Topology Independence and Information Leakage in Replicated Storage Systems

Vasileios Lekakis
lex@cs.umd.edu

Abstract

A user may collaborate and share data with many other users, and across many devices. Replicated storage systems have traditionally treated all sharing equally and assumed that security can be layered on top. On the contrary, we show that security needs to be integrated into the consistency and replication mechanisms to prevent information leakage.

We describe the design and evaluation of T.Rex, a system that uses role-based access control and novel synchronization and consistency protocols to control sharing and information access across a variety of users, devices, and roles. We show that the overhead of such integration is low by characterizing the CPU, network, and storage overheads caused by supporting our security goals.

1 Introduction

The proliferation of mobile devices has resulted in personalized computing "ecosystems," wherein users employ whichever device suits their needs at the moment: a mobile phone while on the move, a desktop computer while at home, a tablet while in front of the TV, and so on. Users have come to expect that their data be available and *consistent* across all of their devices. Moreover, as collaborative software like Google Docs has become popular, users have grown to expect the data they *share* to be available and consistent, even as their collaborators perform modifications.

Replicated data systems are the canonical approach to achieving consistency across multiple devices. Generally speaking, they function as follows: When a user modifies a document, the system updates a local *version vector*, and records the change. This increases the "entropy" in the system until other devices know about this modification. When two devices (be they owned by the same user or not) encounter one another, they perform an *anti-entropy* session: they compare their version vectors, inform one another of whatever modifications they are missing, and update their version vectors. Entropy decreases with each anti-entropy session, and as updates spread from user to user, the entropy eventually approaches zero. This is the basic underlying mechanism behind both user-to-user replication systems like Bayou [48] and more recent cloud-based services like Dropbox [13]. We use *topology independence* to describe both cases. The salient point is that data does not go directly from the source to its eventual destination. Instead, it may be staged elsewhere in the network, increasing risk of exposure.

The traditional assumption in such systems is that all interacting users trust one another. In the case of TI replication systems, this can severely limit performance: if users only perform anti-entropy sessions with those they trust, then updates can only propagate as quickly as trusted users encounter one another. It would be far more efficient to use untrusted users to "ferry" updates between trusted ones. In the case of cloud-based replication systems, users are required to trust the third-party cloud provider.

In this paper, we present T.Rex, a replicated data system that achieves various forms of consistency *without requiring all interacting users to trust one another*. At first glance, it may appear that simply encrypting the data is enough. While this prevents any information leakage from the data itself, it does not prevent leakage from the *metadata* that users share during anti-entropy sessions. As we discuss, this metadata can reveal user trust relationships, behaviors, and update frequency.

T.Rex uses role-based access control [16] to enable flexible and secure sharing among users with widely varying collaboration types: both users and data items are assigned *roles*, and a user can access data only if they share at least one role. Building on top of this abstraction, T.Rex includes several novel mechanisms: We introduce *role proofs* to prove role membership to others in the role without leaking information to those not

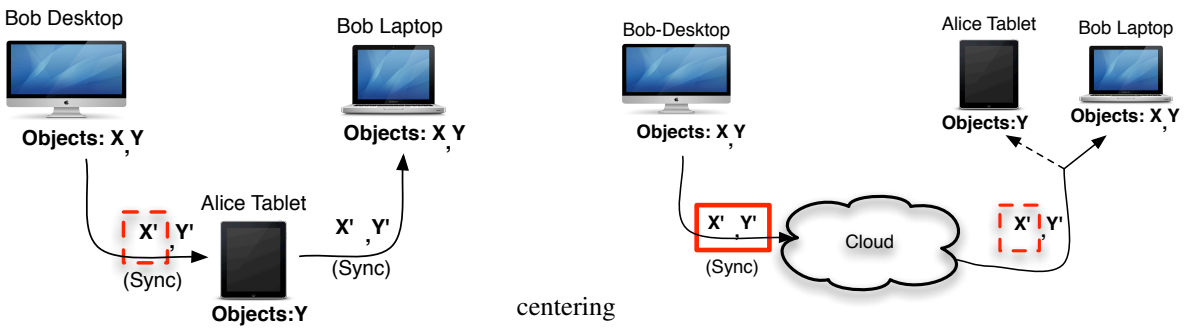


Figure 1: (a) A personal sharing system might schedule the desktop's updates to be delivered to the laptop via the tablet. Both the desktop and the laptop access X and Y, while the tablet only accesses (*has rights for*) object Y. The data shown inside a box is potentially *leaked*, in that the tablet not only can see that the desktop just modified X, but might also be able to see the contents of the update. (b) Cloud services add cloud replicas to the set of user replicas that might stage data for which they have no access rights. Data, and information about data, might leak to any of the user or cloud replicas.

in the role. We introduce *role coherence* to prevent updates from leaking across roles. Finally, we use Bloom filters as *opaque digests* to enable querying of remote cache state without being able to enumerate it. We believe these to be generally applicable beyond replicated data systems.

We combine these mechanisms to develop a novel, cryptographically secure, and efficient anti-entropy protocol. To demonstrate its general-purpose use, we have built several different consistency schemes on top of it, including eventual consistency [48], PRAM [28], causal consistency [6], and causal+ consistency [29]. We have implemented T.Rex and these consistency schemes, and with an evaluation on a local testbed, we demonstrate that it achieves security with modest computational and storage overheads.

The rest of this paper is organized as follows. Section 2 describes our model and security goals. Section 3 describes T.Rex's overall design. Section 4 describes the implementation, and how this implementation meets security challenges. Section 5 covers the experimental evaluation of T.Rex, Section 6 describes application of our approach to cloud services, and we present related work and conclude in Sections 7 and 8.

2 Motivation and model

Topology independent protocols [48, 9] allow replicas to push updates to any other replicas in the system, effectively using them as data relays. This approach is flexible, and can handle varying connectivity while maximizing the use of scarce resources. However, data relays at least temporarily host updates for objects in which they are not interested, and potentially have no rights to access. We show that such data placement exposes conventional replication and consistency protocols to data leakage.

Figure 1(a) demonstrates an example of this process in an anti-entropy [48] protocol. We assume a replication protocol that supports TI with three participating replicas: a desktop, a laptop and a tablet. Both the desktop and the laptop store objects X and Y. The third device, a tablet, stores only object Y and is owned by a second user: Bob.

The system starts with a consistent set of states: both X and Y have the same values everywhere. The desktop updates X and Y, creating new versions X' and Y'.

The figure shows the desktop sending new updates X' and Y' to the laptop, using Alice's tablet as a data relay. The issue is whether the tablet learns anything untoward as it passes X' on to the laptop. If X' is not encrypted, Alice learns about its existence and contents. If it is encrypted, Alice could learn the metadata of the file. Finally, if data and metadata are both encrypted naively, Alice's tablet would have no way to determine where to send the update.

Cloud based replication protocols also take advantage of TI in moving updates among cloud replicas, and between cloud replicas and clients (T.Rex replicas, for example). Figure 1(b) shows a similar example in which the Desktop again creates new updates X' and Y'. A cloud software agent running on the desktop, for example Dropbox [13], propagates the updates to a cloud server, which then updates Alice's other device, the laptop. The cloud is acting as a data relay, resulting in the updates potentially being pushed to *all* of Alice's client replicas, and to an arbitrary number of servers of the

cloud provider.

Whereas the anti-entropy protocol in Figure 1a potentially leaks information to the tablet, the cloud scenario potentially leaks data to replicated cloud servers. Additionally, complex relationships among devices might not be reflected in the cloud service's data schedule, opening further possibilities for data leaks. For example, the tablet might also be owned by Alice, but used with different roles. The tablet could again receive unneeded updates Unless these differences are exposed to the cloud interface.

2.1 System model

T.Rex is a storage management system designed to manage user and application data on personal or cloud servers and mobile devices. Every participating device or server acts as a replica that eagerly replicates all or part of the total data collection.

T.Rex sharing is defined and constrained through interaction between possibly overlapping *roles*. A role consists of a unique name, a secret key, and the *role predicate*. The role predicate is defined over per-file meta-information called *labels*. A predicate for the collection of tax documents might be `filetype=pdf && context=taxes`, for example. A role defines data collections, via the predicates, and access groups, via the group of devices that participate in the role.

Replicas periodically push data (created either locally or received from other replicas) to other replicas through one-way *anti-entropy sessions* [48]. The choice of destination could be random, or determined by availability, stability, or available bandwidth. Data is eventually relayed from where it is created to all interested replicas through a succession of such anti-entropy sessions.

The system as a whole contains many devices and users. We distinguish between *T.Rex replicas*, those running the protocol, and *cloud replicas*, which replicate cloud data under the control of cloud service providers. We assume that devices each play only a small number of roles ($<$10), and are single-user, with the usual convention of a multi-user machine being treated as multiple virtual devices. Each device hosts a single replica, so we use the two terms interchangeably, depending on context.

2.2 Security goals

We assume Byzantine failures [26]. Replicas may maliciously and arbitrarily deviate from correct protocol execution in an attempt to subvert the protocol. By contrast, software running in cloud services is often assumed to be "honest-but-curious" [20]. Such replicas would follow the protocol honestly but might analyze the protocol messages to infer information about users and their data. However, given the multitude of reports of cloud services inadvertently exposing data [31, 41, 23, 12, 33, 22, 46], together with recent disclosures of national intelligence agencies compelling data disclosure, there seems little reason to differentiate the cloud failure model from that of the standard T.Rex Byzantine model.

Our security goals are to provide data confidentiality, data integrity, and to prevent information (including metadata) leakage. More specifically, we attempt to prevent the following types of information flow:

- *replica or user roles* - Other than roles common to both, replica r_i should not be able to learn which roles r_j plays.
- *data written in object updates* - Any data written in the context of role r_i should not be visible in plaintext to a replica that does not play role r_i.
- *role activity* - For roles that it does not play, a replica should not be able to identify the roles of encrypted data.

3 T.Rex design

This section provides a high-level overview of the T.Rex system. A T.Rex "group" consists of a set of devices, possibly owned by many users, each with a T.Rex replica. Replicas interact in the context of a set of roles defined by a single, distinguished *role master*, though this responsibility can be delegated. The role master defines roles and their keys, maps rights and capabilities onto roles, and maps roles onto users or devices. The role master also generates and disseminates certificates for the public keys of devices and users. Interaction between groups (different role masters) is possible,but they share in the context of a single group.

The role master is also responsible for key revocation. We follow an approach similar to SPORC [15]. A replica is removed from an existing role by creating a new *update-role-key* message signed by the role master. This message contains a new role key and a timestamp, encrypted with the public keys of the replicas that remain in the role. Replicas removed from the role can still see old updates but not those created after the update-role-key message.

Replica state Each replica has a global unique ID named *RID*. Each replica also has a *clock*, which is a counter that is incremented each time a local object update occurs. Every object in the replica has a version consisting of the triple `<OID,RID,clock>` where `OID` is a unique ID for the object. Each object also has a set of key-value attributes defining the object's labels. Replicas also have version vectors, which describe the latest updates seen from the rest of the systems replicas.

A data item consists of two parts: metadata (label, size, permissions) and the actual data of the item. Each object stored by a replica must have labels that satisfy at least one of the replica's role predicates.

Replica updates The time between two anti-entropy sessions is called the *update period*, During the update period, a replica merely records object updates created locally. Updates include object creations, deletions, and modifications, and object label definitions and deletions.

At the end of an update period, a replica gathers the actions created in the previous period and creates two types of per-role entities: *kernels*, which contain meta-information describing the actions of the previous period, and *shards*, which include the same meta-information as well as any actual update data. The kernel contains a unique GUID, a role ID, connection information about the replica, and the version vectors of the replica at the beginning and the end of the current period.

Update periods are always non-overlapping, as consistency protocols are otherwise quite complex and can duplicate information flow (Section 3.1). We enforce this invariant by ending the current periods when data is requested by another replica or when metadata arrives from another replica in an anti-entropy session.

Update policies A replica's *update policy* determines whether it will send kernels or shards when communicating with other replicas. These choices are analogous to choosing an invalidate or update policy in a shared memory multiprocessor. Currently, T.Rex supports three policies. Under the *kernel* policy replicas exchange only kernels (metadata). A replica locally invalidates copies of objects specified by incoming kernels, requiring the corresponding data to be demand-fetched before they can be used again. The *shard* policy sends both where data and metadata together. Finally, *kernel-shard-based* during which the source replica of the update sends shards, data and metadata for the roles that shares with the target replica and only kernels (metadata) for the rest.

3.1 Ramifications of consistency in T.Rex

T.Rex implements several different consistency protocols, including eventual consistency [48], PRAM [28], causal consistency [6], and causal+ consistency [29]. The choice of consistency protocol is not strictly relevant to this work. More relevant is whether these consistency protocols are supported at the object or role level.

Users might creating overlapping roles. This choice is necessary, as roles are defined by high-level user-defined predicates. Ensuring that logical predicates are non-overlapping is difficult, and may not always be possible.

Consider a replica, r_i, that plays two roles: $role_x$ and $role_y$, and suppose that the intersection of $role_x$ and $role_y$ is object o_a. If r_i receives an update for object o_a in the context of $role_x$, a straightforward implementation would immediately apply it to the local copy of o_a, assuming consistency requirements are met. However, if r_i now reads o_a in the context of $role_y$, it sees the altered value. Worse, its copy of o_a is now different from copies on replicas that only play $role_y$, and *the distinct versions will never converge*. This violates the base consistency model supported by nearly all mobile storage systems, eventual consistency, which requires that all copies of the same object *eventually* converge.

One way to support eventual consistency is to transfer the update from one role to another. However, this transfer raises several questions. First, might two replicas playing both $role_x$ and $role_y$ each transfer the same update? How are the two resulting updates to be recognized as the same?

Second, a model like causal consistency demands that if update u_1 is read at a replica before u_2 is created, u_1 *happens-before* u_2, and must be applied before u_2 anywhere u_2 is applied. If updates are transferred from one role to another, the transferred updates must obey the same consistency constraints. However, this effectively implies that two updates of $role_x$ might be ordered only through updates that were created in an entirely different role. The correctness criteria in such situations is not clear. Further, additional ordering constraints can also affect performance.

A more damning critique is that moving updates among roles can be seen as a leak of information from one role to another. Users should have the option of allowing updates to leak across roles, but the system should be able to enforce the stronger semantics.

For all of these reasons, we take a second approach: splitting any object that is updated in the context of more than one role into two physical representations, effectively forking the object's version history.

4 T.Rex implementation

This section describes protocol and mechanism detail of our implementation. T.Rex's protocols differ from that of "personal" sharing systems [42, 37, 17, 35, 39] in the use of cryptographic primitives, extra handshakes for cryptographic challenges, and in guiding consistency transfers through *opaque digests*.

4.1 Replication and information leakage

A conventional replication protocol handshake consists of the target replica t initiating a session with a request for the target's version vector. The source replica com-

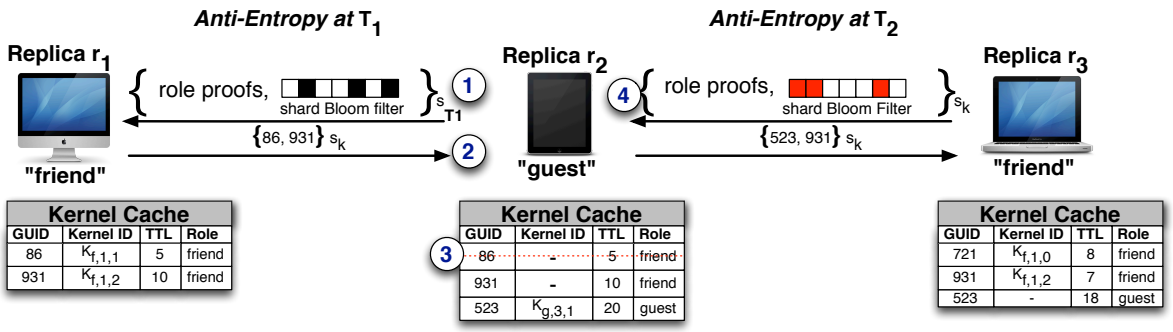


Figure 3: **Anti-entropy:** Replicas r_1 and r_3 communicating through r_2, which is of another role. $K_{r,x,y}$ refers to the kernel in role r for period y of replica x. For r_1 to push data to r_2, it first sends a request to r_2 (not shown). (1) r_2 responds with proofs describing roles played locally, and a Bloom filter describing locally cached kernels. (2) r_1 responds with new encrypted kernels not represented in the Bloom filter. Only the GUIDs (86, 931) of $K_{f,1,1}$ and $K_{f,1,2}$ are visible to r_2, the rest is encrypted. (3) After this exchange, r_2 locally makes a decision to drop kernel 86 because the TTL expires, and (4) r_2 initiates anti-entropy with r_3. The replication update then repeats between r_2 and r_3. r_3 is able to decrypt kernel $K_{f,1,2}$ but not apply it, as it is missing $K_{f,1,1}$ (GUID 86) (see Section 3.1).

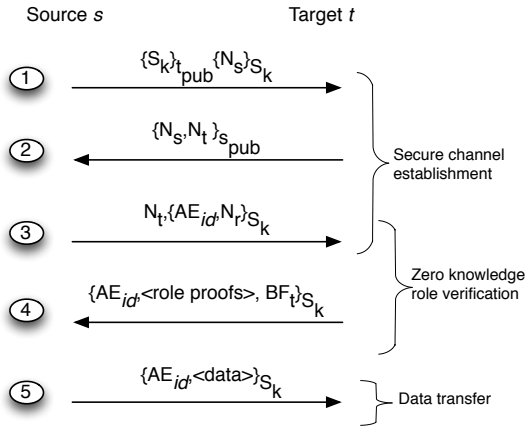


Figure 2: T.Rex-Sync

pares its version vector with the target's vector to determine the set of updates that should be sent to the target.

T.Rex's handshake is designed to make this determination without allowing either the source or target learn information data held by the other, except for roles in common. The first three messages of the handshake shown in Figure 2 are used by the source and target to authenticate each other. However, the third message also contains the role challenge and session ID (AE_{id}), and the fourth allows the target to respond with role proofs. The challenge is just a random nonce. A proof is an HMAC of the nonce and the *role key* (rk_i is the role key for role i): a per-role key shared by all devices that play that role. Aside from HMAC proofs, the key is also used to encrypt data and updates for that role[1]. A device that plays multiple roles will return a proof for each role. The source compares the target's proofs against locally created proofs to identify roles in common.

The challenge and proofs allow the target to prove it plays specific roles, but only to other replicas that also play those roles. A source replica does not learn of any target role that it does not also share. This is a fundamental difference between T.Rex and existing anti-entropy protocols, where targets completely summarize their state for arbitrary sources in the very first message.

The fourth message also contains the target's Bloom filter. We term these Bloom filters *opaque digests* to point out that the Bloom filters have security value. The Bloom filter is constructed with a cryptographic hash, and values in the filter are kernel GUIDs chosen uniformly at random from a large (128-bit) space. The source gains no information about any kernels present at the target unless it has those kernels as well. This Bloom filter summarizes IDs of kernels known to the target, much like a summary cache [14]. The source queries the Bloom filter for kernel IDs cached locally, sending to the target any not contained in the filter.

Bloom filters are used to obfuscate data cached by the target. Kernels are encrypted to prevent information leaking to other roles. In more detail:

> *An encrypted kernel reveals neither the role in which it was created, nor the local clock value when it was created, nor the replica at which it was created.*

This *update anonymity* means that a target only learns

[1]Our implementation currently uses an AES key as a role key, but in the future we may use a public key, with hybrid encryption for confidentiality.

explicit information about kernels, or updates, of roles shared with the source. Replica i does not know which kernels it has seen from replica j, as kernels created by j for roles not shared by i will be encrypted.

A target can only describe its state by enumerating all kernels seen locally, or by using some sort of summary data structure. An enumeration is not bandwidth-efficient, but more importantly would reveal explicit information about the target. A listing of kernels cached at the target reveals its *target policy* (how it chooses partners for anti-entropy), and *caching policy* (which kernels to cache for forwarding to other replicas in subsequent anti-entropy sessions). A replica's local caching policy will often prefer kernels of roles it shares. Therefore, a snapshot of cached kernels at replica i, when compared with a snapshot of cached kernels at replica j, could reveal commonalities. These commonalities, in turn, could show that either the target or cache policies of the two replicas are similar, giving good evidence that the replicas share at least one role.

The final step of the protocol is data transfer. The source identifies roles in common with the target by comparing role proofs. Call this set of common roles R_{com}. Kernel $k_{r,i,j}$ is the j^{th} kernel created by replica i for role r. All kernels $k_{r,i,j}$, such that $k_{r,i,j} \notin BF_t$ and $r \in R_{com}$, can be sent to the target without extra encryption by the role key rk_i, since both source and target share role $role_i$. The data will still be encrypted with the session key before it is sent out over the network. Kernels for roles not common to both the source and target are doubly encrypted: first with the role key, and again with the session key.

Kernels not common to both the source and target still might be in a role played by the target. The target therefore must try to decrypt the header of each unknown kernel with each of it's keys. This is not a large overhead given our assumption that devices play only a small number of roles.

Figure 3 shows the outline of two complete anti-entropy sessions in T.Rex. Replica r_1 initiates a session with r_2, with which it shares no roles. Two kernels are sent to r_2, and at some future point in time r_2 makes a local decision to drop kernel 86 from it's cache. Hence, kernel 86 never makes it to r_3. Replica r_3 will discover this omission when it later learns of subsequent updates in the same role, or by the same device.

4.2 Role Coherence and object forking

We term the object-forking approach *role coherence*. Objects are initially created tagged with all roles in which they are visible. An object updated in the context of one role is forked into a version history for that role, and a version history for all other roles that can see the object. At the limit, an object visible in n roles is forked into n logically distinct objects, each of which may continue to be updated. Inside a role, all of the consistency schemes work as before. This approach is similar to that of Qufiles [51], in which a single file can have multiple physical instantiations, transcoded for different bandwidth requirements.

Role coherence requires accesses to be associated with a single role. Updates from remote replicas are carried in role-specific containers (shards), and therefore have explicit role associations. An early version of T.Rex handled local reads and writes by passing the role context through the POSIX file system interface as a modifier on the base filename, as in "`filename@role`". We currently handle local reads and writes by defining an interface that allows applications to specify a current role, and use this role to implicitly tag object reads and writes. Labeling could also be done automatically [44] or by using the hints of a provenance tracking subsystem [32].

Object forking creates overhead in both metadata and object data. A straightforward implementation of the underlying object-forking storage mechanism would fork a new copy of an object with each incoming update, causing storage overheads to increase rapidly with the number of roles.

Instead, we represent objects as *recipes*, which are ordered lists of block hashes [49, 19]. An object's data consists of the concatenation of the blocks from the recipe, in order. Instead of replicating the whole object to create a new version, we only update the recipe (metadata) of the object to include the new blocks, and save only modified blocks to disk.

4.3 Attacks

In this section we describe how T.Rex defends against a variety of attacks.

Passive eavesdropping: Anti-entropy sessions use public keys to establish secure channels. The secure channel is established before replicas reveal any information about replica identity or state.

Impersonation: Session establishment is secure against replay attacks because of fresh nonce challenges. An attacker could replay the first message from an earlier exchange. However, replaying the third message would fail because the target's challenge nonce in message two will change.

Role Leakage: In this family of attacks a malicious replica tries to find role commonality among two other replicas. Trudy could initiate anti-entropy sessions with both Alice and Bob, using the same role challenge, N_r, with both. Both would return vectors of role proofs (message 4 in Figure 2), one for each role they play. Trudy

could determine how many roles the two play in common by comparing the proof vectors.

T.Rex prevents this attack by tying the challenge to the current session. N_r is defined as being equal to $N_s \oplus N_t$.

Activity Inference: We define activity inference as a replica learning about updates in roles that it does not share. Assume Trudy and Alice participate in an anti-entropy session, but do not share any roles. The protocol should allow Trudy to update Alice without Alice revealing anything about her state, or what she knows about other replicas. This is a *fundamental* difference between T.Rex and other anti-entropy protocols, which require the target of an update to summarize its state to the source. T.Rex lets Alice summarize state through an opaque digest, revealing nothing about updates unknown to Trudy.

Drop out-of-role updates: Trudy could discard all updates that do not belong to her roles. This will not affect either liveness or correctness, but it will impact performance.

4.4 Freshness

We use Bloom filters to compare replicas' cache contents without revealing unnecessary information. Bloom filters may be efficiently queried without allowing enumerations, but they can grow large if they contain many objects, or are required to have low false positive rates. Our Bloom filters are currently structured to have false positive rates of less than or equal to 0.5%.

Constantly adding newly-seen kernels to a Bloom filter with a constant false positive rate would result in ever-increasing filter sizes. We bound this growth by defining a *freshness* (f) interval. The target is constrained to include in his Bloom filters all cached kernels not older than the freshness interval, and kernels older than the freshness interval are dropped from the cache once they have been applied locally.

Freshness allows us to bound Bloom filter growth, but admits the possibility that kernels may not survive long enough to be propagated everywhere. T.Rex's consistency module eventually detects these events and requests the shards directly from the creator, which is guaranteed to hold onto locally-created shards for some tunable, but long, period of time.

For example, a replica with f=10 is interested in updates that have been created (or received, since replicas can act relays) from the source during its last 10 anti-entropy periods. If a target is tuned to initiate one anti-entropy session each minute, then f=10 means that the replica is interested for updates that are at most 10 minutes old. In other words, freshness is a metric of time, defined as the number of anti-entropy sessions since an update was created, or received. Systems in highly-connected environments might use a low value of freshness, since high connectivity allows them to quickly learn of newly created updates. On the other hand, devices and systems with low connectivity might use higher values of f. By default, freshness is a per-device attribute, but T.Rex also supports per-role f values.

4.5 Prototype

T.Rex's primary interface is the POSIX file system interface, using FUSE to bind our user-level servers to the Linux kernel's VFS interface. The T.Rex prototype consists of approximately 24,000 lines of C code compiled with gcc-4.6.3. The prototype uses Google protocol buffers 2.5 [50] to serialize messages exchanged between replicas, and the ZMQ-2.2.0 [4] networking library to communicate. We use libTomcrypt-1.17 [2] to implement all cryptographic operations, with 160-bit SHA-1 hashes, 256-bit AES for payload encryption, and 2048-bit RSA keys. Metadata is stored in sqlite3. We also use several data structures provided by the UTHash library [3].

T.Rex is divided into a set of high-level, communicating modules. The *net* module handles all network communications, and implements *T.Rex-Sync* logic. The *consistency* module checks consistency-related prerequisites of incoming kernels and applies or blocks them. The *sk-factory* module produces and manages the storage of shards and kernels, both locally and remotely created. Finally, the *FS* module provides a file system interface for the system by plugging into the kernel's virtual file system (VFS) layer.

In more detail, an incoming kernel is passed by the *net* module to *sk-factory*. If the kernel matches a local role, *sk-factory* will decrypt it, verify it, and pass it to the *consistency* module. Kernels that do not match any local role are sent to disk. The *consistency* module checks whether the kernel can be applied without violating consistency invariants. If so, each non-stale (overwritten by a logically later update) update contained in the kernel is applied to the local object store. If not, it is stored on a *pending* queue until later kernels unblock it, or supersede it. Updates are generally applied by replaying appropriate file system operations from the *FS* module. Finally, the state of the replica is updated in the database.

Outgoing kernels are created by *sk-factory* at the start of an anti-entropy session (Section 4.1). The *net* module, implementing T.Rex-Sync, determines whether each kernel should be pushed to the target.

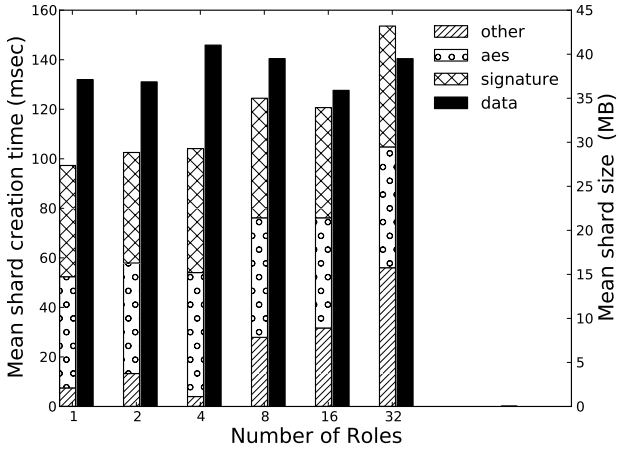


Figure 4: Shard creation CPU costs, and shard sizes.

5 Performance evaluation

Our goal in building T.Rex is to explore the potential for eliminating information leaks in replicated storage systems. Sections 3 and 4 discuss new functionality in our replication and consistency protocols; this section quantifies the cost of supporting that functionality.

We evaluate three categories of overhead. First, CPU overheads arise from the extra cryptographic operations used for authentication and confidentiality. Second, our protocols include extra authentication messages, and potentially send duplicated data because of update anonymity. Finally, the new functionality adds space overhead both because of new data structures, and because of duplicated data across roles.

We drive our evaluation through a dataset modeled on a large collection of images from the online picture-sharing site *500px* [1]. There are 18,315 files, with mean size of 358.1 KBytes and median size of 335.1 KBytes.

5.1 CPU costs

The goal of our first experiment is to compare T.Rex-Sync with existing synchronization protocols. Existing protocols roughly follow the same approach. The replicas exchange version vectors to inform each other of updates they have seen. Next, the source uses the target's version vector to determine which locally seen updates should be sent to the target. Our first experiment compares T.Rex-Sync with a stripped down version of T.Rex-Sync called `trad-sync`, which models the conventional anti-entropy approach.

We use two replicas, residing on different machines and communicating over a 100-Mbit local area network (LAN). During each 60-second period, the source selects 10,000 files from the initial dataset and performs random data and metadata updates. The source then initiates an anti-entropy with the target. The total duration of a single run of this experiment is five hours. We vary the number of roles, R, that the two replicas play, where $R \in [1, 32]$. For each R we performed three runs, each using a different distribution of updates in update periods. The distributions we used are *uniform*, *zipf* with $s = 2$, and *Poisson* with $\lambda = 50$. We present results only for the *uniform* case, as the other results were qualitatively similar. Both machines run Linux Ubuntu 12.04 with an Intel i5-7502.67 GHz for the source and an Intel Core 2 Duo-E84003.00GHz for the target.

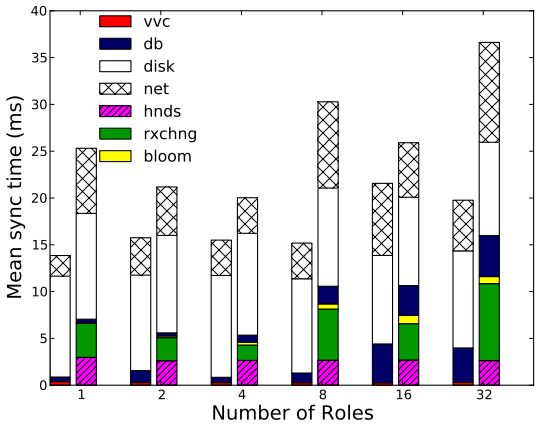
5.1.1 Shard creation

Figure 4 shows the average data exchanged over the course of the 300 anti-entropy sessions, with varying number of roles. The average shard is 40MB in size, and requires approximately 100ms to be created for runs where the replicas participate in 4 or fewer roles, or 120ms or more when R is higher. The extra cost is caused by larger R values increasing the percentage of updates that must be role-encrypted. A creation time of 100ms to 140ms is large, but is amortized across 10,000 distinct updates, 4k bytes each.

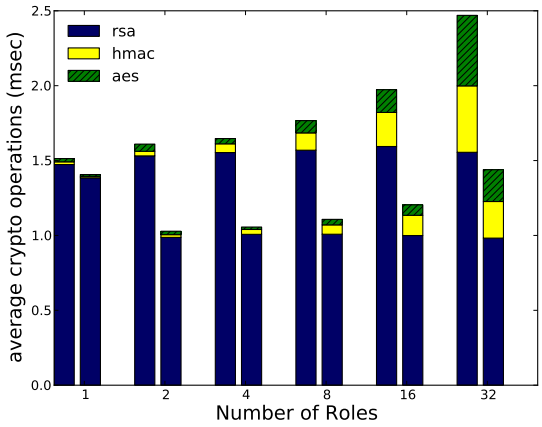
As shown by the left bars, roughly comparable portions of shard creation time are spent on public-key encryption (the initial handshake, plus signature generation and verification for shards), and AES encryption (all outgoing and incoming data). The remaining time, marked 'other' on the figure, includes disk accesses to store shard data and metadata, serialization of messages to and from protocol buffers, and miscellaneous copying. This latter category increases linearly as a function of R. The public-key overhead is constant.

5.1.2 Anti-entropy

Figure 5a shows the average time needed to complete a single anti-entropy session for *trad-sync* (the bar on the left) and *T.Rex-Sync* (the bar on right). *Version vector check* ("vvc") is the time spent by the source in checking for shards needed by the target, as indicated by the target's version vector. This cost is associated only with *trad-sync* since T.Rex uses Bloom filters to determine this information. *Database* ("db") is time spent by the source in loading all the metadata needed to construct shards. *Disk* is the overhead of loading the shard's actual data from the disk. *Network* ("net") is the aggregate time-on-the-wire network costs throughout an anti-entropy session, *minus* the network latency of sending the shard data. The actual sending of data can happen



(a) Traditional synchronization protocol (`trad-sync`) on left, T.Rex-Sync on right.



(b) Costs of cryptographic operations. Bars on the left are for the source, and on the right for the target.

Figure 5: Cryptographic overhead.

asynchronously (if using kernels rather than shards), and is therefore not strictly relevant here. Note, however, that the cost of sending 40MB of updates dwarfs all of these costs, even that of the public-key crypto. *Handshake* ("hnds") is the time to create the first two messages of the handshake in Figure 2. This cost is dominated by the creation and verification of the public-key challenges, and only exists in T.Rex-Sync. *Role exchange* ("rxchng") is the duration of the role-exchange procedure, as described in Section 4.1. As in the case of the handshake, the value of *rxchang* represents the time spent on local computations at the replicas, rather than time on the wire. Tasks include role proof creation and verification, database operations to retrieve role keys, and message preparation costs. The cost of the extra RPCs is shown in *net*. *Bloom-Check* ("bloom") is the time needed to query the target's Bloom filter.

As Figure 5a indicates, the largest overheads incurred by either T.Rex-Sync or *trad-sync* are from disk accesses. Our current implementation saves all file data, including new updates, on disk. This performance could be improved by writing to the disk asynchronously, and possibly through use of a cache in DRAM. However, as much of the data is merely being relayed among multiple replicas, a cache might have little locality to exploit.

Handshake cost is insensitive to the number of roles, as the number of public-key cryptographic operations is constant. On the other hand, the cost of checking role proofs does increase with roles, as the number of available roles determines the number of HMAC operations needed by the source and target. Network costs (*net*) are higher for T.Rex, as T.Rex-sync requires more RPCs during an anti-entropy session. The cost of using Bloom filters (*bloom*) is comparable to that of using version vectors (*VVC*). However, T.Rex's "freshness" constraint allows the costs associated with Bloom filters to be bounded (Section 5.2.1).

Figure 5b shows the data another way, breaking down costs of cryptographic operations with varying numbers of roles. Public-key operations are the most expensive, but as discussed above, do not vary with the number of roles. The numbers of HMAC and AES operations do vary. HMAC operations are needed for each role proof during the role exchange step, and all data is encrypted, but data for roles not known to either of the source and target is doubly encrypted. As the number of roles increases, the odds that a given role is not played by both replicas increases, implying that more data is doubly encrypted.

The new overheads are not negligible, but only become significant with many roles. However, recent user studies [24] have shown that even a static allocation of four roles serves many users well. While four roles might not suffice for the more expansive vision of sharing assumed by this work, the number of roles that can usefully be used in a group is likely to remain relatively low. Additionally, Figure 5 does not consider the costs of sending data, which dwarfs the overhead we consider.

5.2 Costs of update anonymity

T.Rex has update anonymity in that kernels for roles not played locally are opaque; the local replica knows literally nothing about them other than the randomly-created kernel GUID. While a conventional protocol can summarize known kernels through version vectors, T.Rex uses

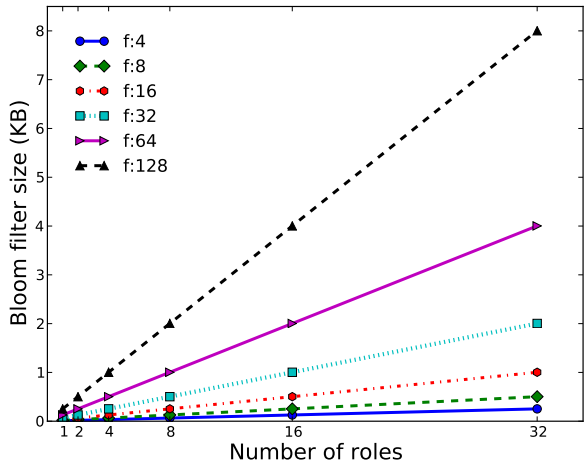


Figure 6: Bloom filter overhead with varying freshness.

Number of Replicas	Version Vector (Bytes)
5	118
15	360
35	836
55	1295

Table 1: Representative version vector sizes (after serialization)

Bloom filters and local caches to store and forward kernels obliviously. This section investigates the overhead of this oblivious approach. We measure the direct overhead of using and sending Bloom filters instead of the more common version vectors.

5.2.1 Bloom filters *versus* version vectors

Figure 6 shows the size of the Bloom filters needed by T.Rex to support a false positive error rate of less than 0.5%. The number of elements in a Bloom filter is the number of roles multiplied by the degree of freshness, e.g., the largest Bloom filter shown in Figure 6 contains $128x32 = 2^{12}$ elements. The figure demonstrates a worst-case cost, as kernels are only generated for those roles with activity during an update period. Hence, actual Bloom filter sizes are usually smaller than the numbers shown.

Version vectors grow linearly with the number of replicas that participate in the system. Our prototype implements vectors a set of tuples, each containing two integers. However, their final size is determined based on serialization algorithm of the protocol buffer implementation. Table 1 shows the size of the version vectors with different group sizes. These numbers were collected *after* the version vectors were serialized into protocol buffers. Protocol buffers shrink the space consumed by integers by representing them in as few bits as possible.

Roles	Baseline(MB)	Role Based (MB)	Cost (%)
1	30.08	38.93	29.4%
2	30.44	41.06	34.62%
4	30.96	45.88	45.54%
8	31.08	55.27	77.93%
16	34.26	73.52	114.59%
32	38.62	110.05	184.95%

Table 2: Absolute values of ***metadata*** storage overhead between the baseline case and the storage mechanism supporting the role based consistency. The intersection level is at 50% over the total number of files. Role-based overhead scales linearly with the number of roles.

5.3 Storage costs

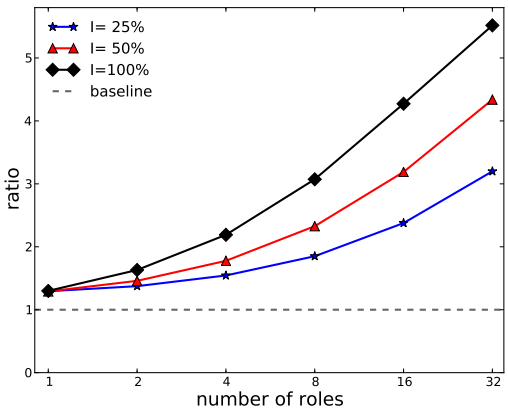
A final type of overhead is increased storage costs. Most of our protocol adds only small constant overheads, but role coherence duplicates objects across roles, and therefore potentially increases storage costs by a factor of R. However, this only occurs if objects are shared and modified in all roles, an unlikely worst-case scenario.

To summarize, updates to objects shared by multiple roles are forked (split into two distinct versions), rather than allowing updates from one role to be seen in another role. Assume $role_x$ and $role_y$ have intersection $I_{x,y}$. An update $U_i \in I_{x,y}$, created in role $role_x$, will not be seen in role $role_y$.

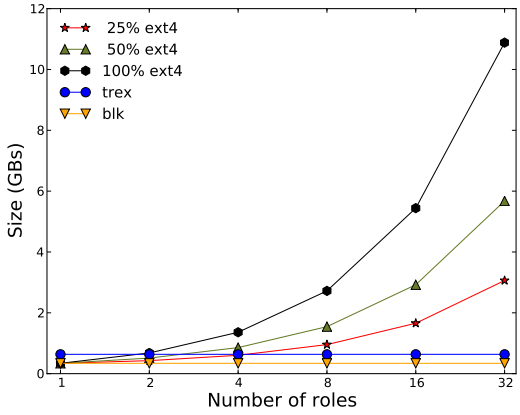
We quantify the overhead of role coherence by directly measuring the increase in storage costs as the number of roles increases. Our experimental setup consists of 1,000 randomly selected objects from our dataset, with differing intersection sizes, I. We varied I from 25% of the original group, to 50%, and finally to 100%.

Files that fall into the intersection are assigned every available role r in the system with $r \in [1-32]$. The remaining files are each placed into a single, randomly selected role. Each run consists of 10,000 4KB file updates. Files to be updated are selected using a zipf distribution that biases towards smaller files. This is intended to model "hot" files, small files that are used and updated frequently. We also ran experiments selecting files uniformly at random, but the results were similar. We selected 4KB as the update size by observing block changes as image filters are applied to image files.

Table 2 shows the absolute overhead for storing the role-based versions when the intersection $I = 50\%$. Figure 7a shows the relative overhead of the role-based ap-



(a) Metadata overhead with varying number of roles.



(b) Total data consumption with varying number of roles. `blk` and `trex` are flat.

Figure 7: Data overheads after 10000 updates determined to 1000 files. The *x* axis is log-scale, while *y* is not.

proach over the baseline case, where objects are shared among all intersecting roles. Metadata costs for the role-based approach are roughly twice those of the baseline case for all I size and up to 4 roles, but rises to nearly six times as much for 32 roles with 100% intersection.

Again, however, these numbers represent worst-case overheads. For example, 32 roles with 100% intersection describes a system with 32 absolutely identical roles, neither useful nor likely in practice.

Figure 7b shows the computed overhead in file data storage for the experiment described in Figure 7a, for three distinct types of systems. We use `ext4` as a straw-man to represent a generic whole-file forking approach. Assume a file *x* is present in both roles `friends` and `colleagues`, and then written in the context of `friends`. The whole-file approach of `ext4` would duplicate the entire file and then modify one version, resulting in an overhead of 100%. The more sophisticated `blk` represents files as ordered sets of fixed-size blocks [38, 19], and only duplicates blocks that differ. A 4KByte update might only modify a single file block. Finally, `trex` represents our prototype, which uses the `blk` approach, but also retains blocks from prior file versions. There are three different lines for `ext4`, as the overhead for the whole-file approach varies according to how much overlap there is among roles. Storage requirements of systems using a block-based approach, however, are unaffected by the number of roles. The storage costs of `trex` are slightly higher than those of `blk` because of the need to store old blocks.

6 T.Rex on the cloud

Thus far, we have presented T.Rex from the perspective of users who directly interact with one another. Recall from Figure 1, however, that many of the same issues encountered with local replicated services also present with cloud providers. Both data and metadata can leak to replicated cloud servers. The T.Rex mechanisms deal with these issues effectively, so it is natural to explore whether T.Rex mechanisms can be used effectively in concert with cloud providers.

T.Rex potentially addresses another issue with cloud services: consistency. Cloud services vary greatly in their guarantees. Some guarantee response times, some make *session guarantees* [47], and at least one makes global guarantees of single-object coherence.

T.Rex makes hard consistency guarantees, and could potentially be used to regularize guarantees across multiple clouds similarly to bolt-on consistency [7]. Our approach would differ in that the consistency guarantees would be made along with security guarantees.

A paper design T.Rex-Cloud would differ from standard T.Rex mostly in that all communication would be through the cloud. If each T.Rex replica is connected to the same cloud services account, we can effectively use the cloud as a fast and wide communication channel.

The use of cloud services would allow us to dispense with the anti-entropy protocol. Kernels and shards would be created as in T.Rex, and then deposited into cloud storage where they would be retrieved by other T.Rex replicas.

Clouds and shards contain encrypted dependency information. A replica pulling a shard out of the cloud

channel only *applies* if the consistency criteria in Section 3.1 are met. Security guarantees would also still be enforced, as kernels are created and encrypted as before.

Versus T.Rex, T.Rex-Cloud might have more durability; cloud updates would be present on many cloud servers, and might be backed up. T.Rex-Cloud also gets communication scheduling for free, and is simplified by the elimination of the anti-entropy protocol.

7 Related work

Many of the ideas in replicated storage systems were pioneered by Bayou [48, 36], which first used version vector-limited anti-entropy sessions to replicate data across weakly-connected devices. Our work is also inspired by work in personal data management systems, which use predicates to define high-level policies for data placement and device transparency [40, 42, 45, 39], replication [37], energy conservation and data movement [35, 34]. UIA [17] pioneered the idea of context-specific namespaces. More recently, HomeViews [18] mixed capabilities with SQL to create user-specific read-only views over data. This work identified security issues that would need to be solved in a multiple-user environment, but it is not a replicated storage system. PADS [10] and PRACTI [9] are a policy architecture and a replication framework that can be used to build highly flexible replicated storage systems. Finally, ZZFS [30], is another storage system with full metadata replication that its novelty is a new low-power network interface that is able to wake-up devices and inform them about new data placements.

As demonstrated in the previous paragraph, most of the work in this area concentrates on data placement, consistency, and coherence. The assumption is that security issues like access control, authentication, confidentiality are orthogonal to issues of data replication and consistency, and could be handled by mechanisms in higher levels. Another common assumption is that all participating devices have the same access control policies. In this work we have shown that straightforward application of these assumptions come with the cost of leaking information.

One exception is the access control extension of Cimbiosys [52] that defines access rules through SecPal [8]. This approach, when used with Cimbiosys [39], may be problematic because policies are propagated as regular objects, and Cimbiosys does not support eventual consistency. However, though SecPal statements are signed, data is unencrypted, the system assumes a single trusted authority, and the underlying system relies on a tree-structured replica topology.

Finally, our object coherence has similarities to Qufiles [51], and SUNDR [27]. Qufiles are a system abstraction that gives a server the ability to support different transcoded versions of a file under one common umbrella. SUNDR proposed *fork consistency*, which allows attacks to be constrained to forking version histories, which can then be detected.

At the other end of the spectrum are replicated object systems that tolerate Byzantine faults, like PBFT [11], Farsite [5], and even Oceanstore [25]. These systems differ from T.Rex in their use (T.Rex is designed for sharing and collaboration, with roles defined by high-level predicates) and in their goals (T.Rex adds information leakage to access control and confidentiality).

8 Conclusion

Traditional replicated data systems have assumed that all interacting users trust one another. In this paper, we have challenged this assumption by arguing that it can lead to suboptimal performance and, particularly in the case of cloud-based systems, can lead to information leakage. Moreover, we have demonstrated that *various consistency schemes can be efficient and secure without requiring all interacting users to trust one another*. T.Rex makes this possible by combining several novel mechanisms to create a cryptographically secure, efficient anti-entropy protocol. Our implementation and testbed evaluation of T.Rex demonstrate that it achieves security with modest computation and storage overheads.

We view T.Rex as the first step towards securing replicated storage systems; there remain many interesting open problems. Users worried about metadata leakage might also worry about visible communication paths and other subtle issues; combining T.Rex's techniques with anonymous communication systems like Onion Routing [21] is an interesting area of future work. We have sketched the design of a cloud-enabled T.Rex: one that would allow users to benefit from the reliability and availability of the cloud while maintaining control over the privacy and consistency of their data. The basic approach we take in doing so is to treat the cloud provider as a communication channel, one that potentially increases reliability. Another interesting area of future work is to understand how well consistency schemes "layer" upon one another. Moreover, with a system like T.Rex implementing end-to-end consistency, it is worth investigating whether cloud systems must invest in sophisticated consistency schemes of their own, or if the end-to-end argument [43] should be applied.

References

[1] 500px. http://500px.com.

[2] Libtomcrypt. `http://libtom.org`.

[3] Uthash. http://uthash.sourceforge.net.

[4] Zeromq. http://www.zeromq.org.

[5] ADYA, A., BOLOSKY, W. J., CASTRO, M., CHAIKEN, R., CERMAK, G., DOUCEUR, J. R., HOWELL, J., LORCH, J. R., THEIMER, M., AND WATTENHOFER, R. FARSITE: Federated, available, and reliable storage for an incompletely trusted environment. In *OSDI* (2002).

[6] AHAMAD, M., HUTTO, P. W., AND JOHN, R. Implementing and programming causal distributed shared memory. In *Proceedings of the 11th International Conference on Distributed Computing Systems* (May 1991).

[7] BAILIS, P., GHODSI, A., HELLERSTEIN, J. M., AND STOICA, I. Bolt-on causal consistency. In *SIGMOD* (2013).

[8] BECKER, M. Y., FOURNET, C., AND GORDON, A. D. Secpal: Design and semantics of a decentralized authorization language. *J. Comput. Secur. 18*, 4 (Dec. 2010), 619–665.

[9] BELARAMANI, N. M., DAHLIN, M., GAO, L., NAYATE, A., VENKATARAMANI, A., YALAGANDULA, P., AND ZHENG, J. PRACTI replication. In *NSDI* (2006), USENIX.

[10] BELARAMANI, N. M., ZHENG, J., NAYATE, A., SOULÉ, R., DAHLIN, M., AND GRIMM, R. PADS: A policy architecture for distributed storage systems. In *NSDI* (2009), J. Rexford and E. G. Sirer, Eds., USENIX Association, pp. 59–74.

[11] CASTRO, M., AND LISKOV, B. Practical byzantine fault tolerance. In *Proceedings of the third symposium on Operating systems design and implementation* (1999), OSDI '99.

[12] DHIRU KHOLIA, P. W. Looking inside the (drop) box. In *7th Usenix Wokshop on offensive Technologies* (2013).

[13] DROPBOX. Your stuff anywhere. `http://dropbox.com`.

[14] FAN, L., CAO, P., ALMEIDA, J., AND BRODER, A. Z. Summary cache: a scalable wide-area web cache sharing protocol. *IEEE/ACM Trans. Netw. 8*, 3 (June 2000), 281–293.

[15] FELDMAN, A. J., ZELLER, W. P., FREEDMAN, M. J., AND FELTEN, E. W. Sporc: group collaboration using untrusted cloud resources. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation* (Berkeley, CA, USA, 2010), OSDI'10, USENIX Association, pp. 1–.

[16] FERRAIOLO, D., AND KUHN, R. Role-based access control. In *National Computer Security Conference* (1992).

[17] FORD, B., STRAUSS, J., LESNIEWSKI-LAAS, C., RHEA, S., KAASHOEK, F., AND MORRIS, R. Persistent personal names for globally connected mobile devices. In *OSDI* (Seattle, Washington, Nov. 2006).

[18] GEAMBASU, R., BALAZINSKA, M., GRIBBLE, S. D., AND LEVY, H. M. Homeviews: peer-to-peer middleware for personal data sharing applications. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data* (2007), SIGMOD '07.

[19] GHEMAWAT, S., GOBIOFF, H., AND LEUNG, S. The Google file system. In *Proceedings of the nineteenth Symposium on Operating Systems Principles (SOSP'03)* (Bolton Landing, NY, USA, Oct. 2003), ACM, ACM Press, pp. 29–43.

[20] GOLDREICH, O., MICALI, S., AND WIGDERSON, A. How to play any mental game. In *Proceedings of the nineteenth annual ACM symposium on Theory of computing* (1987), ACM, pp. 218–229.

[21] GOLDSCHLAG, D., REED, M., AND SYVERSON, P. Onion routing. *Communications of the ACM 42*, 2 (1999), 39–41.

[22] KASSNER, M. Researchers reverse-engineer the dropbox client: What it means. `http://goo.gl/YVdguD`.

[23] KHOLIA, D. Long promised post module for hijacking dropbox accounts. `https://github.com/rapid7/metasploit-framework/pull/1497`, 2013.

[24] KIM, T. H.-J., BAUER, L., NEWSOME, J., PERRIG, A., AND WALKER, J. Challenges in access right assignment for secure home networks. In *Proceedings of the 5th USENIX conference on Hot topics in security* (Berkeley, CA, USA, 2010), HotSec'10, USENIX Association, pp. 1–.

[25] KUBIATOWICZ, J., BINDEL, D., CHEN, Y., CZERWINSKI, S., EATON, P., GEELS, D., GUMMADI, R., RHEA, S., WEATHERSPOON, H., WEIMER, W., WELLS, C., AND ZHAO., B. Oceanstore: An architecture for global-scale persistent storage. In *ASPLOS* (2000).

[26] LAMPORT, L., SHOSTAK, R., AND PEASE, M. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems (TOPLAS) 4*, 3 (1982), 382–401.

[27] LI, J., KROHN, M., MAZIÈRES, D., AND SHASHA, D. Secure untrusted data repository (sundr). In *Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation - Volume 6* (2004).

[28] LIPTON, R., AND SANDBERG, J. *PRAM: a scalable shared memory*. No. no. 180 in Research report // Princeton University, Department of Computer Science. Princeton University, Department of Computer Science, 1988.

[29] LLOYD, W., FREEDMAN, M. J., KAMINSKY, M., AND ANDERSEN, D. G. Don't settle for eventual: scalable causal consistency for wide-area storage with cops. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (2011), SOSP '11.

[30] MAZUREK, M. L., THERESKA, E., GUNAWARDENA, D., R.HARPER, AND SCOTT, J. Zzfs: A hybrid device and cloud file system for spontaneus users. In *FAST* (2012).

[31] MULAZZANI, M., SCHRITTWIESER, S., LEITHNER, M., HUBER, M., AND WEIPPL, E. Dark clouds on the horizon: Using cloud storage as attack vector and online slack space. In *USENIX Security Symposium* (2011).

[32] MUNISWAMY-REDDY, K.-K., HOLLAND, D. A., BRAUN, U., AND SELTZER, M. I. Provenance-aware storage systems. In *USENIX* (2006), USENIX, pp. 43–56.

[33] NEWTON, D. Dropbox authentication: insecure by design. `http://dereknewton.com/2011/04/dropbox-authentication-static-host-ids/`, 2011.

[34] NIGHTINGALE, E. B., AND FLINN, J. Energy-efficiency and storage flexibility in the blue file system. In *OSDI* (2004), pp. 363–378.

[35] PEEK, D., AND FLINN, J. Ensemblue: Integrating distributed storage and consumer electronics. In *OSDI* (2006), pp. 219–232.

[36] PETERSEN, K., SPREITZER, M. J., TERRY, D. B., THEIMER, M. M., AND DEMERS, A. J. Flexible update propagation for weakly consistent replication. In *Proceedings of the sixteenth ACM symposium on Operating systems principles* (New York, NY, USA, 1997), SOSP '97, ACM, pp. 288–301.

[37] POST, A., NAVARRO, J., KUZNETSOV, P., AND DRUSCHEL, P. Autonomous storage management for personal devices with podbase. In *Annual Technical Conference* (2011), USENIX.

[38] QUINLAN, S., AND DORWARD, S. Venti: A new approach to archival storage. In *Proceedings of the First USENIX conference on File and Storage Technologies* (Monterey,CA, January 2002), pp. 89–101.

[39] RAMASUBRAMANIAN, V., RODEHEFFER, T. L., TERRY, D. B., WALRAED-SULLIVAN, M., WOBBER, T., MARSHALL, C. C., AND VAHDAT, A. Cimbiosys: A platform for content-based partial replication. In *NSDI* (2009), J. Rexford and E. G. Sirer, Eds., USENIX Association, pp. 261–276.

[40] RIVA, O., YIN, Q., JURIC, D., UCAN, E., AND ROSCOE, T. Policy expressivity in the anzere personal cloud. In *Proceedings of the 2nd ACM Symposium on Cloud Computing* (2011), SOCC '11, ACM, pp. 14:1–14:14.

[41] RUFF, N., AND LEDOUX, F. A critical analysis of dropbox software security. *ASFWS 2012, Application Security Forum* (2012).

[42] SALMON, B., SCHLOSSER, S. W., CRANOR, L. F., AND GANGER, G. R. Perspective: semantic data management for the home. In *Proccedings of the 7th conference on File and storage technologies* (Berkeley, CA, USA, 2009), USENIX Association, pp. 167–182.

[43] SALTZER, J. H., REED, D. P., AND CLARK, D. D. End-to-end arguments in system design. *ACM Trans. Comput. Syst. 2*, 4 (Nov. 1984), 277–288.

[44] SOULES, C. A. N., AND GANGER, G. R. Connections: using context to enhance file search. In *Proceedings of the twentieth ACM symposium on Operating systems principles* (New York, NY, USA, 2005), SOSP '05, ACM, pp. 119–132.

[45] STRAUSS, J., PALUSKA, J. M., LESNIEWSKI-LAAS, C., FORD, B., MORRIS, R., AND KAASHOEK, F. Eyo: device-transparent personal storage. In *Proceedings of the 2011 USENIX conference on USENIX annual technical conference* (Berkeley, CA, USA, 2011), USENIXATC'11, USENIX Association, pp. 35–35.

[46] SUTHERLAND, G. Installing dropbox? prepare to lose aslr. `http://codeinsecurity.wordpress.com/2013/09/09/installing-dropbox-prepare-to-lose-aslr/`, 2013.

[47] TERRY, D. B., DEMERS, A. J., PETERSEN, K., SPREITZER, M. J., THEIMER, M. M., AND WELCH, B. B. Session guarantees for weakly consistent replicated data. In *PDIS '94: Proceedings of the third international conference on on Parallel and distributed information systems* (Los Alamitos, CA, USA, 1994), IEEE Computer Society Press, pp. 140–150.

[48] TERRY, D. B., THEIMER, M. M., PETERSEN, K., DEMERS, A. J., SPREITZER, M. J., AND HAUSER, C. H. Managing update conflicts in bayou, a weakly connected replicated storage system. *SIGOPS Oper. Syst. Rev.* (1995).

[49] TOLIA, N., KOZUCH, M., SATYANARAYANAN, M., KARP, B., PERRIG, A., AND BRESSOUD, T. Opportunistic use of content addressable storage for distributed file systems. In *Proceedings of the USENIX 2003 Annual Technical Conference* (San Antonio, TX, June 2003), pp. 127–140.

[50] VARDA, K. Protocol buffers: Google's data interchange format. http://google-opensource.blogspot.com/2008/07/protocol-buffers-googlesdata.html, 2008.

[51] VEERARAGHAVAN, K., FLINN, J., NIGHTINGALE, E. B., AND NOBLE, B. qufiles: the right file at the right time. In *FAST* (2010).

[52] WOBBER, T., RODEHEFFER, T. L., AND TERRY, D. B. Policy-based access control for weakly consistent replication. In *Proceedings of the 5th European conference on Computer systems* (New York, NY, USA, 2010), EuroSys '10, ACM, pp. 293–306.