it achieves and our approaches.

Goal 1: security against malicious users. The standard solution to hide access patterns, oblivious RAM (ORAM) [46–53], is inherently *single-user*, meaning that if the ORAM storage is shared with malicious users, the privacy vanishes. To share files securely with many users, new approaches are needed.

• *Approach:* We make Titanium secure against malicious users by *minimizing the users' participation* in the protocol. The only operations that the users perform are sending requests and receiving the responses, through an API we define in §III. The users never touch the data stored on the servers. This approach provides a clean interface and makes it easy to reason about security against malicious users.

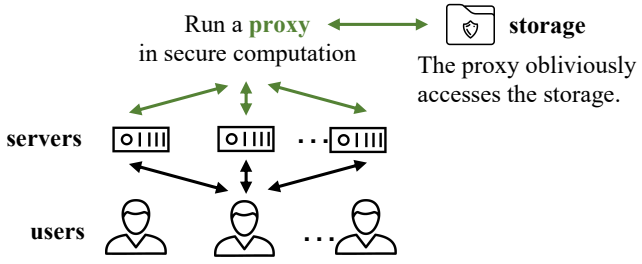


Fig. 2: Titanium's system model.

As Fig. 2 shows, when a Titanium user wants to read or write a file, the user sends a request to the N servers. This request is sent in secret shares so that any $N-1$ servers do not know what the request is. The servers together run a **proxy** in an N-party secure computation, where the proxy's state is hidden from the servers. The proxy checks whether the user's request is legitimate, and if so, on behalf of the user, accesses the storage and responds to the user (through the servers). Now,

malicious users can at most craft an unauthorized request, but the proxy will not process it. By doing so, Titanium achieves malicious security against users.

Goal 2: integrity for data storage. Prior work cannot guarantee integrity for data in the storage against malicious servers. A malicious server can, for example, perform an rollback attack, where a user receives an outdated file from malicious servers. In §II-B, we show that malicious servers (or users) can break integrity in some prior semi-honest constructions.

• *Approach:* We want files to be written only by users who are authorized to write, and wants all the authorized readers to see the latest version of the file. This requires some sort of message authentication code (MAC), over the entire file storage. What is challenging is that such MACs must not reveal user or file identities. Titanium uses authenticated secret sharing [54] to store the data, which hides the MACs from all the parties and thus retains confidentiality. Moreover, at least one of the servers is assumed to be honest. The MACs enable this server to detect if an incorrect version of the file is sent to the user or another file that the user did not request, thereby ensuring integrity, as shown in Fig. 3.

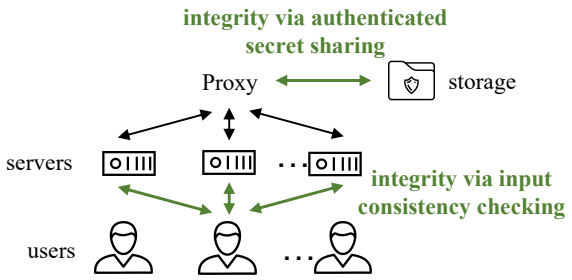


Fig. 3: Providing integrity in Titanium for (1) data in the storage and (2) user inputs and outputs.

Goal 3: integrity for user inputs and outputs. Though authenticated secret sharing provides integrity for the data storage, it does not provide integrity for data sent to and received from the user. If a malicious server can modify such data, a user may read another file, write data into another file, or share a file with a stranger. Attaching a signature does not work because it is not anonymous. An existing primitive designed for the client-server model, secret-sharing non-interactive proofs (SNIP) proposed in Prio [55–57], does not work either because it does not offer integrity against malicious servers.

• *Approach:* We design a maliciously secure input/output protocol between the servers and users, so users can confirm that the **proxy** receives the correct inputs and the user receives the correct outputs from the **proxy**, as shown in Fig. 3. This protocol also provides privacy, as the servers do not see the inputs and outputs. We adopt a tool commonly used in cryptographic proof systems, Schwartz-Zippel lemma [58–60], for this consistency checking.

Goal 4: efficient file access. As Tab. I shows, several prior works [32–34] have a linear server overhead. Though linear-time protocols could sometimes be faster than sublinear protocols, as shown by Floram [61], it is not the case when there are a lot of files. Linear-time protocols may cause a long waiting time for the users and incur an unreasonable amount of cost for the servers.

• *Approach:* Titanium distributes the trust among N servers in a way that it tolerates up to $N-1$ servers being corrupted. This model allows Titanium to avoid the linear lower bound in [32]. Then, Titanium uses an existing sublinear oblivious access protocol, Circuit ORAM [45], and makes improvements to reduce its overhead, as shown in Tab. II. Our improvement removes a significant amount of unnecessary computation in Circuit ORAM, and is equivalent to the original algorithm.

TABLE II: Comparison with the eviction in Circuit ORAM [45] and our improved eviction. The number of AND gates represents the cost of secure computation in boolean circuits.

	Circuit ORAM [45]	Improved (§V)	Improvement
# AND gates (4 KB blocks)	14.2 million	7.7 million	1.8×
# AND gates (16 KB blocks)	55 million	29 million	1.9×

D. Summary of contributions

Our contributions are as follows.

- We study the vulnerabilities of semi-honest file-sharing systems against malicious attackers with concrete attacks (§II).
- We present Titanium, a metadata-hiding file-sharing system with confidentiality and integrity against both malicious servers and users (§VIII).
- We design new protocols for users to communicate with N servers with confidentiality and integrity against malicious servers (Tab. III).
- We propose an optimized algorithm to perform the Circuit-ORAM eviction [45] in secure computation more efficiently (§V). Our method reduces the overhead of Circuit-ORAM eviction in secure computation by up to a half.

II. Why Malicious Security?

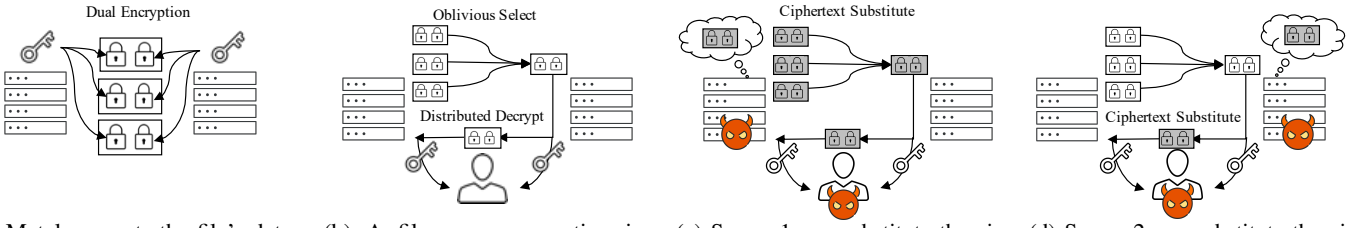
In this section, we discuss why semi-honest security is insufficient in practice, by presenting several classes of attacks. We stress that all these attacks fall beyond the threat models of these semi-honest systems, and are not one of their design goals. We show these attacks to support our statement that malicious security is necessary in practice.

A. Data confidentiality attacks

Semi-honest constructions may be vulnerable against a malicious attacker, which might allow a malicious attacker to break basic security guarantees, such as data confidentiality.

Ciphertext-substitution attack in Metal [35]. Metal assumes two semi-honest servers that do not collude. We show that in Metal, if one of the servers is malicious and colludes with a malicious user, it can learn the content of *any* file of its choice, by substituting ciphertexts in the protocol with the ciphertext that the server wants to decrypt.

As Fig. 4 shows, (a) in Metal, each file is encrypted under two keys, and each belongs to one server; (b) when the user accesses a file, the two servers run an oblivious selection to locate the ciphertext for the file that the user requests, and then perform a distributed decryption over the ciphertext, using their keys; here, Server 1 provides the candidates for the selection and Server 2 receives the selection result and initiates the distributed



(a) Metal encrypts the file's data under two keys, where each server has one of the keys.

(b) A file access operation in Metal consists of oblivious selection and distributed decryption.

(c) Server 1 can substitute the ciphertext in oblivious selection and learns its plaintext.

(d) Server 2 can substitute the ciphertext in distributed decryption and learns its plaintext.

Fig. 4: A data confidentiality attack to Metal [35], in which a malicious server substitutes ciphertexts in the protocol.

decryption over it. Ciphertexts are properly rerandomized so the data access is oblivious.

(c) Server 1 sees all the ciphertexts of the files in the system. If Server 1 wants to decrypt a specific ciphertext C^* (the gray box in Fig. 4), it colludes with a user (which can be a secret account owned by Server 1) and lets the user initiate an arbitrary file access request. During the protocol, Server 1 replaces all the ciphertexts for oblivious selection with C^*. So regardless of the oblivious selection, Server 2 receives C^*, and will run a distributed decryption protocol for C^* with Server 1. The user who colludes with Server 1 receives the decryption of C^* and forwards it to Server 1, which concludes this attack.

(d) Server 2 can perform a similar attack as follows. Server 2 also sees a lot of ciphertexts in Metal. If Server 2 wants to decrypt a specific ciphertext C^*, it colludes with a user and lets the user initiate an access request. After the oblivious selection protocol, Server 2 simply ignores the result of the selection and initiates a distributed decryption of C^*, as in Fig. 4. The user receives the decryption of C^* and forwards it to Server 2.

We stress that this attack is concerning because the malicious attacker will never be caught. Since all the ciphertexts in Metal are re-randomized, they are indistinguishable from one another, and the honest server will never know if the other server is malicious. This limitation may frustrate users, as there is no deterrence for a server to become malicious.

B. Data integrity attacks

Semi-honest constructions may be vulnerable against a malicious attacker who wants to tamper with the honest user's data. Here we present a few examples.

Ciphertext-substitution attack in Metal [35]. The attack in Fig. 4 can be used to make an honest user receive a manipulated copy M^* of the file. When an honest user requests a file, the malicious server simply encrypts M^* to obtain ciphertext C^* and performs the same attack with C^*, so that the honest user receives M^* instead of the actual file data.

Overwriting attack in DPF-MCORAM [34]. In [34], a user writes to a file by sending a distributed point function (DPF) to the servers. A "good" DPF only updates the user's file, but a malicious user can craft a "bad" DPF that modifies someone else's file or overwrites the entire storage with random data. An existing solution, verifiable DPF [62], can only address the latter but not the former and is extremely slow in this setting. Solving this problem may require zero-knowledge proofs on AES operations, which is costly. In addition, since each writing operation in [34] changes the entire storage on each server, there are unlikely any frequent backups of the storage, and such attacks may make data unavailable to honest users, which affects data availability.

Rollback attacks in single-server constructions. There is an impossibility result [36–41] saying that single-server storage systems cannot offer integrity guarantees against the malicious server, as the server can always present an old version of the storage to certain users. A separate system, either another server [39] or blockchain [40, 41], has to be used to recover such integrity guarantees. In Titanium, we prevent rollback attacks by authenticated secret sharing that tolerates up to $N-1$ out of N servers being malicious. That is, as long as one honest server has the authentication tag of the storage if other (malicious) servers provide outdated versions, the integrity check will fail with overwhelming probability.

C. Metadata confidentiality attacks

Selective-failure attacks [63–68] enable an attacker to learn a small amount of metadata based on whether a failure happens. When a malicious server deviates from the protocol, it can observe whether the user receives the correct data or not (through side information). If the user receives the incorrect data and behaves differently, it is considered a *failure*. Whether or not a failure happens may depend on the metadata, so a malicious server can learn some metadata using this attack. We now give two examples of selective-failure attacks in existing systems.

Selective-failure attack in Metal [35]. When Server 1 participates in oblivious selection in Fig. 4, Server 1 can replace the first K blocks with dummy data. If a failure happens, it means that the user is reading a file that has been accessed *recently*, which is a small metadata leakage. This is because Metal uses a tree-based ORAM construction, where the top of the tree often stores files that are accessed recently.

Selective-failure attack in PIR-MCORAM [32] and FHE-MCORAM [34]. In a single-server construction that uses private information retrieval (PIR), a malicious server can perform the PIR over a database in which a subset of the data is replaced with dummy. If a failure happens, it means that the user is reading a file that belongs to the modified subset, which is a small metadata leakage. This issue is common in single-server constructions based on PIR.

III. Overview

In this section, we define Titanium's system model, threat model, and out-of-scope leakages and assumptions.

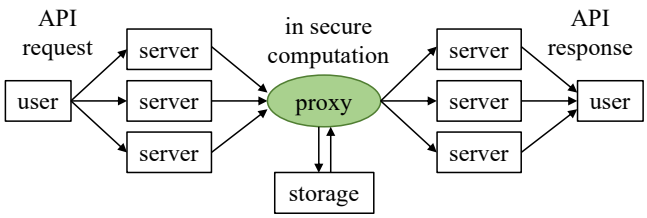


Fig. 5: The workflow of Titanium.

A. System model

We consider a file-sharing system with many users and N servers. The servers collaboratively provide storage services to the users. Each user can store a number of files on the servers and share these files with other users, under some access control policies. Each user can read or write a file that the user has permission to.

As Fig. 5 shows, for a user to perform an operation in Titanium, the user first makes an API request to the servers. The request is in secret shares, so any $(N-1)$ servers cannot recover what is in the request. The N servers, upon receiving the user's request in secret shares, forward the requests to the proxy. The proxy is not a separate entity in the system but is a program executed in secure computation by the N servers. Since the proxy's state is hidden from the servers, we present it as a separate part for ease of presentation. The proxy checks (in secure computation) if the user has permission to perform the action, and if so, interacts with the storage on behalf of the user, such as reading a file or writing to a file.

Next, the proxy sends the API response to the user and asks the servers to forward the response. The response is forwarded also in secret shares, so any $(N-1)$ servers do not see what is in the response. The user reconstructs the proxy's response from secret shares, which concludes an API call in Titanium.

Titanium's API. Users in Titanium interact with the servers through the API defined as follows.

- $\texttt{CreateAccount}() \to (\texttt{uid}, \texttt{credential})$. A new user joins the file-sharing system by calling this API. If the system has capacity for the new user, the user will be assigned a user ID and the corresponding credential, which is used later by the proxy to authenticate the user.
- $\texttt{CreateFile}(\texttt{credential}) \to \texttt{fid}$. Each user can create a new file by calling this API. If the system has capacity, the proxy assigns a new file, and the user obtains the file ID and becomes the owner of this file. The user can grant and revoke permission of the files to other users.
- $\texttt{Read}(\texttt{credential}, \texttt{fid}) \to \texttt{data}$. A user can request the latest version of a file that the user has read permission to by calling this API. The proxy checks if the user has permission using the credential. If the user has permission, the proxy responds to the user with the file data.
- $\texttt{Write}(\texttt{credential}, \texttt{fid}, \texttt{data}^*) \to \perp$. A user can update a file with new data that the user has write permission to by calling this API. The proxy updates the file accordingly.
- $\texttt{Grant}(\texttt{credential}, \texttt{fid}, \texttt{uid}, \texttt{perm}) \to \perp$. The owner of a file can grant permission (defined as "read" and "write") to another user, given that the owner knows the other user's ID, by calling this API. The proxy checks that the caller of the API is indeed the owner of the file and modifies the access control policies of the file accordingly.
- $\texttt{Revoke}(\texttt{credential}, \texttt{fid}, \texttt{uid}, \texttt{perm}) \to \perp$. Similarly, the owner of a file can revoke permission previously granted to another user. The proxy checks the caller's ownership of the file and updates the file's access control policies.

Toward more privacy. In Titanium, the user and file IDs are hidden to the servers and to any users that the honest user does not interact with. However, the API above requires the owner to know the recipient's user ID before sharing, and a recipient of a file knows the file ID after being granted the permission. For more privacy, the user and file IDs can be *hidden* from these users by replacing these IDs with "randomized user ID" and "randomized file ID". A user can have many randomized user IDs that can be provided to a file owner to gain permission, while the owner cannot link this randomized ID to the other ID that a user has. Similarly, the owner does not need to provide the unique file ID to each other user who has access to the file, but a randomized file ID suffices.

Such randomization has been done and formalized in [35], where randomized user and file IDs are called *anonyms* and *capabilities*, respectively. For ease of presentation, Titanium's API does not specify these IDs to be randomized, but Titanium can support it by adopting these primitives (which are pretty lightweight) directly from [35].

B. Threat model

Titanium's threat model is as follows. Up to $N-1$ out of the N servers can be malicious and collude with one another. We assume at least one server is honest, which does not collude with any corrupted servers. All the corrupted servers can arbitrarily deviate from the protocols.

Titanium can tolerate an arbitrary number of users to be malicious and collude with one another and with the corrupted servers. Malicious users may, for example, attempt to access the data of honest users even though they do not have permission. For honest users to remain anonymous from the servers in the system, they are expected to use some sort of anonymity network where the IP addresses and communication patterns, such as latency, do not reveal the user identities. Tor [69] can be one of such solutions, but Titanium can also work with other anonymous communication solutions.

Metadata-hiding properties. Titanium offers all the metadata-hiding guarantees covered in existing works, modeled as follows. Let $\mathcal{A}$ denote the adversary that controls all the corrupted parties. Let U be the honest user who has access to file F.

(a) *Data secrecy and integrity.* If U has never granted read or write permission of F to a malicious user corrupted by $\mathcal{A}$, then $\mathcal{A}$ neither learns anything about F nor modifies F.

(b) *Read obliviousness.* $\mathcal{A}$ cannot distinguish which data block was read by U, even if $\mathcal{A}$ has full permission on the entire storage. That is, $\mathcal{A}$ cannot distinguish a read operation from another read operation, by another honest user, to another file.

(c) *Write obliviousness.* If U has never granted the read permission of F to a malicious user corrupted by $\mathcal{A}$, then $\mathcal{A}$ will not realize if F is updated. But if someone among the corrupted users has read permission to F, they legitimately learn that F has been changed, but if at least two honest users have write permission to F, $\mathcal{A}$ does not know who has changed it.

(d) *Read/write indistinguishability.* If no one in $\mathcal{A}$ has read permission on F, then $\mathcal{A}$ does not know whether an honest user reads or writes to F.

(e) *Anonymity.* $\mathcal{A}$ cannot distinguish which honest user made the access requests, if the honest users communicated with the servers in a way that hid network information.

Formalization. We define the security of Titanium in the real-ideal paradigm [70], and show that Titanium securely realizes an ideal functionality, shown in Fig. 6, in Appendix A.

Definition 1. *A protocol Π is said to securely realize $\mathcal{F}_{\mathsf{FileSharing}}$ in the presence of static malicious adversaries that compromise up to $N-1$ out of N servers, if, for every non-uniform probabilistic polynomial time adversary $\mathcal{A}$ in the real world, there exists a non-uniform probabilistic polynomial time simulator $\mathcal{S}$ in the ideal world, such that the outputs of the two worlds are computationally indistinguishable.*

Out-of-scope leakages and assumptions. In this paper, we make the following assumptions: (1) DoS attacks are out-of-scope; (2) the size of a file is fixed, so there is no size leakage; (3) all the requests are processed in sequential order. These are also standard assumptions in prior work [34, 35, 45, 71–73].

There are some solutions to partially remove these assumptions. To mitigate DoS attacks from users, anonymous payment or client puzzles can be used (see Appendix B). For size leakage, to our knowledge, there is no efficient way to prevent it without padding to the largest file size (which is extremely costly), but there is mitigation including partial padding [74], differential privacy, delayed accessing by downloading different chunks of the file at different times, or only accessing part of the file that is needed. Finally, to our knowledge, it is unclear how to enable parallel oblivious access without requiring a strong assumption [75–78]. Thus, we leave it as an open-research problem.

Ideal functionality $\mathcal{F}_{\mathsf{FileSharing}}$

(For ease of presentation, we omit the session sid "sid" below.)

Initialize. Upon invocation, it sends (`init`) to $\mathcal{S}$. $\mathcal{F}_{\mathsf{FileSharing}}$ maintains the following maps:
- a credential-user map $\mathcal{C}$ from `credential` to user ID `uid`
- a file-storage map $\mathcal{D}$ from `fid` to file data
- a file-permission map $\mathcal{P}$ from (`fid`, uid) to a permission attribute set $p \subseteq \{$'`read`', '`write`', '`own`'$\}$.

CreateAccount. If the system has capacity for the new user, it creates a random credential, sets $\mathcal{C}$[`credential`] := `uid`, sends (`uid`, `credential`) to the user, and sends a signal (`create-account`) to $\mathcal{S}$.

CreateFile. If the system has capacity for the new file, it creates a new file with ID `fid`, assigns the user with owner permission $\mathcal{P}$[`fid`, `uid`] := $\{$'`read`', '`write`', '`own`'$\}$, and sends a signal (`create-file`) to $\mathcal{S}$.

Read. If the user with ID `uid` and correct credential sends a request to read a file with ID `fid`, it checks if '`read`' $\in \mathcal{P}$(`fid`, `uid`). It returns the data $\mathcal{D}$[`fid`] (or dummy if the user is not authorized) to the user and a signal (`access`) to $\mathcal{S}$.

Write. If the user with ID `uid` and correct credential wants to write new data to a file with ID fid, it checks if '`write`' $\in \mathcal{P}$(`fid`, `uid`), and if so, it updates $\mathcal{D}$[`fid`] and sends a signal (`access`) to $\mathcal{S}$.

Grant. When the user with ID `uid` and correct credential wants to grant read or write permission of file with ID `fid` to another user `uid`*, it first checks if '`own`' $\in \mathcal{P}$(`fid`, `uid`), and if so, adds the corresponding permission to $\mathcal{P}$(`fid`, `uid`*) and sends a signal (`grant`) to $\mathcal{S}$.

Revoke. When the user with ID `uid` and correct credential wants to revoke read or write permission of file with ID `fid` to another user `uid`*, similarly, it first checks if '`own`' $\in \mathcal{P}$(`fid`, `uid`), and if so, drops the corresponding permission from $\mathcal{P}$(`fid`, `uid`*) and sends a signal (`revoke`) to $\mathcal{S}$.

Fig. 6: The file-sharing ideal functionality $\mathcal{F}_{\mathsf{FileSharing}}$.

IV. Making the Proxy's Access to the Storage Maliciously Secure

In this section, we describe the instantiation of the Titanium storage and proxy and how they are made maliciously secure.

A. File storage with authenticated secret sharing

Data in the file storage is secret-shared among the N servers using authenticated secret sharing [79]. In Titanium, we represent the file storage as elements in a field $\mathbb{F}$. Each of the N servers has a share of every field element x, and the sum of these shares equals x, as illustrated below.

$$\sum_i x^{(i)} = x \in \mathbb{F}$$

where $x^{(i)}$ is the share of x held by the i-th server. The secret-sharing of a field element x effectively hides the value of x from the servers, thereby providing confidentiality.

Each field element is authenticated by a MAC $m = \alpha \cdot x$, where α is the MAC key. The MAC is secret-shared among the N servers in a similar way. Each server has a share of m, and the sum equals m, as illustrated below.

$$\sum_i m^{(i)} = m \in \mathbb{F},\ m = \alpha \cdot x \in \mathbb{F}$$

The MAC key, α, is also secret-shared among the N servers. This allows them to perform a few operations over these secret shares. For example, the N servers can work together to add, subtract, or multiply two secret field elements that are represented in secret shares, and the computation results are also in secret shares. The servers can also perform integrity checks on the computation results. Therefore, when a malicious server manipulates the results of the computation, an honest server among the N servers can detect such manipulation and refuse to release the incorrect results to the user. As a result, when a user in Titanium receives the response from the N servers, the user is assured that an honest server has checked the response.

In Titanium, the proxy performs two basic types of computation over the file storage. The first is oblivious data selection, in which the proxy is given two field elements and wants to select one of them in secure computation. The proxy performs such selection in an oblivious manner so that the servers running the secure computation do not know which one is being chosen. Each selection operation incurs an overhead, so for efficiency, we want to do as few selections as possible. This is reflected in Titanium's effort to reduce the overhead of Circuit ORAM in §VIII, which minimizes the number of data selections.

The second is to compute a random linear combination of a (large) number of field elements, which is used in our input/output checking protocol described in §VI. This can be done efficiently, which the input/output checking protocol leverages.

B. Running the proxy in secure multiparty computation

In Titanium, the proxy receives the API request from users, accesses the file storage, and sends the API response to the users. This implies that the proxy knows all the secret information in Titanium, and therefore its state and execution must be hidden from the servers and users. To do so, Titanium runs the proxy in an N-party secure computation, which ensures that up to $N-1$ out of the N servers cannot see what is being executed in the secure computation, and if malicious servers want to manipulate the results, the honest servers can detect such discrepancy and refuse to provide the incorrect results to the users.

Secure multiparty computation [70, 80–85] enables N parties to evaluate a function $f(x_1, \ldots, x_N)$ where the i-th party provides input x_i. The results, denoted by $(y_1, \ldots, y_N)$, are released to the parties, such that the i-th party receives y_i. This allows the proxy to have its private state.

Secure computation comes with an overhead, as it runs much slower than plaintext computation. Therefore, Titanium must minimize the amount of computation. The dominating part of the computation is the *data selection* during the oblivious accesses. We discuss how we reduce this overhead in §VIII.

In practice, we can alleviate the user from waiting for the secure computation to finish and receiving the API response, by having the N servers precompute some of the proxy's secure computation before a user sends an API request and use the precomputation to run the secure computation faster. We analyze how this can be useful for users in §IX.

V. Making the Proxy's Access to the Storage Oblivious and Efficient

In this section, we describe how we make the proxy in Titanium access the file storage obliviously and efficiently.

- The first requirement, obliviousness, means that the servers should not know which file is being accessed or which users have access to this file. This is necessary to hide the metadata, as we describe in §III-B.
- The second requirement is efficiency, which is to make the Titanium protocol practical enough for certain use cases where such a high level of privacy is needed.

Titanium makes the following efforts to fulfill these two requirements. Titanium leverages a state-of-the-art oblivious RAM protocol, Circuit ORAM [45] and then improves its eviction procedure (the main overhead) for better efficiency, which achieves an improvement of up to $2\times$.

A. Background on Circuit ORAM

We now present some necessary background of Circuit ORAM for readers to understand how our improvement works.

Circuit ORAM enables the proxy in Titanium to access the file storage, without revealing which file is being accessed. To do so, Circuit ORAM assigns a random location for each file, and when a file is being accessed, it is assigned to a new random location. As a result, for any sequence of file accesses, the locations being accessed on the physical storage are random and do not depend on which files are being accessed.

The challenge that Circuit ORAM faces is to *securely* and *efficiently* move the file to a new location, after each access. First, the moving of the file must be done in a way that hides the new location, otherwise, the servers can associate the old and new locations. Second, the moving must be done efficiently, meaning that it should ideally access only a few locations on the physical storage. A linear pass of the entire storage, in which security can be achieved trivially, is too expensive.

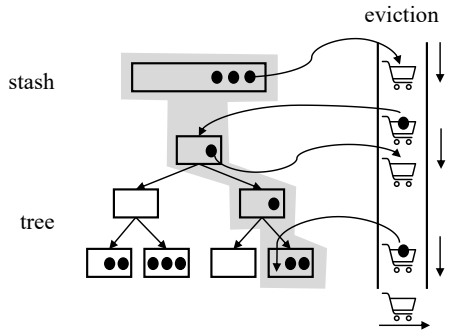


Fig. 7: A "small shopping cart" illustration of how the Circuit ORAM eviction works.

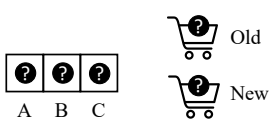
The solution is to use layers of "write caches" for the storage, and instead of moving the file to the new location, the file is first moved to the write cache. These caches are, during the subsequent accesses to the storage, gradually being evicted to lower levels of caches, and eventually to the actual locations where the file should reside. For this reason, when the proxy wants to read a file, the proxy also needs to look at the write caches, as the file may have not yet been evicted out from layers and layers of write caches.

We illustrate the write cache that Circuit ORAM uses in Fig. 7, which follows a binary tree structure, of k levels. The leaf layer of the tree contains 2^{k-1} buckets, where each bucket can store Z files (often $Z = 3$). All the other layers, including the root, are write caches for their descendants. An extra array, called stash, is the top level of the write caches. Readers who are familiar with CPU architecture can consider the stash as the L1 cache, the root of the tree as the L2 cache, and the layer immediately before the leaf layer as the L$(k+1)$ cache.

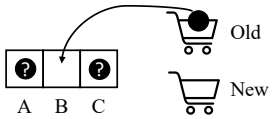
A problem that write caches must handle is space, since the stash and each bucket in the tree has a limited size, files in the write cache must be relocated to lower layers of the cache, or the leaf layer of the tree, to release some space for higher layers of the cache. Circuit ORAM provides an eviction procedure for this purpose. Such eviction is in essence similar to cache replacement for CPU.

This eviction procedure can be illustrated with "small shopping cart", as in Fig. 7. Circuit ORAM chooses a random path of the tree, illustrated in a gray background in the figure, and does a pass over all the caches on this path. The algorithm picks a file from the top level of the cache, puts it into a "shopping cart" that is so small that it can only carry at most one file, and can choose to pick or drop files along the way. A file can move to any write cache that is its ancestor. The shopping cart is empty in the end, and write caches along the path are updated.

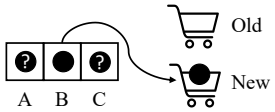
So far, Circuit ORAM sounds like an efficient algorithm, but in order to achieve obliviousness, the shopping cart must update the buckets along the path, in a way that hides what kind of movement is done. This requires the algorithm, running inside the proxy in secure computation, to perform the same data operations regardless of the eviction plan, and therefore it needs to pad the actual eviction plan with a lot of dummy



nothing happens



drop an element

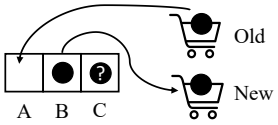


pick up an element

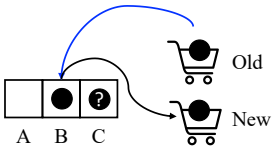
Circuit ORAM (6 conditional data operations)		
`cond_set(new <= A, false)`	`cond_set(new <= A, false)`	`cond_set(new <= A, false)`
`cond_set(new <= B, false)`	`cond_set(new <= B, false)`	`cond_set(new <= B, true)`
`cond_set(new <= C, false)`	`cond_set(new <= C, false)`	`cond_set(new <= C, false)`
`cond_set(A <= old, false)`	`cond_set(A <= old, false)`	`cond_set(A <= old, false)`
`cond_set(B <= old, false)`	`cond_set(B <= old, true)`	`cond_set(B <= old, false)`
`cond_set(C <= old, false)`	`cond_set(C <= old, false)`	`cond_set(C <= old, false)`
Improved version (3 conditional data operations)		
`cond_swap(A <=> old, false)`	`cond_swap(A <=> old, false)`	`cond_swap(A <=> old, false)`
`cond_swap(B <=> old, false)`	`cond_swap(B <=> old, true)`	`cond_swap(B <=> old, true)`
`cond_swap(C <=> old, false)`	`cond_swap(C <=> old, false)`	`cond_swap(C <=> old, false)`
`new <= old`	`new <= old`	`new <= old`

Fig. 8: Comparison of the eviction procedure in Circuit ORAM and the improved version. Part 1: the simple cases.

drop at the pickup location



pick and drop an element in Circuit ORAM



pick and drop an element in the improved version

Circuit ORAM (6 operations)	Improved version (3 operations)
`cond_set(new <= A, false)`	`cond_swap(A <=> old, false)`
`cond_set(new <= B, true)`	`cond_swap(B <=> old, true)`
`cond_set(new <= C, false)`	`cond_swap(C <=> old, false)`
`cond_set(A <= old, true)`	`new <= old`
`cond_set(B <= old, false)`	
`cond_set(C <= old, false)`	

Fig. 9: Part 2: the case when pick and drop happen in the same level.

For secure computation over boolean data:
`cond_swap(L <=> R, cond)`
– `diff = 000...0`
– `cond_set(diff <= L ⊕ R, cond)`
– `left <= left ⊕ diff`
– `right <= right ⊕ diff`
For secure computation over arithmetic data:
`cond_swap(L <=> R, cond)`
– `diff = 000...0`
– `cond_set(diff <= L - R, cond)`
– `left <= left - diff`
– `right <= right + diff`

Fig. 10: Implementation of `cond_swap`.

data operations. As a result, it becomes the main overhead of the entire oblivious file access operation.

Circuit ORAM, in order to reduce the amount of data operations, intentionally makes the eviction *a single pass along the path* and makes the shopping cart *small*, to reduce the number of fake operations used for padding. Circuit ORAM has been implemented in many libraries [61, 86–94]. We now introduce an improved eviction subroutine that cuts the eviction overhead by up to a half, and this approach has not been explored before in the literature or implemented in any existing libraries.

B. Our improved eviction procedure

One challenge in developing Titanium proxy is to reduce the file access latency. When the file size increases, the eviction overhead in secure computation becomes the dominating cost. An eviction in a store of 2^{20} 4KB files requires conditional data operations on about 1MB of data. For efficiency, it is important to reduce the number of such conditional data operations.

We contribute an improved eviction procedure for Circuit ORAM, which reduces the overhead by up to a half, as already shown in Tab. II. The new procedure is general-purpose in that it improves Circuit ORAM in any setting, but the improvement is larger when file sizes are large. The new procedure is equivalent to the original Circuit ORAM algorithm, thereby reusing its security analysis. We believe that existing implementations of Circuit ORAM should use this new procedure.

Our observation is that Circuit ORAM (shown in Fig. 11) is paying unnecessary overhead due to its *modular design*. At each level, eviction consists of "pick" and "drop". When dropping, the original procedure invokes a general-purpose subroutine to conditionally insert the file. However, such a modular design obscures an opportunity for optimization. There is a specialized subroutine, in which "pick" and "drop" do not need to be separate, and their conditional data operations can be combined.

We formally present our eviction algorithm in Fig. 12 and compare it with the original Circuit ORAM algorithm in Fig. 11. But we feel it is easier to understand by illustrating the differences with the "small shopping cart" example again.

As shown in Fig. 8 and Fig. 9, when the shopping cart arrives at a specific layer, it interacts with the bucket that represents the write cache. We present the cart before the interaction as "old" and the cart after as "new". There are four cases: (1) nothing happens, (2) drop an element, (3) pick up an element, and (4) pick and drop an element. In all these cases, Circuit ORAM performs six conditional-set operations, which amount to six data selections. Especially, in the most complicated case in Fig. 9, six seems necessary because "pick" and "drop" may interact with different slots in the bucket, since Circuit ORAM always drops the element in the first availability in each bucket.

```
Get the target array from prior steps.
hold :=⊥, dest :=⊥.
for i = 0 to L do
    towrite :=⊥.
    if (hold ≠⊥) and (i == dest) then
        towrite := hold.
        hold :=⊥, dest :=⊥.
    if target[i] ≠⊥ then
        hold := read and remove deepest block in path[i].
        dest := target[i].
    Place towrite into bucket path[i] if towrite ≠⊥.
```

Fig. 11: The original Circuit ORAM eviction algorithm.

```
Get target, isdeepest, and isfirstempty arrays from prior steps.
hold :=⊥, dest :=⊥.
for i = 0 to L do
    for b = 0 to Z do                                  ▷ Z = 3
        swap := false.
        if target[i] ≠⊥ then                ▷ pick or pick and drop
            swap := isdeepest[i][b].
            dest := target[i].
        else if i = dest then                          ▷ drop only
            swap := isfirstempty[i][b].
        swap path[i][b] and hold if swap is true.
```

Fig. 12: The improved variant of eviction algorithm.

Indeed, the first availability is not a requirement. As the slots in the bucket are equivalent, the file in the old cart can be dropped to any empty slot in the bucket. It happens that after we pick up the file, the place where we pick becomes empty, and is okay to drop the file simply at the location where we pick. With this observation, we can replace the six conditional-set operations with three conditional-swap operations, while each conditional-swap can be efficiently implemented with one conditional-set, for both secure computations based on boolean and arithmetic circuits, as we show in Fig. 10.

The new procedure also appears simpler than the original algorithm. As shown in Fig. 12, the swap conditions can be computed simply from two arrays isdeepest and isfirstempty, indicating whether a slot is deepest and whether it is the first empty in the bucket, which are free byproducts of prior steps in the computation of the eviction plan.

VI. Securing the Proxy's Communication with Users

We now describe the protocol that enables the proxy to communicate with the user securely. The protocols must ensure that malicious servers cannot see the content of such communication or manipulate such communication, as long as at least one of the servers is honest. At the core of this protocol is an efficient batch check protocol that ensures the user receives the same data as what the proxy wants to send. This batch check uses an algebraic tool, Schwartz-Zippel lemma, which is a common tool in cryptographic proof systems.

Batch checking using Schwartz-Zippel lemma [58–60]. Let us consider that the proxy wants to send n field elements to the user, denoted by $s_1, s_2, ..., s_n$. The proxy can sample a random number r and ask the servers to send their secret shares of these field elements to the user, so the user receives $r', s'_1, s'_2, ..., s'_n$ where $r = r'$, $s_i = s'_i$ unless malicious servers have manipulated the data.

Now the user and the proxy want to check if they have the same data. There are many possible ways to do so, such as evaluating a collision-resistant hash function over these field elements. However, since these methods must be evaluated inside the secure computation, they would be much slower than the one based on polynomial identity testing we now present. Titanium lets the proxy and the user each construct a univariate degree-n polynomial using the data.

$$\text{Proxy}: f(x) = r + s_1 \cdot x + s_2 \cdot x^2 + s_3 \cdot x^3 + ... + s_n \cdot x^n$$
$$\text{User}: f'(x) = r' + s'_1 \cdot x + s'_2 \cdot x^2 + s'_3 \cdot x^3 + ... + s'_n \cdot x^n$$

The proxy outputs data to the user, as follows:

1: To send $s_1, s_2, ..., s_n$, the proxy samples a random number r and asks the servers to send shares of r and $\{s_i\}_n$ to the user.
2: The user reconstructs r and $\{s_i\}_n$ from the shares, samples a random $\beta \in \mathbb{F}$, and broadcasts β to all the servers.
3: The proxy receives β from the servers, computes $f(\beta) \leftarrow r + \sum_{i=1}^{n} \beta^i \cdot s_i$, releases $f(\beta)$ to the servers, and asks each server to forward $f(\beta)$ to the user.
4: The user receives $f(\beta)$, computes $f'(\beta)$ locally, and checks if $f(\beta) = f'(\beta)$.

The proxy receives data from the user, as follows:

1: The proxy samples some random elements $r_1, r_2, ..., r_n$ and uses the output protocol to deliver them to the user securely.
2: The user broadcasts $s'_i \leftarrow s_i - r_i$ to all servers.
3: The servers provide $\{s'_i\}_n$ to the proxy, which reconstructs $s_i \leftarrow s'_i + r_i$.

Fig. 13: The maliciously secure input/output protocols.

The user then chooses a random point $x = \beta \in \mathbb{F}$ and tells all the servers this point. If at least one server is honest, the proxy can either receive the correct β or detect a malicious attack (and terminate the protocol). Now, both the proxy and the user knows β, they evaluate this polynomial over point β, and the proxy releases $f(\beta)$ to all the servers.

Each server sends $f(\beta)$ to the user. If at least one server is honest, the user can either receive the correct $f(\beta)$ or detect a malicious attack and terminate the protocol. The user checks if $f(\beta) = f'(\beta)$. If so, the Schwartz-Zippel lemma shows that, with a probability of $1 - n/|\mathbb{F}|$, the two polynomials are the same: that is, $s_i = s'_i$. We present the protocols in Fig. 13.

Reducing the number of client-server rounds with Fiat-Shamir transform [95]. The protocols in Fig. 13 have two client-server rounds for input and output. It can be reduced to one round, with some small amount of additional computation, if one uses Fiat-Shamir transform. In the output protocol, the servers can first commit to the shares of r and $\{s_i\}_{1,..,n}$ they are going to send to the user and broadcast these commitments to each other. The broadcast needs to be made in a way that servers cannot change their messages after seeing someone else's. This is done by requiring the servers to commit to the message first and then open it. Then, they use a cryptographically secure hash function (modeled as a random oracle) to derive β from these commitments and let the proxy compute $f(\beta)$. So, each server now, in addition to sending the shares, also sends the commitments that they receive from each other, the opening of their own commitments, as well as β and $f(\beta)$ to the user.

TABLE III: Comparison with prior works on input/output check.

	[96]	[97]	Ours
Input			
◇ # client-server comm. rounds	1	1	2 or 1
◇ # random shares	$n \cdot N + 3$	$3n$	$n + 1$
◇ # multiplications	1	$2n$	0
Output			
◇ # client-server comm. rounds	1	1	2 or 1
◇ # random shares	$n \cdot N$	$2n$	1
◇ # multiplications	0	$2n$	0

N: number of servers, n: number of inputs/outputs.

If the user receives the same set of commitments from all the servers, the user verifies that β is derived correctly, that commitments are opened correctly, and that $f'(\beta) = f(\beta)$. The input protocol, which uses the output protocol as a subroutine, also reduces to one client-server round.

Comparison with prior works. Securing a client's communication to an entity in secure computation has been explored before, by Jakobsen, Nielsen, and Orlandi [96] and Damgård, Damgård, Nielsen, Nordholt, and Toft [97]. Compared with them, our protocol requires much fewer operations in secure computation, at the cost of one more round (a very minor cost in our use case), or no additional round if Fiat-Shamir transform is used, as shown in Tab. III.

VII. Performing File Access Control in the Proxy

In Titanium, we want to enable the owner of a file to grant permission to another user and to revoke previously granted permission. A user has one of the five types of permission to a file: no permission, read-only, write-only, read-and-write, and ownership. There are several ways to implement this access control without leaking metadata, with different trade-offs, and one may be more suitable than others sometimes. We summarize their pros and cons in Tab. IV.

A. Our main approach: access control matrix

The main approach for Titanium is to store the access control policies in a matrix of size that is the number of users times the number of files, where each entry of the matrix stores a three-bit value representing which permission the user has for this file. This appears to be practical in a few setups. For example, assuming that the number of users equals $\sqrt{\text{number of files}}$, for a system with 2^{24} files, each file stores about 1 KB of the data for a column in the access control matrix. To check and update permission, the proxy simply does a linear scan. This method allows the owner to grant and revoke permission with the same experience as in traditional file-sharing systems. A file can also be shared with a large number of users. For this reason, we use it as the main approach and evaluate it in §IX.

B. Alternative: access control list

If the system has many users, the access control matrix becomes impractical. In practice, most files are shared with only a few users, and to handle a file shared with a lot of users, one may instead share a special "group" account between these users instead of adding each of them into the file's access control list. In this case, we can limit the number of users/groups who share a file to a number d that is much smaller than the total number of users in the system (e.g., $d = 10$), which suffices for many use cases, and the overhead is much smaller.

TABLE IV: Summary of pros and cons of several access control methods that hide metadata.

Method	Pros and cons
Access control matrix (§VII-A)	Pros: efficient revocation, many users can share the same file Cons: large server storage overhead
Access control list (§VII-B)	Pros: efficient revocation; small server storage overhead Cons: a file can only be shared with a few users (or entities)
Capabilities (§VII-C)	Pros: small server overhead; many users can share the same file Cons: users need to store capabilities locally; inconvenient revocation

C. Alternative: capabilities

Another approach, proposed in Metal [35], is, instead of storing access control lists on the servers, the proxy gives users some cryptographic tokens (called "capabilities"), where each token represents permission to a file. To check permission, the proxy checks if the token is valid. This alleviates the servers from storing any access control data. However, in this approach, it is hard to revoke permission: the owner has to invalidate the old file (aka, revoke everyone), create a new file and reshare the permission with any user whose permission is not revoked. The cost increases when the file is shared with many users.

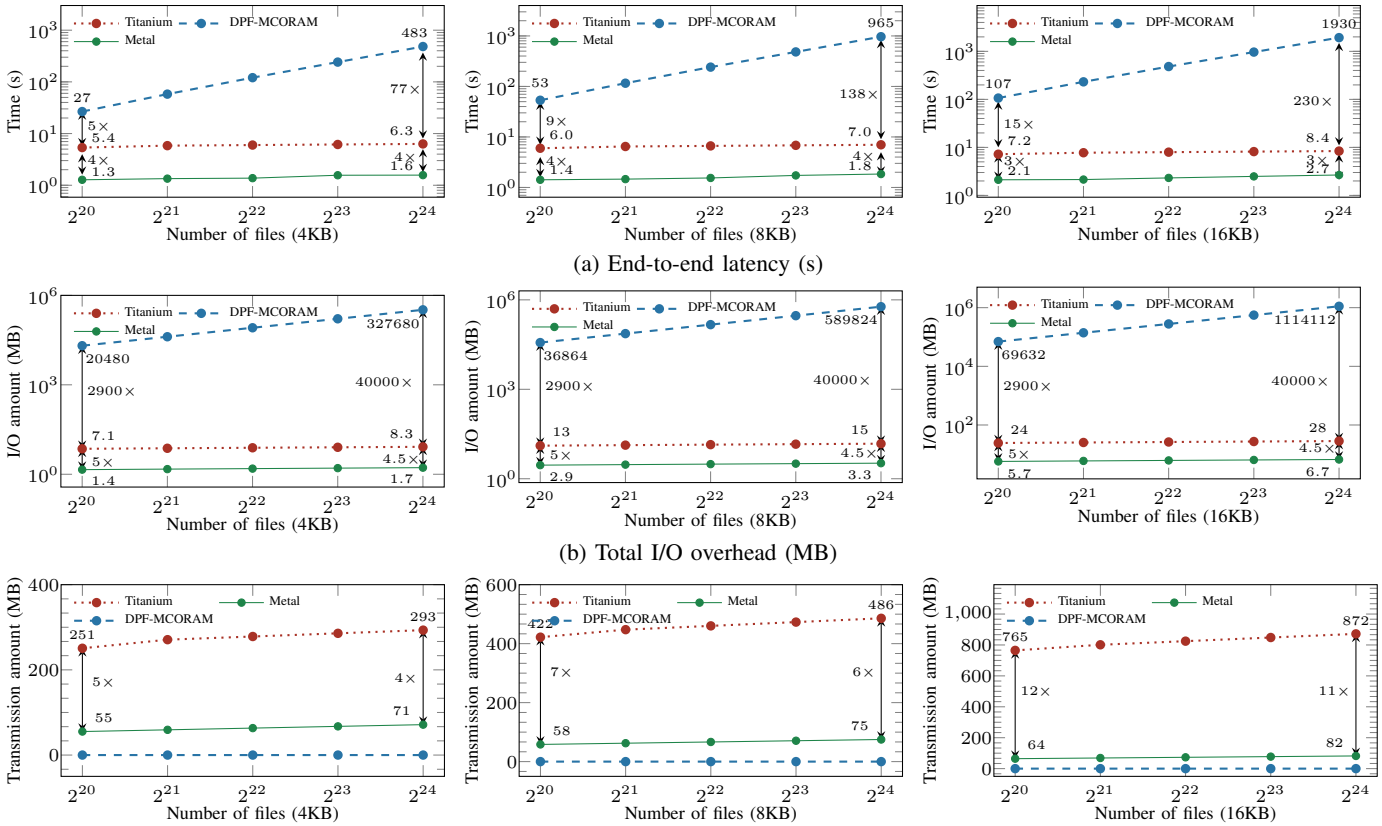
VIII. Putting It Together

We have presented all the components of Titanium and showed how they provide malicious security. In this section, we describe how they are put together as a metadata-hiding file-sharing system. For ease of description, we use the example of Alice and the journalist from §I.

Onboarding. Alice joins the storage system by installing a client software from a trusted source, in which Alice selects a set of N servers who are providing this service and already have a lot of users (for anonymity). Throughout the description, we assume Alice has found a way to hide the network patterns. This includes using the free WiFi services in a café or store, using relays such as Mozilla VPN or Apple Private Relay, using Tor, or a combination of them. The anonymity network does not need to be perfect, as the servers can only know an API request comes from a specific IP address, but do not know what is inside the API request. We assume the client software can block other software on the device from connecting to the Internet during the use of the software, which reduces the risk of de-anonymization due to other software on the device.

Create an account and a file. Alice sends an API request for registration, and the proxy running inside secure computation (§IV-B) on the server-side assigns a user ID and credential to Alice. The journalist joins the system as well. To prepare the materials to be shared with the journalist, Alice creates a file in the storage system via an API call. The proxy assigns a file ID to Alice, and Alice is the owner of the file (§VII). Alice receives the user ID, the credential, and the file ID through the output protocol (§VI) for integrity.

Uploading the file. Alice uploads the file using the client software. The client software makes the API call and transfers



(a) End-to-end latency (s)

(b) Total I/O overhead (MB)

(c) Total inter-server network overhead (MB)

Fig. 14: Comparison between DPF-MCORAM (tolerating 1-out-of-4 corruption), Titanium, and Metal (both tolerating 1-out-of-2).

the data to the proxy using the input protocol (§VI) for integrity. Then the proxy checks if Alice has permission to write to the file, by looking up the access control matrix (§VII), from which the proxy knows Alice is the owner of the file. If so, the proxy accesses the file storage (§IV-A). The file access is followed by an eviction procedure using our improved algorithm (§VIII). In the end, though this is a write request, the proxy still returns a dummy file to make read and write indistinguishable from the servers and network. Alice receives the dummy file using the output protocol (§VI).

Sharing the file. Alice obtains the journalist's user ID through a reliable mechanism, such as from an in-person meeting with the journalist. Alice uses the client software to invoke the API to grant read permission to the journalist, through the input protocol (§VI). The proxy, after checking the integrity of the request and Alice's ownership of the file, modifies the access control matrix to grant the journalist read permission to the file. The journalist, who receives the file ID from Alice in some way, such as through an anonymous broadcast service already shown in Metal [35], can now read the file. To make this read request indistinguishable from a write request, the journalist's client software uploads a dummy file in the input protocol. The remaining operations are similar to writing a file.

Revoking the permission. When the journalist receives the file and reports the scandal, Alice can revoke the journalist's access to the file, as a precaution in case the journalist's account is compromised. To do so, Alice uses the client software to communicate with the proxy and submit the revocation API request using the input protocol (§VI).

In Appendix A, we give proof sketches of the following security theorem for Titanium.

Theorem 1. *Assuming secure multiparty computation and other standard cryptographic assumptions, Titanium securely realizes* $\mathcal{F}_{\mathsf{FileSharing}}$ *against malicious adversaries that corrupt up to* $N-1$ *out of the* N *servers and an arbitrary number of users. For every non-uniform probabilistic polynomial-time adversary* $\mathcal{A}$ *in the real world, there exists a non-uniform probabilistic polynomial time simulator* $\mathcal{S}$ *in the ideal world, such that the outputs of the two worlds are computationally indistinguishable.*

IX. Evaluation

In this section, we answer the following questions:

1) What is the overhead of Titanium?
2) How does Titanium compare with the state-of-the-art?
3) What is the breakdown of the overhead of Titanium?
4) How does the improved ORAM eviction protocol in §VIII compare with the original Circuit ORAM algorithm?
5) If servers can do precomputation, what is the latency that a user in Titanium would experience?

A. Setup

We implemented Titanium and benchmarked its performance using standard libraries for secure computation, including EMP-toolkit [98], SCALE-MAMBA [94], and MP-SPDZ [92]. Specifically, since MP-SPDZ has an efficient offline phase via LowGear, we used it to generate Beaver triples for secure computation in arithmetic circuits. On the other hand, we used SCALE-MAMBA for the online phase due to its support of mixed circuit. The field size for authenticated secret sharing

is 64 bits. Since the users simply interact with the API, our evaluation focuses on the server overhead.

We used `c4.8xlarge` AWS instances for the servers, each with 36 CPU cores and 60 GB RAM. We used Linux `tc` tool to limit the bandwidth of each server to 2 Gbps and added a network round-trip latency of 20 ms between them.

We evaluate on a file storage with 2^{20} to 2^{24} files of size ranging from 4 KB to 16 KB, which we consider to be the practical region of Titanium. As for the access control matrix, we consider $\sqrt{2^{24}} = 2^{12}$ users, the same as in [35].

We note that due to ORAM security, read and write requests in Titanium are provably indistinguishable and incur the same overhead (i.e., memory usage, latency, network communication). In fact, the user who wants to read a file is also performing a dummy write to the storage and vice versa (see §IX-E).

B. Performance of Titanium over prior multi-server schemes

To understand the price that Titanium pays for malicious security, we consider Metal, a two-server semi-honest scheme, as our baseline for comparison. We then compare Titanium with DPF-MCORAM scheme in [34] that offers partial malicious security to showcase our advantages. We compare the case of $(t, N) = (1, 4)$ in DPF-MCORAM with $(1, 2)$ in Titanium and Metal (note that $(1, 2)$ is more secure than $(1, 4)$). Given that the experimental evaluation of DPF-MCORAM is not available, we estimated its overhead using the state-of-the-art library in Express [99] for its distributed point functions (DPF).

Fig. 14 presents the performance of Titanium and its counterparts in terms of end-to-end latency, I/O access, and inter-server communication overhead for each file access request.

As shown in Fig. 14, in a setup with two servers, the overhead of Titanium grows almost polylogarithmically to the number of files. The overhead also grows linearly to the file size. This matches the expectation of the Titanium algorithm: (1) Titanium uses Circuit ORAM, which can access the storage in time sublinear to the number of files; (2) when the proxy performs data operations on the files, the amount of computation naturally grows linearly to the file size.

Compared with Metal, Titanium is approximately $3-4\times$ slower in terms of end-to-end delay. Specifically, Metal takes 1.3 s $-$ 2.7 s to access a file with sizes ranging from 4 KB to 16 KB in storage with 2^{20} to 2^{24} files, while Titanium takes 5.4 s $-$ 8.4 s per access. Titanium also incurs higher I/O access (i.e., around $5\times$) and inter-server network communication (e.g., $5 \times -11\times$) than Metal. This is because Titanium needs to authenticate the data in secure computation, while Metal does not as it only offers semi-honest security.

Compared with DPF-MCORAM, Titanium is several orders of magnitude faster and has lower I/O overhead as shown in Fig. 14. In the log scale, we can see that the overhead of Titanium grows slowly because Titanium's overhead is polylogarithmic to the number of files, while DPF-MCORAM grows linearly to the number of files (so, it is a straight line in the figure). One advantage that DPF-MCORAM offers over Titanium and Metal is that it does not require servers to communicate with each other as shown in Fig. 14c. Despite such advantage, DPF-MCORAM requires linear processing and, as a result, its I/O access, and computation cost is two to three orders of magnitude of Titanium's.

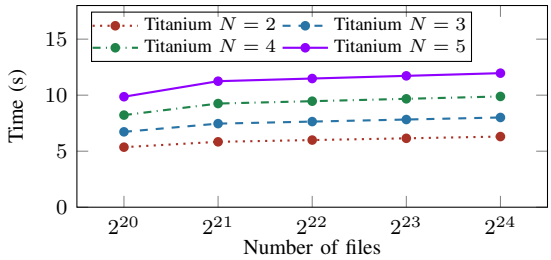


Fig. 15: Overhead with 4KB files and varying number of servers.

C. Experiments with varying numbers of servers

We evaluated the scalability of Titanium with a varying number of servers. Fig. 15 presents the access latency of Titanium with two to five servers. We can see that the overhead of Titanium grows almost linearly to the number of servers. This matches the expectation as well because the cost of each operation that the proxy is performing grows linearly to the number of servers running the secure computation. The linear behaviors also show a trade-off between security and efficiency. To reduce metadata leakage for large files, it is preferred to increase the file size limit, but this would increase the overhead linearly. To better distribute the trust among the servers, it is preferred to increase the number of servers, but this would also increase the overhead. In practice, a user chooses a specific Titanium system depending on the trade-off that the user would like to make.

Titanium is more scalable than DPF-MCORAM when increasing the number of servers for higher corruption tolerance. Remark that DPF-MCORAM needs N servers to tolerate $t = \sqrt{N}$ corruptions while Titanium requires N servers for $t = N - 1$ corruptions. As shown in Tab. V, the cases of $(t, N) = (2, 9)$ and $(3, 16)$ in DPF-MCORAM, which are less secure than the cases of $(t, N) = (2, 3)$ and $(3, 4)$ respectively in Titanium, are very expensive. This is due to the high overhead of running DPF inside homomorphic secret sharing (which is commonly implemented with leveled fully homomorphic encryption) over a large amount of data. We use Gentry, Halevi, and Smart's implementation [100] on running ciphers in homomorphic encryption to approximate these numbers. Overall, compared with DPF-MCORAM, Titanium has the following advantages: (1) Titanium provides integrity against malicious servers, (2) Titanium does not do linear passes, (3) with N servers, Titanium tolerates $N - 1$ corrupted servers, while DPF-MCORAM tolerates $\sqrt{N}$, (4) Titanium can support many servers, while DPF-MCORAM is restricted to $N = 4, 9, 16$.

Note that we did not compare Titanium against Metal in this experiment because Metal is designed for the semi-honest setting and is restricted to the two-server model.

TABLE V: More comparison with DPF-MCORAM.

	DPF-MCORAM (2,9) or (3,16)	Titanium (2,3)	Titanium (3,4)
2^{20} files	0.6 years	6.7 s	8.2 s
2^{21} files	1.3 years	7.5 s	9.3 s
2^{22} files	2.6 years	7.6 s	9.5 s
2^{23} files	5.1 years	7.8 s	9.7 s
2^{24} files	10.2 years	8.0 s	9.9 s

D. Comparison with single-server counterparts

We compare Titanium with two notable state-of-the-art single-server counterparts including PIR-MCORAM [32] and FHE-MCORAM [34]. As discussed previously, the single-server model means that the system must incur a linear overhead and cannot provide full integrity against malicious servers.

Since PIR-MCORAM does not have an open-source implementation, we extrapolate its results reported in [32] and estimate that it would take at least 100 s for the same file access in our setting.

Though there is neither evaluation nor implementation of FHE-MCORAM, we can estimate a lower bound of its overhead based on the FHE cost. To make FHE bootstrapping efficient, parameter choices are important. An efficient instantiation is shown by Halevi and Shoup [101], using packed FHE ciphertexts on a specific cyclotomic ring, so that the per-plaintext-bit cost of bootstrapping is small. We estimated that the overhead of single file access in FHE-MCORAM, for a store of 2^{20} files of size 4 KB, would take 55 days, and for 2^{24} files of size 16 KB, it would be about nine years.

E. Cost analysis

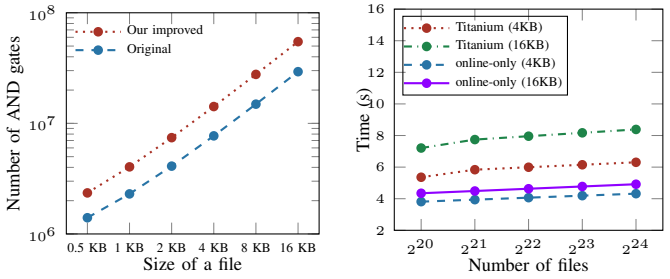
We perform a cost breakdown analysis to investigate how each processing phase impacts the performance of Titanium. Tab. VI presents the detailed costs of Titanium in terms of end-to-end latency, memory usage, and communication when reading or writing a 4KB file in storage with 2^{20} files.

Due to Circuit ORAM, for each file request, Titanium incurs two major processing phases: retrieval and eviction. The retrieval is to read a file from the storage while the eviction is to write a file to the storage. Any file request from the user incurs both retrieval and eviction processing at the proxy. For read requests, the retrieval phase reads the requested file from the storage, while the eviction phase writes the retrieved file back to the storage. For write requests, the retrieval phase reads the file from the storage but ignores the results, while the eviction phase writes the file with new data to the storage. Therefore, any file request in Titanium incurs the same processing overhead regardless of whether the type of the request is read or write. The proxy decides whether the user sees the file (in the case of "read") and whether the user's input becomes the new data of the file (in the case of "write") using oblivious selection over the request type indicated by the user.

As shown in Tab. VI, we can see that eviction is the most dominating part, especially in end-to-end latency where it contributes more than 80% to the total cost. It is because the eviction needs to compute a complicated eviction plan in secure computation and perform more data operations. The inter-server communication and I/O costs of the eviction phase are also higher than that of the retrieval phase. This is because the eviction in Circuit ORAM performs about twice the number of operations on the files compared with the retrieval phase.

TABLE VI: Cost breakdown of Titanium per file request in storage with 2^{20} files and file size of 4 KB. Read and write requests incur the same amount of processing overhead.

	Latency (s)	Memory usage (MB)	Inter-server comm. (MB)
ORAM retrieval • read request: read the requested file • write request: read but ignore	0.53	1.8	108
ORAM eviction • read request: evict the file back • write request: evict with new file	2.7	5.3	142



(a) Eviction (b) User waiting time

Fig. 16: (a) Our improved eviction vs. original one; (b) User waiting time in Titanium with vs. without precomputation (b).

F. Improvement of ORAM eviction procedure

To understand how the new algorithm in §VIII improves over Circuit ORAM, we evaluate their costs in the eviction procedure. As shown in Fig. 16a, the computation cost, represented by the number of AND gates in secure computation based on boolean circuits, grows linearly to the file size. For all the file sizes that we consider, the improved protocol performs better than the original protocol. This improvement varies by file size. When the file size is 0.5 KB, the improvement brought by §VIII is 1.7×, and when the file size is 16 KB, the improvement becomes 1.9×. This is because in Circuit ORAM, besides the data operations, there is a cost to compute the eviction plan, which is independent of the file size. When the file is small, there is still an improvement but is smaller because the cost of computing the eviction plan remains a significant part. When the file is large, the data operations dominate the overhead of eviction, and the improvement gets closer to 2×.

G. User waiting time given precomputation

The main overhead of Titanium is the proxy's operations, which are computed in the N-party secure computation, by the N servers. A common approach to reducing the running time of secure computation is to do precomputation—when there are no API requests, the servers themselves precompute part of the secure computation, so later when a user wants to access a file, the servers can run the proxy with such precomputation, so the running time is shorter. However, the precomputation is one-time. If there is insufficient precomputation, the user will need to wait for the original execution of the proxy. In practice, this is still useful in that it can reduce the user waiting time when the storage system is not overly loaded. Precomputation can also be done by additional machines.

Precomputation is useful for a few Titanium API functions, specifically reading a file. As shown in Fig. 16b, with two servers, the online-only version, which uses precomputation, reduces the waiting time. The saving is larger with larger file size. For example, the online-only version can reduce the waiting time by 30% when the file size is 4 KB. It becomes 40% when the file size is 16 KB. This is because when the file size is small, the network latency, which precomputation does not help, contributes more to the user waiting time.

X. Related Work

We now discuss the related work, organized into four categories.

Revealing secret information from patterns. Several areas of security have been exploring how to learn secret information from patterns. We know social connections can be used to reveal identities in the social network [13–21], encrypted databases may leak access patterns that reveal data [22–28, 102], and secure hardware suffers from side-channel attacks [29, 30].

Single-user oblivious storage. Single-user oblivious storage has been explored in various system settings including single-server model [47, 50–53, 103, 104] and distributed (i.e., multi-server) model [71, 105–108]. Some other works focus on specific demands for privacy, such as hiding only the write patterns [109–111], enabling parallel accesses [72, 77, 78, 112, 113], and building a usable system [75, 114–116].

Sharing oblivious storage among many users. Maffei et al. [32] showed that a single-server multi-user oblivious storage must have at least linear server computation for secure data sharing against client-server collusion. As a result, all single-server constructions are either linear or insecure upon collusion [31, 33, 117]. It is also impossible to achieve integrity against the malicious server in the single-server setting, as the server can always violate the integrity by serving outdated data to some users [36–39, 41]. A few recent works [33–35, 117] explored how to share oblivious storage among many mutually distrusting users by distributing the trust to multiple servers.

Oblivious storage using trusted hardware. There is a line of works using secure hardware for oblivious storage systems, generally built on FPGAs [73, 114] and secure enclaves [74, 76, 118–121]. Although secure hardware-based constructions tend to be very efficient, it requires a strong security assumption on the hardware (e.g., isolation, tamper-resistant, enclaves) unlike Titanium, which only requires the hardware to operate as normal. To hide the access patterns, these protocols often need to perform oblivious operations inside the secure hardware to defend against side-channel attacks [73, 122, 123].

XI. Conclusion

We presented Titanium, a metadata-hiding file-sharing system that offers confidentiality and integrity guarantees against malicious servers and users. We designed new protocols that allow the user to share files efficiently with integrity guarantees. We also introduced a new optimization technique to reduce the cost of Circuit ORAM. Experiments showed that Titanium is one to two orders of magnitude faster than the state-of-the-art construction that offers partial malicious security, while achieving competitive performance with the semi-honest counterpart.

Acknowledgment

Weikeng Chen is supported by NSF CISE Expeditions Award CCF-1730628, NSF CAREER 1943347, and gifts from the Sloan Foundation, Bakar Fellows Program, Alibaba, Amazon Web Services, Ant Group, Ericsson, Facebook, Futurewei, Google, Intel, Microsoft, Nvidia, Scotiabank, Splunk, and VMware. Thang Hoang is supported by the unrestricted gift from Robert Bosch LLC. Attila A. Yavuz is supported by the unrestricted gift from the Cisco Research Award, and the NSF CAREER Award CNS-1917627. 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.

References

[1] "Boxcryptor: No. 1 cloud encryption made in Germany," https://www.boxcryptor.com/en/.
[2] "Icedrive: Secure encrypted cloud storage," https://icedrive.net/encrypted-cloud-storage.
[3] "Introducing the Keybase filesystem," https://book.keybase.io/docs/files.
[4] "Mega: Secure cloud storage and communication," https://mega.io/.
[5] "pCloud encryption: Best secure encrypted cloud storage," https://www.pcloud.com/encrypted-cloud-storage.html.
[6] "Preveil: Encrypted email and file sharing for the enterprise," https://www.preveil.com/.
[7] "Sync: Secure cloud storage, privacy guaranteed," https://www.sync.com/.
[8] "Tresorit: Secure file sharing & content collaboration with encryption," https://tresorit.com/secure-file-sharing.
[9] B. Auxier, L. Rainie, M. Anderson, A. Perrin, M. Kumar, and E. Turner, "Americans and privacy: Concerned, confused and feeling lack of control over their personal information," Pew Research Center (2019).
[10] R. Brandom, G. Blackmon, and W. Joel, "Ten years of breaches in one image: Nearly 8 billion usernames have leaked since june 2011," Available at https://www.theverge.com/22518557/data-breach-infographic-leaked-passwords-have-i-been-pwned.
[11] J. Mayer, P. Mutchler, and J. C. Mitchell, "Evaluating the privacy properties of telephone metadata," in *PNAS '16*.
[12] A. Rusbridger, "The Snowden leaks and the public," in *The New York Review of Books—NYR Daily November 21, 2013*.
[13] C. C. Aggarwal, "On k-anonymity and the curse of dimensionality," in *VLDB '05*.
[14] L. Backstrom, C. Dwork, and J. Kleinberg, "Wherefore art thou r3579x?: Anonymized social networks, hidden patterns, and structural steganography," in *WWW '07*.
[15] A. Narayanan and V. Shmatikov, "Robust de-anonymization of large sparse datasets," in *S&P '08*.
[16] G. Wondracek, T. Holz, E. Kirda, and C. Kruegel, "A practical attack to de-anonymize social network users," in *S&P '10*.
[17] S. Nilizadeh, A. Kapadia, and Y.-Y. Ahn, "Community-enhanced de-anonymization of online social networks," in *CCS '14*.
[18] 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*.
[19] 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*.
[20] G. G. Gulyás, B. Simon, and S. Imre, "An efficient and robust social network de-anonymization attack," in *WPES '16*.
[21] J. Feng, M. Zhang, H. Wang, Z. Yang, C. Zhang, Y. Li, and D. Jin, "DPLink: User identity linkage via deep neural network from heterogeneous mobility data," in *WWW '19*.
[22] M. S. Islam, M. Kuzu, and M. Kantarcioglu, "Access pattern disclosure on searchable encryption: Ramification, attack and mitigation," in *NDSS '12*.

[23] ——, "Inference attack against encrypted range queries on outsourced databases," in *CODASPY '14*.
[24] D. Cash, P. Grubbs, J. Perry, and T. Ristenpart, "Leakage-abuse attacks against searchable encryption," in *CCS '15*.
[25] G. Kellaris, G. Kollios, K. Nissim, and A. O'neill, "Generic attacks on secure outsourced databases," in *CCS '16*.
[26] M.-S. Lacharité, B. Minaud, and K. G. Paterson, "Improved reconstruction attacks on encrypted data using range query leakage," in *S&P '17*.
[27] C. V. Rompay, R. Molva, and M. Önen, "A leakage-abuse attack against multi-user searchable encryption," in *PETS '17*.
[28] L. Blackstone, S. Kamara, and T. Moataz, "Revisiting leakage abuse attacks," in *NDSS '20*.
[29] X. Zhuang, T. Zhang, and S. Pande, "HIDE: An infrastructure for efficiently protecting information leakage on the address bus," in *ACM SIGOPS Operating Systems Review '04*.
[30] T. M. John, S. K. Haider, H. Omar, and M. Van Dijk, "Connecting the dots: Privacy leakage via write-access patterns to the main memory," in *IEEE TDSC '17*.
[31] M. Maffei, G. Malavolta, M. Reinert, and D. Schröder, "Privacy and access control for outsourced personal records," in *S&P '15*.
[32] M. Maffei, G. Malavolta, M. Reinert, and D. Schröder, "Maliciously secure multi-client ORAM," in *ACNS '17*.
[33] M. Backes, A. Herzberg, A. Kate, and I. Pryvalov, "Anonymous RAM," in *ESORICS '16*.
[34] S. S. Chow, K. Fech, R. W. Lai, and G. Malavolta, "Multi-client oblivious RAM with poly-logarithmic communication," in *ASIACRYPT '20*.
[35] W. Chen and R. A. Popa, "Metal: A metadata-hiding file-sharing system," in *NDSS '20*.
[36] D. Mazières and D. Shasha, "Building secure file systems out of Byzantine storage," in *PODC '02*.
[37] J. Li, M. Krohn, D. Mazières, and D. Shasha, "Secure untrusted data repository (SUNDR)," in *OSDI '04*.
[38] A. J. Feldman, W. P. Zeller, M. J. Freedman, and E. W. Felten, "SPORC: Group collaboration using untrusted cloud resources," in *OSDI '10*.
[39] N. Karapanos, A. Filios, R. A. Popa, and S. Capkun, "Verena: End-to-end integrity protection for web applications," in *S&P '16*.
[40] A. Tomescu and S. Devadas, "Catena: Efficient non-equivocation via Bitcoin," in *S&P '17*.
[41] Y. Hu, S. Kumar, and R. A. Popa, "Ghostor: Toward a secure data-sharing system from decentralized trust," in *NSDI '20*.
[42] "Dropbox," https://www.dropbox.com/.
[43] F. Chen, T. Luo, and X. Zhang, "CAFTL: A content-aware flash translation layer enhancing the lifespan of flash memory based solid state drives," in *FAST '11*.
[44] C. Li, P. Shilane, F. Douglis, H. Shim, S. Smaldone, and G. Wallace, "Nitro: A capacity-optimized SSD cache for primary storage," in *ATC '14*.
[45] X. Wang, H. Chan, and E. Shi, "Circuit ORAM: On tightness of the Goldreich-Ostrovsky lower bound," in *CCS '15*.
[46] O. Goldreich, "Towards a theory of software protection," in *CRYPTO '86*.
[47] ——, "Towards a theory of software protection and simulation by oblivious RAMs," in *STOC '87*.
[48] O. Goldreich and R. Ostrovsky, "Software protection and simulation on oblivious rams," in *JACM '96*.
[49] B. Pinkas and T. Reinman, "Oblivious RAM revisited," in *CRYPTO '10*.
[50] E. Shi, T.-H. H. Chan, E. Stefanov, and M. Li, "Oblivious RAM with O $(\log N^3)$ worst-case cost," in *ASIACRYPT '11*.
[51] E. Stefanov, E. Shi, and D. Song, "Towards practical oblivious ram," in *NDSS '12*.
[52] 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*.
[53] 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*.
[54] I. Damgård, M. Keller, E. Larraia, V. Pastro, P. Scholl, and N. P. Smart, "Practical covertly secure MPC for dishonest majority–or: breaking the SPDZ limits," in *ESORICS '13*.
[55] H. Corrigan-Gibbs and D. Boneh, "Prio: Private, robust, and scalable computation of aggregate statistics," in *NSDI '17*.
[56] D. Boneh, E. Boyle, H. Corrigan-Gibbs, N. Gilboa, and Y. Ishai, "Zero-knowledge proofs on secret-shared data via fully linear PCPs," in *CRYPTO '19*.
[57] S. Addanki, K. Garbe, E. Jaffe, R. Ostrovsky, and A. Polychroniadou, "Prio+: Privacy preserving aggregate statistics via Boolean shares," in *IACR ePrint 2021/576*.
[58] J. T. Schwartz, "Fast probabilistic algorithms for verification of polynomial identities," in *JACM '80*.
[59] R. Zippel, "Probabilistic algorithms for sparse polynomials," in *EUROSAM '79*.
[60] R. A. Demillo and R. J. Lipton, "A probabilistic remark on algebraic program testing," in *Information Processing Letters '78*.
[61] J. Doerner and A. Shelat, "Scaling oram for secure computation," in *CCS '17*.
[62] E. Boyle, N. Gilboa, and Y. Ishai, "Function secret sharing: Improvements and extensions," in *CCS '16*.
[63] J. Kilian, "Founding cryptography on oblivious transfer," in *STOC '88*.
[64] M. Naor and B. Pinkas, "Oblivious transfer with adaptive queries," in *CRYPTO '99*.
[65] C. Dwork, M. Naor, O. Reingold, and L. Stockmeyer, "Magic functions," in *FOCS '99*.
[66] J. Camenisch, G. Neven, and A. Shelat, "Simulatable adaptive oblivious transfer," in *EUROCRYPT '07*.
[67] Y. Lindell and B. Pinkas, "Secure two-party computation via cut-and-choose oblivious transfer," in *TCC '11*.
[68] 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-A*.
[69] R. Dingledine, N. Mathewson, and P. F. Syverson, "Tor: The second-generation onion router," in *SEC '04*.
[70] Y. Lindell, "How to simulate it: A tutorial on the simulation proof technique," in *Tutorials on the Foundations of Cryptography: Dedicated to Oded Goldreich*.
[71] T. Hoang, J. Guajardo, and A. A. Yavuz, "MACAO: A maliciously-secure and client-efficient active ORAM framework," in *Proceedings of the 26th Network and Distributed System Security Symposium*, ser. NDSS '20.

[72] E.-O. Blass, T. Mayberry, and G. Noubir, "Multi-client oblivious RAM secure against malicious servers," in *ACNS '17*.
[73] 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*.
[74] T. Hoang, M. O. Ozmen, Y. Jang, and A. A. Yavuz, "Hardware-supported ORAM in effect: Practical oblivious search and update on very large dataset," in *PETS '19*.
[75] C. Sahin, V. Zakhary, A. El Abbadi, H. Lin, and S. Tessaro, "Taostore: Overcoming asynchronicity in oblivious data storage," in *S&P '16*.
[76] T. Hoang, R. Behnia, Y. Jang, and A. A. Yavuz, "MOSE: Practical multi-user oblivious storage via secure enclaves," in *CODASPY '20*.
[77] E. Boyle, K.-M. Chung, and R. Pass, "Oblivious parallel RAM and applications," in *TCC '16-A*.
[78] B. Chen, H. Lin, and S. Tessaro, "Oblivious parallel ram: Improved efficiency and generic constructions," in *TCC '16-A*.
[79] I. Damgård, V. Pastro, N. Smart, and S. Zakarias, "Multiparty computation from somewhat homomorphic encryption," in *CRYPTO '12*.
[80] A. C.-C. Yao, "How to generate and exchange secrets," in *FOCS '86*.
[81] O. Goldreich, S. Micali, and A. Wigderson, "How to play any mental game: A completeness theorem for protocols with honest majority," in *STOC '87*.
[82] M. Ben-Or, S. Goldwasser, and A. Wigderson, "Completeness theorems for non-cryptographic fault-tolerant distributed computation," in *STOC '88*.
[83] D. Chaum, C. Crépeau, and I. Damgård, "Multiparty unconditionally secure protocols," in *STOC '88*.
[84] D. Beaver, S. Micali, and P. Rogaway, "The round complexity of secure protocols," in *STOC '90*.
[85] J. B. Nielsen and S. Ranellucci, "Reactive garbling: Foundation, instantiation, application," in *ASIACRYPT '16*.
[86] C. Liu, X. S. Wang, K. Nayak, Y. Huang, and E. Shi, "ObliVM: A programming framework for secure computation," in *S&P '15*.
[87] "ObliVMGC: The garbled circuit backend for the ObliVM framework," https://github.com/oblivm/ObliVMGC.
[88] "FlexSC: A flexible efficient secure computation backend," https://github.com/wangxiao1254/FlexSC.
[89] 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*.
[90] "ORAM library for Obliv-C," https://github.com/samee/sqrtOram.
[91] "The absentminded crypto kit," https://bitbucket.org/jackdoerner/absentminded-crypto-kit/src.
[92] "Multi-protocol SPDZ," https://github.com/data61/MP-SPDZ/.
[93] M. Keller, "MP-SPDZ: A versatile framework for multi-party computation," in *CCS '20*.
[94] "SCALE and MAMBA," https://github.com/KULeuven-COSIC/SCALE-MAMBA.
[95] A. Fiat and A. Shamir, "How to prove yourself: Practical solutions to identification and signature problems," in *CRYPTO '86*.
[96] T. P. Jakobsen, J. B. Nielsen, and C. Orlandi, "A framework for outsourcing of secure computation," in *CCSW '14*.
[97] I. Damgård, K. Damgård, K. Nielsen, P. S. Nordholt, and T. Toft, "Confidential benchmarking based on multiparty computation," in *FC '16*.
[98] "Efficient multiparty computation toolkit (EMP-toolkit)," https://github.com/emp-toolkit.
[99] S. Eskandarian, H. Corrigan-Gibbs, M. Zaharia, and D. Boneh, "Express: Lowering the cost of metadata-hiding communication with cryptographic privacy," in *SEC '21*.
[100] C. Gentry, S. Halevi, and N. P. Smart, "Homomorphic evaluation of the AES circuit," in *CRYPTO '12*.
[101] S. Halevi and V. Shoup, "Bootstrapping for HElib," in *EUROCRYPT '15*.
[102] P. Grubbs, T. Ristenpart, and V. Shmatikov, "Why your encrypted database is not secure," in *HotOS '17*.
[103] L. Ren, C. W. Fletcher, A. Kwon, E. Stefanov, E. Shi, M. van Dijk, and S. Devadas, "Constants count: Practical improvements to oblivious RAM," in *SEC '15*.
[104] H. Chen, I. Chillotti, and L. Ren, "Onion ring ORAM: Efficient constant bandwidth oblivious RAM from (leveled) TFHE," in *CCS '19*.
[105] T. Hoang, C. D. Ozkaptan, A. A. Yavuz, J. Guajardo, and T. Nguyen, "S3oram: A computation-efficient and constant client bandwidth blowup ORAM with Shamir secret sharing," in *CCS '17*.
[106] T.-H. H. Chan, J. Katz, K. Nayak, A. Polychroniadou, and E. Shi, "More is less: Perfectly secure oblivious algorithms in the multi-server setting," in *ASIACRYPT '18*.
[107] S. Lu and R. Ostrovsky, "Distributed oblivious ram for secure two-party computation," in *TCC '13*.
[108] S. D. Gordon, J. Katz, and X. Wang, "Simple and efficient two-server oram," in *ASIACRYPT '18*.
[109] E.-O. Blass, T. Mayberry, G. Noubir, and K. Onarlioglu, "Toward robust hidden volumes using write-only oblivious RAM," in *CCS '14*.
[110] D. S. Roche, A. Aviv, S. G. Choi, and T. Mayberry, "Deterministic, stash-free write-only ORAM," in *CCS '17*.
[111] A. J. Aviv, S. G. Choi, T. Mayberry, and D. S. Roche, "ObliviSync: Practical oblivious file backup and synchronization," in *NDSS '17*.
[112] T.-H. H. Chan and E. Shi, "Circuit OPRAM: Unifying statistically and computationally secure ORAMs and OPRAMs," in *TCC '17*.
[113] T.-H. H. Chan, K.-M. Chung, and E. Shi, "On the depth of oblivious parallel RAM," in *ASIACRYPT '17*.
[114] J. R. Lorch, B. Parno, J. Mickens, M. Raykova, and J. Schiffman, "Shroud: Ensuring private access to large-scale data in the data center," in *FAST '13*.
[115] P. Williams, R. Sion, and A. Tomescu, "PrivateFS: A parallel oblivious file system," in *CCS '12*.
[116] E. Stefanov and E. Shi, "Multi-cloud oblivious storage," in *CCS '13*.
[117] A. Hamlin, R. Ostrovsky, M. Weiss, and D. Wichs, "Private anonymous data access," in *EUROCRYPT '19*.
[118] S. Sasy, S. Gorbunov, and C. W. Fletcher, "ZeroTrace: Oblivious memory primitives from Intel SGX," in *NDSS '18*.
[119] S. Eskandarian and M. Zaharia, "ObliDB: Oblivious

query processing for secure databases," in *VLDB '19*.
[120] P. Mishra, R. Poddar, J. Chen, A. Chiesa, and R. A. Popa, "Oblix: An efficient oblivious search index," in *S&P '18*.
[121] A. Ahmad, K. Kim, M. I. Sarfaraz, and B. Lee, "Obliviate: A data oblivious filesystem for Intel SGX," in *NDSS '18*.
[122] C. Bao and A. Srivastava, "Exploring timing side-channel attacks on Path-ORAMs," in *HOST '17*.
[123] C. W. Fletcher, L. Ren, X. Yu, M. Van Dijk, O. Khan, and S. Devadas, "Suppressing the oblivious RAM timing channel while making information leakage and program efficiency trade-offs," in *HPCA '14*.
[124] D. Lazar and N. Zeldovich, "Alpenhorn: Bootstrapping secure communication without leaking metadata," in *OSDI '16*.
[125] D. Chaum, "Blind signatures for untraceable payments," in *CRYPTO '82*.
[126] C. Dwork and M. Naor, "Pricing via processing or combatting junk mail," in *CRYPTO '92*.
[127] M. Jakobsson and A. Juels, "Proofs of work and bread pudding protocols," in *CMS '99*.
[128] A. Juels and J. Brainard, "Client puzzles: A cryptographic defense against connection depletion attacks," in *NDSS '99*.

APPENDIX

A. Proof sketches

We now present our proof for the security of Titanium. The proof is straightforward, as we can simply invoke a simulator for secure computation that captures both the computation in boolean circuits (used to find the file and compute the eviction plan) and the authenticated secret sharing that we use.

1) Definitions: We first describe the ideal world, the real world, and the simulator. Let $\mathcal{F}_{\mathsf{FileSharing}}$ and $\mathcal{S}$ denote the ideal functionality and the simulator (i.e., the ideal-world adversary), respectively. Since users are considered anonymous, and the only thing that identifies them is the credential and the user ID. In the following discussion, we assume that adversary $\mathcal{A}$ can decide the requests that clients make through the N servers. The N servers in the ideal world become dummy and simply forward data between the client software and $\mathcal{F}_{\mathsf{FileSharing}}$.

We say that the protocol securely realizes the ideal functionality if the output of the ideal world is computationally indistinguishable from the output of the real world.

Ideal world:

- **Initialization.** $\mathcal{F}_{\mathsf{FileSharing}}$ creates a credential-user map, a file storage map, and a file permission map.
- **Create an account.** $\mathcal{S}$ can instruct the client software to send a request to $\mathcal{F}_{\mathsf{FileSharing}}$ for a new account, where the client software receives the user ID and a credential, which is then forwarded to $\mathcal{S}$
- **Create a file.** $\mathcal{S}$ can instruct the client software to send a request to $\mathcal{F}_{\mathsf{FileSharing}}$ for a new file, using a user ID and credential. The client receives the file ID, which is then forwarded to $\mathcal{S}$.
- **Access a file.** $\mathcal{S}$ can instruct the client software to send a file access request to $\mathcal{F}_{\mathsf{FileSharing}}$, with a credential, a file ID, and the operation to perform. The client either receives the file data or dummy data, which is then forwarded to $\mathcal{S}$.
- **Grant and revoke permission.** $\mathcal{S}$ can instruct the client software to send a permission change request to $\mathcal{F}_{\mathsf{FileSharing}}$, with a credential, the file ID, the other user's ID, and the change to make.
- **Output.** In the end, $\mathcal{S}$ outputs the simulated views of the servers and simulated output of the real adversary $\mathcal{A}$.

Real world:

- **Initialization.** $\mathcal{A}$ chooses a number of servers to corrupt. We let $\mathcal{A}$ choose exactly $N-1$ servers. The servers, following the protocol, initiate the secure computation of the proxy.
- **Create an account.** $\mathcal{A}$ can instruct the client software to send a request to create an account, where the client software receives the user ID and a credential and forwards them to $\mathcal{A}$.
- **Create a file.** $\mathcal{A}$ can instruct the client software to send a request to create a new file, using a credential, where the client receives the file ID and forwards it to $\mathcal{A}$.
- **Access a file.** $\mathcal{A}$ can instruct the client software to send a file access request, with a credential, a file ID, and the operation to perform. The client either receives the file data or dummy data as the response, which is then forwarded to $\mathcal{A}$.
- **Grant and revoke permission.** $\mathcal{A}$ can instruct the client software to send a permission change request, with a credential, the file ID, the other user's ID, and the change to make.
- **Output.** In the end, the servers and $\mathcal{A}$ output their views.

The simulator:

- **Initialization.** We assume $\mathcal{S}$ has a black-box access to $\mathcal{A}$. $\mathcal{S}$ asks $\mathcal{A}$ which servers to corrupt and then simulates the transcript of corrupted servers in the initialization protocol to $\mathcal{A}$, by invoking the simulator for secure computation.
- **Create an account.** If $\mathcal{A}$ wants to create an account, $\mathcal{S}$ forwards this request to the dummy servers and returns the simulated transcript of the corrupted servers to $\mathcal{A}$, by invoking the simulator for secure computation. $\mathcal{S}$ forwards the user ID and the credential to $\mathcal{A}$.
- **Create a file.** When $\mathcal{A}$ wants to create a file, $\mathcal{S}$ forwards this request to the dummy servers and returns the simulated transcript of the corrupted servers to $\mathcal{A}$, by invoking the simulator for secure computation and using random paths to simulate the Circuit ORAM traces.
- **Access a file.** When $\mathcal{A}$ wants to perform a file access operation, $\mathcal{S}$ forwards this request to the dummy servers and returns the simulated transcript of the corrupted servers to $\mathcal{A}$, by invoking the simulator for secure computation and using random paths to simulate the Circuit ORAM traces.
- **Grant and revoke permission.** When $\mathcal{A}$ wants to perform a permission change, $\mathcal{S}$ forwards this request to the dummy servers and returns the simulated transcript of the corrupted servers to $\mathcal{A}$, by invoking the simulator for secure computation and using random paths to simulate the Circuit ORAM traces, similar to the file access.
- **Output.** In the end, $\mathcal{S}$ outputs whatever $\mathcal{A}$ outputs.

2) Security with abort: In Titanium, $\mathcal{A}$ can abort at various stages. We now discuss why such aborting does not help $\mathcal{A}$ learn any secret.

- **Case 1: Aborting the secure computation.** If $\mathcal{A}$ corrupts the servers to perform invalid operations in secure computation, it will be discovered by the honest servers. The honest servers will abort the protocol, and no more operations can be done. The simulator for secure computation can still simulate the

transcript of the corrupted parties up to the aborting, without leaking secret data.

- **Case 2: Manipulating a user's input or output.** The user can verify the output from the servers via the input/output protocol. Due to the Schwartz-Zippel lemma [58–60], the user can detect if $\mathcal{A}$ manipulates the response with an overwhelming probability. The same applies to the user's input.

3) Proof of indistinguishability: We now show that the outputs of the real world and the views of the ideal world are computationally indistinguishable.

Proof. We prove the security using hybrid arguments. That is, we use a sequence of hybrids (denoted by $\mathsf{H}_\bullet$) to show that the output of the ideal world is computationally indistinguishable (denoted by $\approx$) from the output of the real world. Consider that there have been q requests, where q is a number polynomially bounded by the security parameter. We use the same proof strategy as in Metal [35]: we replace the output of each of the q requests one by one with the output from real execution. We let H_t denote the t-th hybrid, in which the outputs of t out of q requests have been replaced. We start with H_0, which is the output of the ideal world.

Without loss of generality, we assume $\mathcal{A}$ does not abort. For each $t \in \{0, ..., q\}$, we define H_t as follows: H_t has the output of the first t requests in the real world, and the output for the remaining requests is in the ideal world (which would be a list of the requests and the output of the simulators). Our goal is to show that for $t \in \{0, ..., q-1\}$, $\mathsf{H}_t \approx \mathsf{H}_{t+1}$. We now consider several cases.

- **Create an account.** Note that H_{t+1} replaces the $\mathcal{S}$'s simulated output of the servers corresponding to creating an account with the real execution. The simulated output is a direct result of the use of the simulator for secure computation. By the security of the secure computation, we have that the simulated output is computationally indistinguishable from the real execution, and therefore $\mathsf{H}_t \approx \mathsf{H}_{t+1}$.
- **Create a file.** Similarly, H_{t+1} replaces the $\mathcal{S}$'s simulated output of the servers corresponding to creating a new file (reserving space for the user's new file and giving the user ownership permission to the space) with the real execution. Besides the use of the simulator for secure computation, another difference is that the ORAM traces (for updating the access control matrix) in the ideal world are sampled from uniform random. Due to the security of Circuit ORAM, the traces in the real execution are also uniformly random and statistically independent from the file being accessed. Adding that the security of secure computation shows that the simulated output is computationally indistinguishable from the real execution, we know that $\mathsf{H}_t \approx \mathsf{H}_{t+1}$.
- **Read or write a file.** We now discuss "read" and "write" together since these two operations are designed to be indistinguishable from each other (see §VIII and Tab. VI) and have the same computation patterns. H_{t+1} replaces the simulated output with the real execution. To generate the simulated output, $\mathcal{S}$ invokes the simulator of secure computation and uses randomly sampled paths in simulating the ORAM. Due to the security of Circuit ORAM and secure computation, the simulated output is computationally indistinguishable from the real execution, and therefore $\mathsf{H}_t \approx \mathsf{H}_{t+1}$.
- **Grant or revoke permission.** In Titanium, the access control matrix described in §VII-A is stored together with the file data. So, granting and revoking is similar to writing to a file, with the difference that the modification is done on the access control matrix. Therefore, when H_{t+1} replaces the simulated output with the real execution, for the same reasons, the simulated output is computationally indistinguishable from the real execution, and therefore $\mathsf{H}_t \approx \mathsf{H}_{t+1}$.

Now, by hybrid arguments, we know $\mathsf{H}_0 \approx \mathsf{H}_q$, which means that when processing q requests, the output of the ideal world and the output of the real world are computationally indistinguishable.

B. Prevent DoS attacks from users

In this paper, we do not consider denial-of-services (DoS) attacks by the users. Nevertheless, it remains an issue in practice. A user who exhausts the servers' resources by uploading or downloading a large amount of data would prevent other users from using the system. There have been many existing defenses in systems *without anonymity*. However, in Titanium, anonymity must be preserved.

Titanium can leverage the anonymous prevention mechanisms discussed in prior work, Ghostor [41] and Alpenhorn [124]: (1) anonymous payment and (2) Proof-of-Work (PoW).

– *Anonymous payment.* Titanium can require a user to pay for each data access. Specifically, each user deposits some money by doing a sender-anonymous payment to the servers, and the user then proves this payment in zero knowledge to the servers. The servers then issue a number of blind signatures [125] according to the paid amount, where one blind signature serves as a "one-time token" for one data access. The user needs to present an unused blind signature to the servers for each file access. Since the servers are "blinded", the user anonymity is preserved.
– *Proof-of-work.* Titanium can deter a malicious user by asking this user to solve a cryptographic puzzle [126–128]—also commonly known as proof of work—for each data access. This does not fully prevent DoS attacks and could be expensive for resource-constrained users, but it can limit the ability of the malicious attackers, and can be combined with other mechanisms.