How to Build a Trusted Database System on Untrusted Storage

Umesh Maheshwari Radek Vingralek William Shapiro

STAR Lab, InterTrust Technologies Corporation, Santa Clara, CA 95054
{umesh, rvingral, shapiro}@intertrust.com

Abstract

Some emerging applications require programs to maintain sensitive state on untrusted hosts. This paper presents the architecture and implementation of a trusted database system, TDB, which leverages a small amount of trusted storage to protect a scalable amount of untrusted storage. The database is encrypted and validated against a collision-resistant hash kept in trusted storage, so untrusted programs cannot read the database or modify it undetectably. TDB integrates encryption and hashing with a low-level data model, which protects data and metadata uniformly, unlike systems built on top of a conventional database system. The implementation exploits synergies between hashing and log-structured storage. Preliminary performance results show that TDB outperforms an off-the-shelf embedded database system, thus supporting the suitability of the TDB architecture.

1 Introduction

Some emerging applications require trusted programs to run on untrusted hosts. For example, vendors of digital goods such as software and music need to control the use of their goods according to their contracts with the consumers. The contracts may be enforced by executing a trusted program on the consumer's computer or playing device [SBV95, IBM00, Xer00].

Often, trusted programs need to maintain some sensitive, persistent state. For example, under a pay-per-use contract, the program may verify and debit the consumer's account. Or, under a limited-use trial, the program may count and limit the number of times the good is used. The amount of such state may grow with the number of vendors, goods, and the types of contracts. Furthermore, the sensitive nature of the state makes it desirable to protect it from both tampering and accidental corruption. Therefore, the state should be stored in a scalable and trusted database system.

Although a trusted program runs on the client, it could maintain its database on a trusted server for best security. However, this may require frequent communication between the trusted program and the server, which is constraining for devices with poor connectivity. Ideally, consumers should be able to use goods distributed on mass media or previously hoarded on their devices, even when they are disconnected from the network. Therefore, it is desirable to maintain the database on the client side.

The party hosting the database storage has the opportunity to alter its state for unauthorized benefits. For example, a consumer could save a copy of the local database, purchase some goods, then *replay* the saved copy, thus eliminating payments for the purchased goods.

It is difficult to secure a trusted program and its database because the hosting party ultimately controls the underlying hardware and the operating system. However, a number of emerging trusted platforms provide a processing environment that runs only trusted programs and resists reverse engineering and tampering. Such platforms employ a hardware package containing a processor, memory, and tamper-detecting circuitry [SPW98, KK99, Wav99, Dal00], or various techniques for software protection [Coh93, Auc96, CTL98]. However, these platforms do not provide trusted persistent storage in bulk because it is difficult to prevent read and write access to devices such as disk and flash memory from outside the trusted platform.

This paper presents the architecture and implementation of a trusted database system, TDB. By "trust" we mean *secrecy* (protection against reading from untrusted programs) and *tamper detection* (protection against writing from untrusted programs). An untrusted program cannot be prevented from tampering with the data, but such data fails validation when a trusted program reads it. This enables the trusted program to reject the data and perhaps refuse further operation.

TDB may also be used to protect a database stored at an untrusted server. Such a database may be used by client devices that do not have enough local storage. In this case, the user may have no incentive to tamper with the client device, so no explicit mechanisms may be required to provide a trusted platform on the client.

1.1 Basic Trust Management

TDB leverages a trusted processing environment and a small amount of trusted storage available on the platform. It provides secrecy by encrypting data with a key hidden in secret storage. It provides tamper detection by leveraging a small amount of tamper-resistant storage, as described below.

A common mechanism for validating data is to sign it with a secret key. However, signed data is vulnerable to replay attacks. The attack is easy because it does not require understanding the data; it works even when the data is encrypted. TDB resists replay attack by storing a collision-resistant hash of the database in tamper-resistant storage [MOV96]. When a trusted program writes and reads database objects, TDB updates and validates the database hash efficiently by maintaining a tree of hash values over the objects, as suggested by Merkle [Mer80].

TDB provides an option to use a tamper-resistant counter, which cannot be decremented, in place of generic tamper-resistant storage. After each database update, TDB increments the counter and generates a certificate containing the counter value and the database hash. The certificate is signed with the secret key and stored in untrusted storage.

1.2 Storage Management

To protect the state from accidental corruption, TDB provides standard database-system services such as crash atomicity, concurrent transactions, type checking, pickling, cache management, and index maintenance.

One might consider building a trusted database system by layering cryptography on top of a conventional database system. This layer could encrypt objects before storing them in the database and maintain a tree of hash values over them. This architecture is attractive because it does not require building a new database system. Unfortunately, the layer would not protect the metadata inside the database system. An attack could effectively delete an object by modifying the indexes. There could be some performance problems as well. For example, the database system could not maintain ordered indexes for range queries on encrypted data.

For these reasons, TDB applies hashing and encryption to a low-level data model, which protects data and metadata uniformly. It also enables TDB to maintain ordered indexes on data.

To protect the sensitive state from media failures such as disk crashes, TDB provides the ability to create backups and to restore valid backups. An attacker might fake a media failure and restore a backup to rollback the state. To limit the extent of a rollback, it is desirable to make frequent backups and disallow restoring old backups. TDB facilitates this by providing incremental backups [HMF99].

We discovered and exploited the synergy between the functions mentioned above and log-structured storage systems [RO91]. Log-structured systems have a comprehensive and hierarchical location map, because all objects are relocatable. Embedding the hash tree in the location map allows an object to be validated as it is located. The checkpointing optimization defers and consolidates the propagation of hash values up the tree. Copy-on-write using the location map provides cheap snapshots, which enables incremental backups. Furthermore, the absence of fixed object locations makes it hard to link multiple updates to the same object, thus resisting some traffic-monitoring attacks.

Preliminary performance results show that TDB outperforms a system that layers cryptography on top of an off-the-shelf database system. The database overhead is dominated by I/O; encryption and hashing represent only 6% of the total overhead.

1.3 Outline

The rest of this paper is organized as follows. Section 2 specifies the infrastructure TDB requires and the service it provides. Section 3 describes the overall architecture of TDB. Sections 4 and 5 describe the integration of encryption and hashing in a low-level data model. Section 6 describes backup creation and restores. Sections 7 and 8 briefly describe the construction of database functions over the low-level data model. Section 9 gives preliminary performance results. Section 10 describes potential extensions to TDB. Section 11 compares TDB with related work. Section 12 draws some conclusions.

2 System Specification

This section specifies the infrastructure TDB requires and the service it provides to applications.

2.1 Required Infrastructure

TDB requires a trusted platform that provides the following, as shown in Figure 1:

- *Trusted processing environment*, which executes only trusted programs and protects the volatile state of an executing program from being read or modified by untrusted programs. The static image of a trusted program need not be secret.

- *Secret store*: a small amount (e.g., 16 bytes) of read-only persistent storage that can be read only by a trusted program.
- *Tamper-resistant store*: a small amount (e.g., 16 bytes) of writable persistent storage that can be written only by a trusted program. Alternatively, the tamper-resistant store may be a counter that cannot be decremented. In either case, we assume that the tamper-resistant store can be updated atomically with respect to crashes.

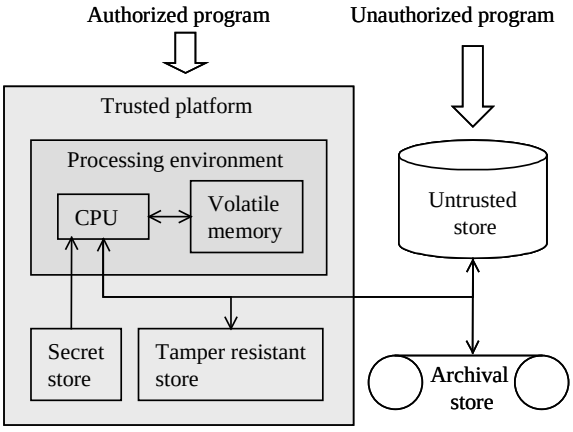

Figure 1: The trusted platform

The trusted platform may be a hardware package such as the IBM secure cooprocessor [SPW98], which contains a processor, battery-backed SRAM, DRAM, and ROM. The ROM firmware loads only trusted programs using a hash supplied during the manufacturing process. The battery-backed SRAM is zeroed if tampering is detected, so it can serve as both secret and tamper-resistant store.

The infrastructure also provides an *untrusted store* to hold the database. It is persistent, allows efficient random access, and can be read and written by any program. This might be a disk, flash memory, or an untrusted storage server connected to the trusted platform.

An *archival store* is needed to recover from the failures of the untrusted store. It is also untrusted. It need not provide efficient random access to data, only input and output streams. It might be a tape or an ftp server. We assume its failures are independent of the untrusted store.

We assume that suitable steps are taken when tampering is detected. The exact nature of such steps is outside the scope of this paper.

2.2 Service Provided

We list the functions of TDB below.

Trusted storage: TDB provides tamper-detection and secrecy for bulk data. This includes resistance to replay attacks and attacks on metadata.

Partitions: An application may need to protect different types of data differently. For example, it may have no need to encrypt some data or to validate other data. TDB allows an application to create multiple logical partitions, each protecting data with its own cryptographic parameters:

- a secret key
- a cipher (an encryption algorithm), e.g., 3DES
- a collision-resistant hash function, e.g., SHA-1

Using appropriate parameters avoids unnecessary time and space overhead. Using different secret keys reduces the loss from the disclosure of a single key. This should not be confused with *access control* among trusted parties, which may be provided in a higher layer, if needed.

Atomic updates: TDB can update multiple pieces of data atomically with respect to fail-stop crashes such as power failures.

Backups: TDB can back up a consistent snapshot of a set of partitions and restore a backup after validation. Backups allow recovery from media corruption. TDB provides fast *incremental* backups, which contain only changes made since a previous backup.

Concurrent transactions: TDB provides serializable access to data from concurrent transactions. Unlike shared databases or file servers, TDB is not designed for simultaneous access by many users. Therefore, its concurrency control is geared to low concurrency. It employs techniques for reducing latency, but lacks sophisticated techniques for sustaining throughput.

Database size: TDB allows the database to scale with gradual performance degradation. It uses scalable data structures and fetches data piecemeal on demand. However, it relies on a cacheable working set for performance because its log-structured storage may destroy physical clustering. Another limitation is its no-steal buffering of dirty data, which does not scale to transactions with many modifications [GR93].

Objects: TDB stores abstract objects that the application can access without explicitly invoking encryption, validation, and pickling. TDB pickles objects using application-provided methods so the stored representation is compact and portable.

Collection and Indexes: TDB provides index maintenance over *collections* of objects. A collection is a set

of objects that share one or more indexes. An index provides scan, exact-match, and range iterators.

3 System Architecture

TDB is designed for use on personal computers as well as smaller devices. The architecture is layered, so applications can trade off functionality for smaller code size. In Figure 2, boxes represent modules and arrows represent dependencies between them. Dashed boxes represent infrastructural modules.

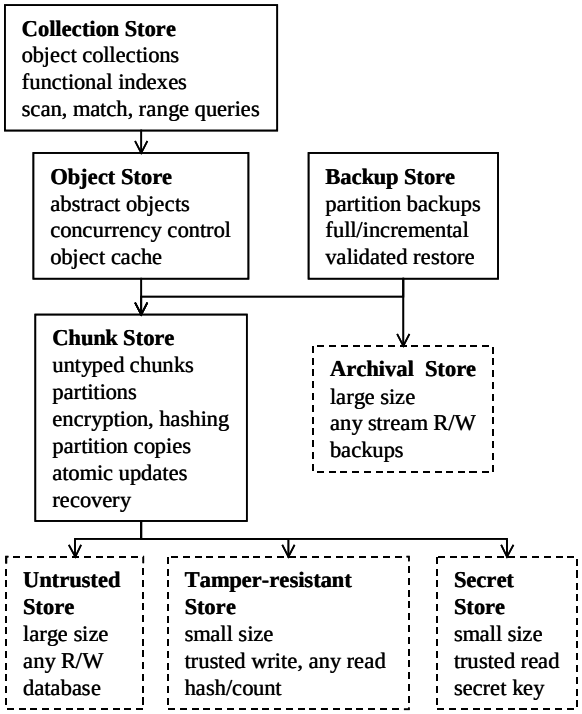

Figure 2: System architecture

The *chunk store* provides trusted storage for a set of named *chunks*. A chunk is a variable-sized sequence of bytes that is the unit of encryption and validation. (We expect chunk sizes between 100 bytes and 10 Kbytes.) All data and metadata from higher modules are stored as chunks. Chunks are logically grouped into partitions with separate cryptographic parameters. Partitions can be snapshot using the copy-on-write technique.

Chunks are stored in the untrusted store. The chunk store supports atomic updates of multiple chunks in the presence of crashes. It hides logging and recovery from higher modules. This architecture does not support logical logging, but the variable-sized chunks form a more compact log than fixed-sized pages.

The *backup store* creates and restores a set of partition backups. The chunk store and the backup store encapsulate secrecy and tamper-detection. This enables the higher modules to provide database management without worrying about trust.

The *object store* manages a set of named objects. It stores pickled objects in chunks—one or more objects per chunk. It keeps a cache of frequently-used or dirty objects. Caching data at this level is beneficial because the data is decrypted, validated, and unpickled. The object store also provides read transactional access to objects using read-write locking.

The *collection store* manages a set of named collections of objects. It updates the indexes on a collection as needed. Collections and indexes are themselves represented as objects.

This paper focuses on integrating trust with storage management in the chunk store and the backup store. It describes higher modules briefly to show that the chunk store is able to support them, and to explain a high-level performance benchmark we use.

4 Chunk Store: Single Partition

To simplify presentation, this section describes the chunk store as it would be in the absence of multiple partitions. Section 5 describes multiple partitions.

4.1 Specification

The chunk store manages a set of chunks named with unique ids. It provides the following operations:

- Allocate() returns ChunkId
  Returns an unallocated chunk id.
- Write(chunkId, bytes)
  Sets the state of chunkId to bytes, possibly of different size than the previous state. Signals if chunkId is not allocated.
- Read(chunkId) returns Bytes
  Returns the last written state of chunkId.
  Signals if chunkId is not written.
- Deallocate(chunkId)
  Deallocates chunkId.
  Signals if chunkId is not allocated.

Tamper Detection: In an idealized secret and tamper-*proof* chunk store, the operations listed above would be available only to trusted programs. Since tampering with the untrusted store cannot be prevented, the chunk store provides tamper-*detection* instead. It behaves like the tamper-proof store, except its operations may signal tamper detection if the untrusted store is tampered with.

Crash Atomicity and Durability: The write and deallocate operations are special cases of a *commit* operation. In general, a number of write and deallocate operations may be grouped into a single commit, which is atomic with respect to fail-stop crashes.

Allocated but unwritten chunks are deallocated automatically upon system restart. We have deliberately separated allocate and commit operations. An alternative is to allocate ids when new, unnamed chunks are committed. However, this alternative does not allow an application to store a newly-allocated chunk id in another chunk during the same commit operation, which may be needed for data integrity. Systems that swizzle application-provided references into persistent ids upon commit do not face this problem. However, the chunk store does not interpret application data chunks.

Concurrency Control: Operations are executed in a serializable manner. However, the chunk store is unaware of transactions. Allocate, read, and commit operations from different transactions may be interleaved.

4.2 Implementation Overview

This section gives an overview of the implementation; subsequent sections give further detail.

The chunk store writes chunks by appending them to a log in the untrusted store. As in other log-structured systems, chunks do not have static versions outside the log [RO91]. When a chunk is written or deallocated, its previous version in the log, if any, becomes obsolete.

The chunk store uses a *chunk map* to locate and validate the current versions of chunks. To scale to a large number of chunks, the chunk map is itself organized as a tree of chunks. Updates to the chunk map are buffered and written to the log occasionally. Updates lost upon a crash are recovered from the log.

Secrecy is provided by encrypting chunks with the key in the secret store. Tamper-detection is provided by creating a path of *hash links* from the tamper-resistant store to every current chunk version. We say there is a hash link from data x to y if x contains a hash of some data that includes y. If x is linked to y via one or more links using a collision-resistant hash function, it is computationally hard to change y without changing x or breaking a hash link [Mer80]. The hash links are embedded in the chunk map and the log.

Serializability of operations is provided through mutual exclusion, which does not overlap I/O and computation, but is simple and acceptable when concurrency is low.

4.3 Chunk Map

The chunk map maps a chunk id to a *chunk descriptor*, which contains the following information:

- status of chunk id: unallocated, unwritten, or written
- if written, current location in the untrusted store
- if written, expected hash value of chunk

Figure 3 shows the tree structure of the chunk map. The leaves are the chunks created by the applications of the chunk store; we call them *data* chunks. (These include chunks containing metadata of higher modules, for example, the indexing data of the collection store.) Each internal chunk, called a *map chunk*, stores a fixed-size vector of chunk descriptors. In the figure, each shaded slot is a chunk descriptor, and an arrow links the chunk containing the descriptor to the chunk described by the descriptor. The chunk at the top contains the descriptor of the root map chunk and some additional metadata needed to manage the tree; we call it the *leader* chunk. The descriptor of the leader chunk is retrieved at startup, as described later. The chunk store interprets map and leader chunks, but not data chunks.

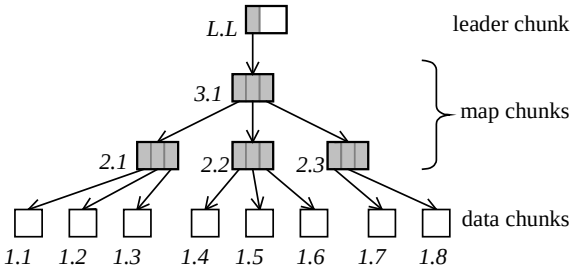

Figure 3: The chunk map

For uniformity of access and storage management, non-data chunks are also named using chunk ids. The id of a chunk encodes its *position* in the tree. The position comprises the height of the chunk in the tree and its rank from the left among the chunks at that height. In the figure, chunk ids are denoted as "*height.rank*". As the tree grows, new chunks are added to the right and to the top, which preserves the positions of existing chunks. (The position of the leader does change, so it is given a reserved id instead.) Besides unifying access to chunks, this approach enables id-based navigation of the map without storing ids in the map explicitly.

4.4 Allocate Operation

Ids of deallocated data chunks are reused to keep the chunk map compact and conserve id space. Deallocated ids are linked through a free list embedded in the descriptors. The head of the list is stored in the leader.

As mentioned, id allocation is not persistent until the chunk is written (committed). Upon system restart, chunk ids that were previously allocated but not written are made available in the free list for re-allocation.

4.5 Read Operation

Given a chunk id *c*, its state may be located and validated by traversing the path of descriptors from the leader to *c*. For each descriptor in the path, the chunk state is found as follows. The encrypted state is read from the location stored in the descriptor. It is decrypted using the secret key. The decrypted state is hashed. If the computed hash does not match that stored in the descriptor, tamper detection is signaled.

For better performance, the chunk map keeps a cache of descriptors indexed by chunk ids. Also, the leader chunk is pinned in the cache. The cached data is decrypted, validated, and unpickled.

If the descriptor for *c* is not in cache, the read operation looks for the descriptor of *c*'s parent chunk. Thus, the read operation proceeds *bottom up* until it finds a descriptor in the cache. Then it traverses the path back down to *c*, reading and validating each chunk in the path. This approach exploits the validated cache to avoid validating the entire path from the leader to the specified chunk.

4.6 Commit Operation

The commit operation hashes and encrypts each chunk to be written, and writes the encrypted state to the log in the untrusted store. We refer to the set of chunks written as the *commit set*.

When a chunk *c* is written or deallocated, its descriptor is updated to reflect its new location, hash, or status. Conceptually, this changes *c*'s parent chunk *d*; if *d* were also written out, its descriptor would be updated, and so on up to the leader, whose descriptor would be written to the tamper-resistant store. Instead, to save time and log space, the chunk store updates *c*'s descriptor in cache and marks it as dirty so it is not evicted. The bottom-up search during reads ensures that the stale descriptor stored in *d* is not used.

4.7 Checkpoint

When the cache becomes too large because of dirty descriptors, all map chunks containing dirty descriptors and their ancestors up to the leader are written to the log. This is done as a special commit operation called a *checkpoint*. In practice, checkpoints happen infrequently compared to regular commits. Other log-structured systems use similar checkpoints to defer and consolidate updates to the location map [RO91]. The chunk store extends the optimization to propagating hash values up the chunk map.

The leader is written last during a checkpoint. We refer to the part of the log written before the leader as the *checkpointed log* and the part including and after the leader as the *residual* log. Figure 4 shows a simple example, where the log tail contains some data chunks, possibly written in multiple commits, a checkpoint containing the affected map chunks and the leader chunk, and some more data chunks. Arrows link chunks as in Figure 3.

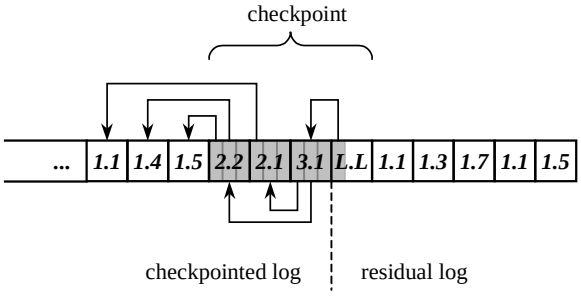

Figure 4: Checkpointing the chunk map

4.8 Recovery

A crash loses buffered updates to the chunk map, but they are recovered upon system restart by rolling forward through the residual log. Section 4.9 describes how the log is represented so the recovery procedure may find the sequence of chunks in the residual log.

For each chunk in the residual log, the recovery procedure computes the descriptor based on its location and hash, and puts the descriptor in the chunk-map cache. This procedure requires additional support from the commit operation to redo chunk deallocations and to validate the chunks in the residual log. This is described in the next two sections.

4.8.1 Chunk Deallocation

For each chunk to be deallocated, the commit operation writes a *deallocate chunk* to the log, which contains the id of the deallocated chunk.

Deallocate chunks are instances of *unnamed chunks*: they do not have chunk ids or positions in the chunk map. This is acceptable because they are used solely for recovery from the residual log and are always obsolete in the checkpointed log.

Like other chunks, unnamed chunks are encrypted with the secret key. They are also protected against tampering, as described in the next section. Otherwise, an at-

tack could cause a chunk to be un-deallocated. Or, an attack could replay the deallocation of a chunk id after it was re-allocated.

4.8.2 Validation of Residual Log

Although checkpointing defers the propagation of hash values up the chunk map, each commit operation must still update the tamper-resistant store to reflect the new state of the database. If the tamper-resistant store kept the hash of the leader and were updated only at checkpoints, the system would be unable to detect tampering with the residual log after a crash. We have implemented two approaches for maintaining up-to-date validation information in the tamper-resistant store.

4.8.2.1 Direct Hash Validation

The chunk store maintains a sequential hash of the residual log. The log hash is stored in the tamper-resistant store and updated after every commit. Upon recovery, the hash in the tamper-resistant store is matched against the hash computed over the residual log. This approach is illustrated in Figure 5.

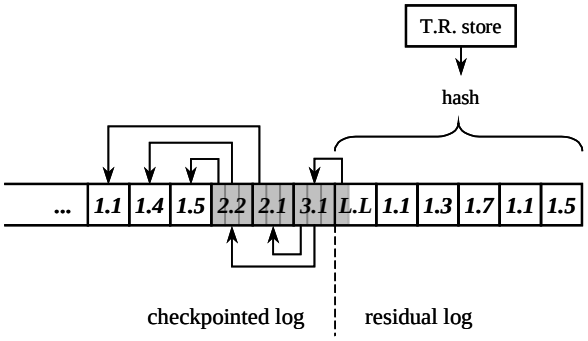

Figure 5: Tamper-resistant store contains database hash

A commit operation waits until the commit set is written to the untrusted store reliably before it updates the hash in the tamper-resistant store. Otherwise, a crash could leave the tamper-resistant store updated when the untrusted store is not, and cause validation to fail upon recovery. The update to the tamper-resistant store is the real commit point: If there is a crash during this update, the previous value stored in the tamper-resistant store is recovered, and the last commit set in the untrusted store is ignored. The commit operation returns after the tamper-resistant store is updated reliably.

Direct hash validation creates paths of hash links from the tamper-resistant store to all current chunk versions—in both the residual log and the checkpointed log. This is true because the tamper-resistant store is directly linked to all chunks in the residual log, which includes the leader from the last checkpoint, and the leader is linked through the chunk map to all current chunk versions in the checkpointed log. Note that all unnamed chunks in the residual log are linked as well. Unnamed chunks in the checkpointed log are not linked, which is not a weakness because all such chunks are obsolete.

4.8.2.2 Counter-based validation

In this approach, upon each commit, a sequential hash of the commit set is stored in an unnamed chunk added to the log, called the *commit chunk*. The commit chunk is signed with the secret key. (The signature need not be publicly verifiable, so it may be based on symmetric-key encryption [MOV96].) An attack cannot insert an arbitrary commit set into the residual log because it will be unable to create an appropriately signed commit chunk. Replays of old commit sets are resisted by adding a count to the commit chunk that is incremented after every commit. Deletion of commit sets at the tail of the log is resisted by storing the current commit count in the tamper-resistant store. This approach is illustrated in Figure 6.

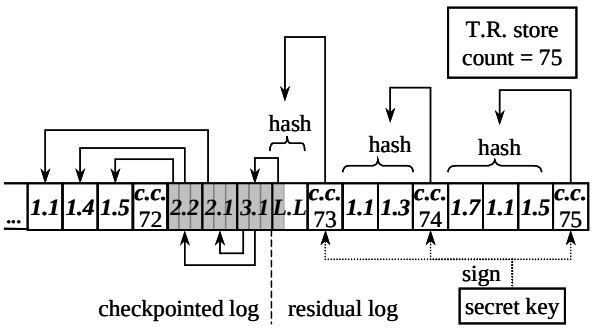

Figure 6: Tamper-resistant store contains commit count

A checkpoint is followed by a commit chunk containing the hash of the leader chunk, as if the leader were the only chunk in the commit set. The recovery procedure checks that the hash of each commit set in the residual log matches that stored in the commit chunk, and that the counts stored in the commit chunks form a sequence. Finally, the procedure compares the count in the last commit chunk with that in the tamper-resistant store. The hash-links created in this approach are similar to those in direct hash validation, except that the commit chunks are signed and linked from the tamper-resistant store through a sequence of numbers.

Counter-based validation has several advantages. First, the tamper-resistant counter is a weaker requirement than a generic tamper-resistant store. Provided the

counter cannot be decremented by *any* program, it does not need additional protection against untrusted programs. There is little incentive for untrusted programs to increment the counter because they would not be able to sign a commit chunk with the increased count.

Second, the commit count allows the system to tolerate bounded discrepancies between the tamper-resistant store and the untrusted store, if desired. For example, the system might allow the count in the tamper-resistant store, *t*, to be a little behind the last count in the untrusted store, *u*. This trades off security for performance. The security risk is that an attack might delete commit sets *t*+1 through *u*. The performance gain is that a commit operation need not wait for updating the count in the tamper-resistant store, provided (*u*-*t*) is smaller than some threshold Δ_{ut}. This is useful if the tamper-resistant store has high update latency. The system might also allow *t* to leap *ahead* of *u* by another threshold Δ_{tu}. This admits situations where the untrusted store is written lazily (e.g., IDE disk controllers often flush their cache lazily) and the tamper-resistant store might be updated before the untrusted store. The only security risk is the deletion of at most Δ_{tu} commit sets from the tail of the log.

A drawback of counter-based validation is that tamper detection relies on the *secrecy* of the key used to sign the commit chunk. Therefore, if a database system needed to provide tamper-detection but not secrecy, it would still need a secret store.

4.9 Log Representation

This section describes the structure of the data written to the log. The log consists of a sequence of chunks; we refer to the representation of a chunk in the log as a *version*.

4.9.1 Chunk Versions

Chunk versions are read for three different functions:

- Read operation, which uses the chunk id and the descriptor to read the current version.
- Log cleaning, which reads a *segment* of the checkpointed log sequentially.
- Recovery, which reads the residual log sequentially.

To enable sequential reading, the log contains information to identify and demarcate chunks. Each chunk version comprises a header followed by a body. The header contains the chunk id and the size of the chunk state. The header of an unnamed chunk contains a reserved id. Both the header and the body are encrypted with the secret key. Similarly, the hash of the residual log or a commit set covers both headers and bodies.

4.9.2 Head of Residual Log

The recovery procedure needs to locate the head and the tail of the residual log. The head of the residual log is the leader. Its location is stored in a fixed place, as in other log-structured storage systems. It need not be kept in tamper-resistant store: With direct hash validation, tampering with this state will change the computed hash of the residual log. With counter-based validation, it is possible for an attack to change the location to the beginning of another commit set. Therefore, the recovery procedure checks that the chunk at the stored location is the leader.

Because the location of the leader is updated infrequently—upon each checkpoint—storing it at a fixed location outside the log does not degrade performance. This location is written after the writes to the untrusted store and the tamper-resistant store have finished. Its update marks the completion of the checkpoint. If there is a crash before this update, the recovery procedure ignores the checkpoint at the tail of the log.

4.9.3 Tail of Residual Log

With direct hash validation, the location of the log tail may be stored in the tamper-resistant store along with the database hash. This works well because the write to the tamper-resistant store is the true commit point.

With counter-based validation, it is possible to infer the location of the tail from the log itself, as in conventional databases [GR93]. The last commit set in the log may have been corrupted in a crash. The hash stored in a commit chunk serves well as a checksum for the commit set. The recovery procedure stops when the hash of a commit set does not match the hash stored in the commit chunk.

4.9.4 Segments

The untrusted store is divided into fixed-size segments to aid cleaning, as in Sprite LFS [RO91]. The segment size is chosen for efficient reading and writing by the cleaner, e.g., on the order of 100 KB for disk-based storage. A segment is expected to contain many chunk versions. The size of a chunk version cannot exceed the segment size. A commit set may span multiple segments.

The log is represented as a sequence of potentially non-adjacent segments. Since the recovery procedure needs to read the residual log sequentially, segments in the residual log contain an unnamed *next-segment chunk* at the end, which contains the location of the next segment.

4.9.5 Log Cleaning

The log cleaner reclaims the storage of obsolete chunk versions and compacts the storage to create empty segments. It selects a segment to clean and determines whether each chunk version is current by using the chunk id in the header to find the current location in the chunk map. It then commits the set of current chunks, which rewrites them to the end of the log [BHS95].

The set of steps from selecting a segment to committing the current chunks happens atomically with respect to externally invoked operations. The cleaner may be invoked synchronously when space is low, but it is mostly invoked asynchronously during idle periods.

The cleaner does not clean segments in the residual log, because that would destroy the sequencing of the residual log. This also resolves what the cleaner should do with unnamed chunks, because they are always obsolete in the checkpointed log. For performance reasons, the cleaner selects segments with low utilization. Details on the utilization metric and the maintenance of this information are beyond the scope of this paper.

The cleaner need not validate the chunks read from the segment provided the commit operation for rewriting current chunks does *not* update the hash values stored in chunk descriptors. If the hashes are recomputed and updated, as they would be in a regular commit, the cleaner must validate the current chunks; otherwise, the cleaner might launder chunks modified by an attack. Because of its simplicity, we have implemented the second, less efficient, approach.

5 Chunk Store: Multiple Partitions

This section describes extensions to the chunk store that provide multiple partitions and partition copies. Multiple partitions enable the use of different cryptographic parameters for different types of data. Partition copies enable fast backups.

5.1 Specification

The chunk store manages a set of named partitions, each containing a set of named chunks. A chunk id comprises the chunk position, as before, and the id of the containing partition. (A chunk in one partition may have the same position as another chunk in another partition.) The chunks in a partition are protected with the parameters associated with it.

The following partition operations are provided:

- Allocate() returns PartitionId
  Returns an unallocated partition id.
- Write(partitionId, secretKey, cipher, hashFunction)
  Sets the state of partitionId to an empty partition with the specified cryptographic parameters.
- Write(partitionId, sourcePId)
  Copies the current state of sourcePId to partitionId. Each chunk in sourcePId is logically duplicated in partitionId at the same position.
- Diff(oldPId, newPId) returns set<ChunkPosition>
  Returns a set containing chunk positions whose state is different in newPId and oldPId.
- Deallocate(partitionId)
  Deallocates partitionId and all of its copies, and all chunks in these partitions.

Furthermore, the chunk allocate operation requires the id of the partition in which the chunk is to be created. A commit operation may include a number of write and deallocate operations on both partitions and chunks. This makes it possible, for example, to store the id of a newly-written partition into a chunk in an existing partition in one atomic step.

The next few sections describe how the extended specification is implemented.

5.2 Multi-partition Chunk Map

Figure 7 shows the structure of the multi-partition chunk map. Each written partition has a *position map*, which maps a chunk position in the partition to a descriptor. This map is like the single-partition map described in Section 4.3. The map chunks in the position map of partition *P* belong to *P*: their partition id is *P* and they are protected using *P*'s cryptographic parameters. In the figure, chunk ids are denoted as *partition:position*.

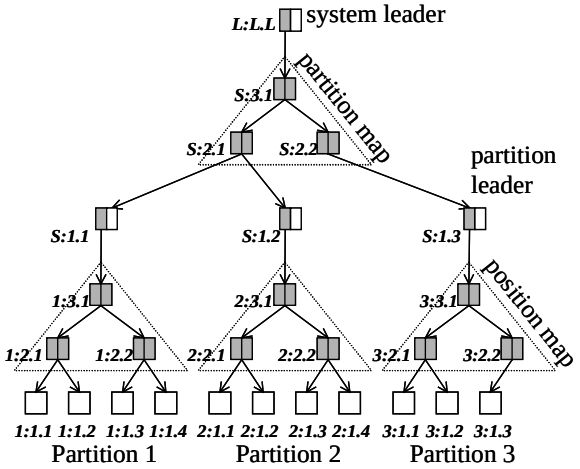

Figure 7: Multi-partition chunk map

The leader chunk for a partition contains information needed to manage the position map, as before, and the cryptographic parameters of the partition, including the secret key. The *partition map* at the top maps a partition id to the partition leader. This map is managed like the position map of a special partition, called the *system* partition, which has a reserved id denoted *S* in the figure. The partition leaders are the data chunks of the system partition and are protected using the cryptographic parameters of the system partition. Many partition operations such as allocating a partition id or reading a partition leader translate into chunk-level operations on the system partition.

Chunks in the system partition and the system leader are protected using a fixed cipher and hash function that are considered secure, such as 3DES and SHA-1 [MOV96]. They are encrypted with the key in the secret store. Thus, secrecy is provided be creating a path of *cipher links* from the secret store to every current chunk version. We say that there is a cipher link from one piece of data to another if the second is encrypted using a key stored in the first.

5.3 Partition Copies and Diffs

To copy a partition *P* to *Q*, the chunk store copies the contents of *P*'s leader to *Q*'s leader. Thus, *Q* and *P* share both map and data chunks, and *Q* inherits the cryptographic parameters of *P*. Thus, partition copies are cheap in space and time.

When chunks in *P* are updated, the position map for *P* is updated, but that for *Q* continues to point to the chunk versions at the time of copying. The chunks of *Q* can also be modified independently of *P*, but the common use is to create a read-only copy, called a *snapshot.*

The chunk store diffs two partitions by traversing their position maps and comparing the descriptors of the corresponding chunks. Commonly, diffs are performed between two snapshots of the same partition.

5.4 Log Representation

A commit set may contain chunks from different partitions. A chunk body is encrypted with the secret key and cipher of its partition. However, chunk headers are encrypted with the system key and cipher, so that cleaning and recovery may decrypt the header without knowing the partition id of the chunk.

The system leader is the head of the residual log, so it is linked from the tamper-resistant store. The residual log is hashed using the system hash function. Thus, each chunk in a commit set is hashed twice: once with its partition-specific hash function to update the chunk descriptor, and once with the system hash function to update the log hash. In principle, the log hash could be computed over the partition-specific hashes of chunk bodies. However, a weak partition hash function could then invalidate the use of the log hash as a checksum for recovery (see Section 5.4). For simplicity, and because hashing is relatively fast, we chose to keep the hashes separate.

5.5 Cleaning and Recovery

Checking whether a chunk version is current is complicated by partition copies. A chunk header contains the id of the partition *P* to which it belonged when the chunk was written. Even if the version is obsolete in *P*, it may be current in some direct or indirect copy of *P*. Therefore, each partition leader stores the ids of its direct copies and the cleaner checks for current-ness in the copies, recursively. The process would be more complex had it not been that the deallocation of a partition deallocates the partition's copies as well.

Suppose the cleaner rewrites a chunk version identified as *P:x* that is current only in partitions *Q* and *R*. The commit procedure updates the descriptors for *Q:x* and *R:x* in the cache. Further, in order that the recovery procedure is able to identify the chunk correctly, the cleaner appends an unnamed *cleaner chunk*, which specifies that the chunk is current in both *Q* and *R*.

6 Backup Store

The backup store creates and restores *backup sets*. A backup set consists of one or more *partition backups*. The backup store creates backup sets by streaming backups of individual partitions to the archival store and restores them by replacing partitions with the backu ps read from the archival store.

6.1 Backup Consistency

The backup store guarantees consistency of backup creation and restore with respect to other chunk store operations. Instead of locking each partition for the entire duration of backup creation, the backup store creates a consistent snapshot of the source partitions using a single commit operation. It then copies the snapshots to archival storage in the background. We assume that restores are infrequent, so it is acceptable to stop all other activity while a restore is in progress.

6.2 Backup Representation

Partition backups may be *full or incremental*. A full partition backup contains all data chunks of the partition. An incremental backup of a partition is created

with respect to a previous snapshot, the *base*, and contains the data chunks that were created, updated, or deallocated since the base snapshot. Backups do not contain map chunks since chunk locations in the untrusted store are not needed. Chunks in a backup are represented like chunk versions in the log.

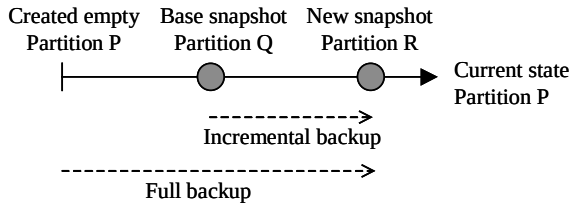

Figure 8: Full and incremental backups

A partition backup contains a *backup descriptor*, a sequence of chunk versions, and a backup signature. The backup descriptor contains the following (illustrated using partition ids from Figure 8):

- id of source partition (*P*)
- id of partition snapshot used for this backup (*R*)
- id of base partition snapshot (*Q*, if incremental)
- backup set id (a random number assigned to the set)
- number of partition backups in the backup set
- partition cipher and hasher
- time of backup creation

The representation of partition backups is illustrated below. Here, H_s denotes the system hash function, H_p denotes the partition hash function, E_s denotes system cipher using the system key, and E_p denotes the partition cipher using the partition key.

PartitionBackup ::=
    E_s(BackupDescriptor)
    (E_s(ChunkHeader) E_p(ChunkBody))*
    BackupSignature
    Checksum

BackupSignature ::=
E_s(H_s(BackupDescriptor H_p((ChunkId ChunkBody)*)))

The backup signature binds the backup descriptor with the chunks in the backup and guarantees integrity of the partition backup. The unencrypted checksum allows an external application to verify that the backup was written completely and successfully.

6.3 Backup Restore

The backup store restores a backup by reading a stream of one or more backup sets from the archival store. The backup store restores one partition at a time, enforcing the following constraints:

- Incremental backups are restored in the same order as they were created, with no missing links in between. This is enforced by matching the base partition id in the backup descriptor against the id of the previous restored snapshot for the same partition.
- If a partition backup is restored, the remaining partition backups in the same backup set must also be restored. This is enforced by matching the number of backups with a given set id against the set size recorded in backup descriptors.

After reading the entire backup stream, the restored partitions are atomically committed to the chunk store. Backup restores require approval from a trusted program, which may deny frequent restoring or restoring of old backups.

7 Object Store

The *object store* adds safety against errors in application programs. It provides type-safe and transactional access to a set of objects. An object is the unit of typed data accessed by the application. The object store implements two-phase locking on objects and breaks deadlocks using timeouts. Transactions acquire locks in either shared or exclusive mode. We chose not to implement granular or operation-level locks because we expect only a few concurrent transactions. The object store keeps a cache of frequently-used or dirty objects. Caching data at this level is beneficial because the data is decrypted, validated, and unpickled.

The object store could store one or more pickled objects in each chunk. We chose to store each object in a different chunk because it results in a smaller volume of data that must be encrypted, hashed, and written to the log upon a commit. In addition, the implementation of the cache is simplified since no chunk can contain both committed and uncommitted objects. On the other hand, storing each object in a different chunk destroys inter-object clustering and increases the database size due to per-chunk overhead (see Section 9.3). Because we expect much of the working set to be cached, the lack of inter-object clustering is not important.

8 Collection Store

The *collection store* provides applications with indexes on *collections* of objects. A collection is a set of objects sharing one or more indexes. Indexes can be dynamically added and removed from each collection. Collections and indexes are themselves represented as objects.

The collection store supports *functional indexes* that use keys extracted from objects by deterministic functions [Hwa94]. The use of functional indexes allows us to avoid a separate data definition language for the database schema. Indexes are maintained automatically as

objects are updated. Indexes may be unsorted or sorted, which is possible because the objects are decrypted.

9 Performance

In this section we describe preliminary performance measurements. First, we present the performance on chunk and backup store operations based on several micro-benchmarks. Then we compare the performance an off-the-shelf database system and TDB using a higher-level benchmark.

9.1 Platform

Performance was evaluated on a 450 MHz Pentium PC with 128 MB of RAM, running the Windows NT 4.0 operating system. TDB is written in C++.

The untrusted store was implemented as an NTFS file on a hard disk with 9 ms average seek and 7200 rpm (4 ms average rotational latency). Using a raw disk partition would be more efficient, but we do not expect the users of TDB to provide one. The total size of TDB caches (including the object cache and the chunk-map cache) was set to 4 Mbytes.

The tamper-resistant store was emulated with an NTFS file on another hard disk to avoid interference with accesses to the untrusted store. This disk has 12 ms average seek and 5200 rpm (6 ms average rotational latency). The access time is similar to that for writing EEPROM, 5 ms [Inf00].

We used counter-based validation and allowed the count in the tamper-resistant store to lag behind that in untrusted store by $\Delta_{ut} = 5$. The tamper-resistant store is flushed only once is Δ_{ut} commits. The untrusted store is flushed upon every commit and we set Δ_{tu} to 0.

9.2 Micro-benchmarks

This section presents the performance of basic cryptographic, disk, chunk store and backup store operations.

9.2.1 Cryptographic and Disk Operations

Encryption: We used 3DES in CBC mode for the system partition, which has a measured bandwidth of 2.5 MB/s (0.4 µs per byte). We used DES in CBC mode for other partitions; the measured bandwidth is 7.2 MB/s (0.14 µs per byte). There are other, more secure, algorithms that run faster than DES [MOV96].

Hashing: We used SHA-1. The measured bandwidth is 21.1 MB/s (0.05 µs per byte). Additionally, the "finalization" of a hash value has a fixed overhead of 5 µs.

Store latency: While the disk specs provide average latency, the measured latency varies widely based on the position of disk head. Furthermore, the latency of the NTFS flush operation for files larger than 512 bytes is doubled because it writes file metadata separately. We measured write latencies of 10 ms to 20 ms for small files and 25 ms to 40 ms otherwise. Therefore, we shall focus on the computational overhead and denote the latencies of the untrusted and tamper-resistant store symbolically as l_u and l_t.

Store bandwidth: The measured bandwidth, b_u, of reading or writing the NTFS file implementing the untrusted store varies between 3.5 and 4.7 MB/s.

9.2.2 Chunk Store Operations

We repeated each operation 10 times and found that the computational overhead does not vary much, typically deviating less than 2%.

Allocate chunk id: This operation does not change the persistent state. The average latency is 6 µs.

Write chunks + commit: We committed sets of 1 to 128 chunks of sizes 128 bytes to 16 KB per chunk, which covers the range we expect. The computational latency, measured using linear regression, is 132 µs + 36 µs per chunk + 0.24 µs per byte of cumulative chunk size. The fixed overhead comes largely from processing the commit chunk (pickling, encrypting, hashing, etc.), the per-chunk overhead from processing the chunk header and finalizing the chunk's hash value, and the per-byte overhead from encryption and hashing the chunk bodies. The I/O overhead is $l_u + l_t/\Delta_{ut} + 1/ b_u$ per byte, which usually dominates the computational overhead.

Read chunk: If the chunk descriptor is cached, the computational latency of reading a chunk is 47 µs + 0.18 µs per byte of chunk size. The fixed overhead comes largely from processing the chunk header and finalizing the hash, and the per-byte overhead from decryption and hashing. The I/O overhead is $l_u + 1/b_u$ per byte. If the descriptor is not cached, the read operation reads in parental map chunks up to one whose descriptor is cached. In our experiments, each map chunk has 64 descriptors and has a size of 1.5 KB.

Write partition + commit: The computational latency of committing a new partition is 223 µs. The computational latency of copying a partition is 386 µs, regardless of the number of chunks in the source partition, owing to our use of the copy-on-write technique.

9.2.3 Backup Store Operations

We benchmarked only backup creation, we assume that backup restore performance is not critical.

Partition backup: We used 512 byte chunks. The computational latency to create an incremental backup of a partition is 675 μs + 9 μs per chunk in the backed up partition + 278 μs per updated chunk. The fixed overhead comes mostly from creating the partition snapshot and processing the backup descriptor and signature. The overhead per chunk in the backed up partition comes from diff-ing the snapshot of the backed up partition against the base snapshot. The overhead per updated chunk comes from copying the chunk.

The size of a backup determines the I/O overhead for writing it. The size of an incremental backup is 456 B + 528 B per updated chunk, which may be significantly less than the size of a full backup.

9.3 Space Overhead

The chunk descriptor, header, and padding add an overhead of about 52 bytes for chunks encrypted using an 8-byte block cipher. The additional overhead per chunk due to the chunk map is small because the fanout degree of the tree is large (64). Obsolete chunk versions in the log add additional overhead. When cleaning in idle periods, the space utilization may be kept as high as 90% with reasonable performance [BHS95].

9.4 Code Complexity

Figure 9 gives the complexity of TDB in terms of number of semicolons in C++ code.

Module	*semicolons*
Collection store	1,388
Object store	512
Backup store	516
Chunk store	2,570
Common utilities	1,070
TOTAL	6,056

Figure 9: TDB code complexity

9.5 Performance Comparison

In this section, we compare the performance of a system using either TDB or an off-the-shelf embedded database system, which we shall call XDB. The XDB-based system layers cryptography on top of XDB. We configured both systems to use the same cryptographic parameters, cache size, and frequency of flushing the tamper-resistant store.

9.5.1 Workload

We measured the performance on a benchmark that models two operations related to vending digital goods:

- **Bind**: A vendor binds three alternative contracts to a digital good.
- **Release**: A consumer releases the digital good selecting one of the three contracts randomly.

The benchmark first creates 30 collections for different object types. Each collection has one to four indexes. The benchmark loads the cache before executing an experiment. The experiment consists of 10 consecutive bind or release operations. Figure 10 gives the number of database operations executed in each experiment.

	read	update	delete	add	commit
release	781	181	10	41	96
bind	1732	733	10	220	292

Figure 10: Number of database operations.

9.5.2 Comparison Results

We repeated each experiment 10 times. Figure 11 shows the average times for the release and bind experiments, the part spent in the database system, and the part thereof spent in commit, which is the major overhead.

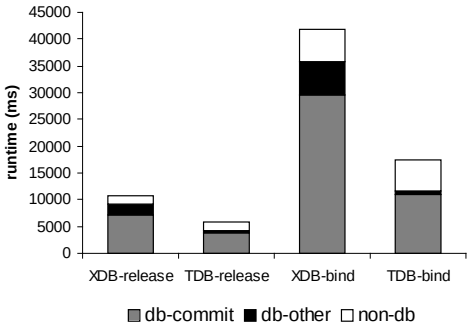

Figure 11: Runtime comparison

TDB outperformed XDB, primarily because of faster commits, but also in the remaining database overhead. We believe that XDB performs multiple disk writes at commit.

The stored size of XDB after running the release experiment was 3.8 MB. The stored size of TDB was 4.0 MB, based on 60% maximum log utilization.

9.5.3 TDB Performance Analysis

Here, we analyze the performance of the release experiment. Figure 12 breaks down the TDB overhead by

module. The time reported for each module excludes nested calls to other reported modules. The figure gives the average time (μ), the standard deviation (σ), and percentage of total (%).

module	μ*(ms)*	σ*(ms)*	*%*
DB TOTAL	4209	484	100
collection store	162	0	4
object store	85	0	2
chunk store	61	1	1
encryption	157	1	4
hashing	93	5	2
untrusted store read	8	0	0
untrusted store write	3353	164	81
tamper-resistant store	229	46	6

Figure 12: TDB runtime analysis

The overhead is dominated by writes to the untrusted store. The experiment flushed the untrusted store 96 times and the tamper-resistant store 19 times. The overhead of writing to the tamper-resistant store may vary significantly depending on the device and the frequency of flushes. There was no checkpoint or log cleaning during the experiment. (In the bind experiment, log cleaning took a total of 1030 ms.)

The overhead of encryption and hashing is only 6% of the database overhead. The effective bandwidths of encryption and hashing are 6.5 MB/s and 20.6 MB/s, which are close to the peak bandwidths reported in Section 9.2.1.

10 Potential Extensions

The current design of TDB has a number of limitations. Below we describe extensions to address them.

Untrusted storage on servers: TDB may be used to protect a database stored at an untrusted server. This application of TDB may benefit from additional optimizations for reducing network round-trips to the untrusted server, such as batching reads and writes.

Trusted paging. The current design assumes that the entire runtime, volatile state of a trusted program is protected by the trusted processing environment. TDB limits its volatile state by controlling its cache size, but this limit is not hard. Therefore, some volatile state may have to be paged out to untrusted storage. This problem may be solved by using a page fault handler to store encrypted and validated pages in the chunk store.

Steal buffer management. Currently, modified objects must remain in the cache until their transaction commits, which may degrade the security and performance of large transactions. Evicting dirty objects would require writing them to the log. This requires additional support in the chunk store.

Logical logging. Logical logging may reduce the volume of data that must be encrypted, hashed, and written to the untrusted store. The chunk store uses logical logging for some operations (for example, deallocation of chunks), but it does not allow higher modules to specify operations that should be logged logically.

11 Related Work

There are many systems aimed at providing secure storage. TDB differs from most of them because of its unique trust model.

In another paper at this conference, Fu et al. describe a read-only file system that may be stored in untrusted servers [FKM00]. A hash tree is embedded in the inode hierarchy. The trusted creator signs the root hash with the time of update and expiration. This system is not designed to handle frequent updates or updates to individual file blocks in the untrusted server.

Techniques for securing audit logs stored on weakly-protected hosts are suitable for securing append-only data that is read infrequently and sequentially by a trusted computer [BY97, SK98]. They employ a linear chain of hash values instead of a tree. When the data needs to be read, it is validated by recomputing the hash over the entire log. These techniques are not suitable for a database system such as ours, which requires frequent and random read-write access to data.

Blum et al. considered the problem of securing various data structures in untrusted memory using a hash tree rooted in a small amount of trusted memory [BEG+91]. This work does not address storage management for persistent data.

Some systems provide secure storage by dispersing data onto multiple hosts, with the expectation that at least a certain fraction of them (for example, two-thirds) will be honest. The data may be replicated as-is for time efficiency [CL99], or it might be encoded to reduce the cumulative space overhead [Rab89, Kra93, GGJ+97]. Read requests are broadcast to all machines and the data returned is error corrected. This approach provides *recovery* from tampering, not merely tamper detection. However, it relies on more trusted resources than are available to TDB. The expectation of an honest quorum is based on the assumption that, under normal operation, the hosts are weakly protected but not hostile, so

the difficulty for a hostile party to take over *k* hosts increases significantly with *k*.

Our use of log-structured storage builds on a previous work on log-structured storage systems [RO91, JKH93]. The Shadows database system is log structured and provides snapshots [Ylo94]. Otherwise, there has been little interest in log-structured database systems, perhaps because of the need to keep large sets of data physically clustered or to keep the log compact using logical logging.

12 Conclusions

We have presented a trusted database system that leverages a trusted processing environment and a small amount of trusted storage to extend tamper-detection and secrecy to a scalable amount of untrusted storage. The architecture integrates encryption and hashing with a low-level data model, which protects data and metadata uniformly. The model is powerful enough to support higher-level database functions such as transactions, backups, and indexing.

We found that log-structured storage is well suited for building such a system. The implementation is simplified by embedding a hash tree in the comprehensive location map that is central to log-structured systems: objects can be validated as they are located. The checkpointing optimization defers and consolidates the propagation of hash values up the tree. Because updates are not made in place, a snapshot of the database state can be created using copy-on-write, which facilitates incremental backups.

We measured the performance of TDB using microbenchmarks as well as a high-level workload. The database overhead was dominated by writes to the untrusted store and the tamper-resistant store, which may vary significantly based on the types of devices used. The overhead of encryption and hashing was only 6% of the total. On this workload, TDB outperformed a system that layers cryptography on an off-the-shelf embedded database system, while also providing more protection. This supports the suitability of the TDB architecture.

Acknowledgements

Olin Sibert and Susan Owicki motivated us to work on this problem. Various members of STAR Lab and InterTrust provided useful comments and help with performance measurement. Our shepherd, Frans Kaashoek, guided us in improving the presentation.

References

[Auc96] D. Aucsmith. Tamper resistant software: an implementation. In *Proc. International Workshop on Information Hiding*, Lecture Notes in Computer Science, Vol. 1174, Cambridge, UK, 1996, pp. 317-333.

[BEG+91] M. Blum, W. Evans, P. Gemmel, S. Kannan, and M. Naor. Checking the correctness of memories. In *Proc. IEEE Conf. on Foundations of Computer Science*, San Juan, Puerto Rico, 1991, pp. 90-99.

[BHS95] T. Blackwell, J. Harris, and M. Seltzer. Heuristic cleaning algorithms in log-structured file systems. In *Proc. USENIX Technical Conference*, New Orleans, LA, 1995, pp. 249-264.

[BY97] M. Bellare and B. Yee. Forward integrity for secure audit logs. *Technical Report*, Computer Science and Engineering Department, University of California at San Diego, 1997.

[Coh93] F. Cohen. Operating system protection through program evolution. In *Computers & Security, 12(6), Oxford*, 1993.

[CL99] M. Castro and B. Liskov. Practical byzantine fault tolerance. In *Proc. Symposium on Operating Systems Design and Implementation*, New Orleans, LA, 1999, pp. 173-186.

[CTL98] C. Collberg, C. Thomborson, and D. Low. Manufacturing cheap, resilient, and stealthy opaque constructs. In *Proc. ACM Principles of Programming Languages*, San Diego, CA, 1998, pp. 184-196.

[Dal00] Dallas Semiconductor secure microcontroller family, http://www.dalsemi.com/products/micros/secure.html, August 2000.

[FKM00] K. Fu, F. Kaashoek, and D. Mazieres. Fast and secure distributed read-only file system. To appear in *Proc. Symposium on Operating Systems Design and Implementation*, San Diego, CA, 2000.

[GGJ+97] J. Garay, R. Gennaro, C. Jutla, and T. Rabin. Secure distributed storage and retrieval. In *Proc. Intl. Workshop on Distributed Algorithms*, Berlin, Germany, 1997, pp. 275-289.

[GR93] J. Gray and A. Reuter. *Transaction processing: concepts and techniques*. Morgan Kaufmann Publishers, 1993.

[HMF+99] N. Hutchinson, S. Manley, M. Federwisch, G. Harris, D. Hitz, S. Kleiman, S. O'Malley. Logical vs.

physical file backup. In *Proc. Symp. on Operating System Design and Implementation*, New Orleans, LA, 1999, pp. 239-249.

[Hwa94] D. Hwang. Function-based indexing for object-oriented databases. *PhD thesis*, Massachusetts Institute of Technology, 1994.

[Inf00] Infineon Technologies. Eurochip II—SLE 5536, http://www.infineon.com/cgi/ecrm.dll/ecrm/scripts/prod_ov.jsp?oid=14702&cat_oid=-8233, August 2000.

[Int00] InterTrust Technologies Corp. Digital rights management. http://www.intertrust.com/de/index.html, August 2000.

[IBM00] IBM. Cryptolope technology, http://www.software.ibm.com/security/cryptolope, August 2000.

[JKH93] W. Jonge, M. F. Kaashoek, W. Hsieh. The logical disk: a new approach to improving file systems. In *Proc. ACM Symposium on Operating Systems Principles*, Asheville, NC, 1993, pp. 15-28.

[KK99] O. Kommerling and M. Kuhn. Design principles for tamper-resistant smartcard processors. *In Proc. USENIX Workshop on Smartcard Technology*, Chicago, IL, 1999.

[Kra93] H. Krawczyk. Distributed fingerprints and secure information dispersal. In *Proc. ACM Symp. on Principles of Distributed Computing*, Ithaca, NY, 1993, pp. 207-218.

[Mer80] R. Merkle. Protocols for public key cryptosystems. In *Proc. IEEE Symposium on Security and Privacy*, Oakland, CA, 1980, pp. 122-134.

[MOV96] A. Menezes, P. van Oorschot, and S. Vanstone. *Handbook of applied cryptography*. CRC Press, 1996.

[Rab89] T. Rabin. Efficient dispersal of information for security, load balancing, and fault tolerance. *Journal of the ACM*, 36(2), 1989, pp. 335-348.

[RO91] M. Rosenblum and J. Ousterhout. The design and implementation of a log-structured file system. In *Proc. ACM Symposium on Operating Systems Principles*, Pacific Grove, CA, 1991, pp. 1-15.

[SBV95] O. Sibert, D. Bernstein, and D. Van Wie. DigiBox: a self protecting container for information commerce. In *Proc. USENIX Conference on Electronic Commerce*, New York, NY, 1995, pp. 171-186.

[SK98] B. Schneier and J. Kelsey. Cryptographic support for secure logs on untrusted machines. In *Proc. USENIX Security Symposium*, San Antonio, TX, 1998, pp. 52-62.

[SPW98] S. Smith, E. Palmer, and S. Weingart. Using a high-performance, programmable secure coprocessor. In *Proc. Intl. Conf. on Financial Cryptography*, Anguilla, British West Indies, 1998.

[Wav99] Wave Systems Corp. The Embassy e-commerce system. http://www.wave.com/technology/Embassywhitepaper.pdf, August 2000.

[Xer00] ContentGuard, Rights management from Xerox, http://www.contentguard.com, August 2000.

[Ylo94] T. Ylonen. Shadow paging is feasible. *Licentiate's thesis*, Helsinki University of Technology, 1994.