immudb: Lightweight, high-speed immutable database

vChain Inc, Houston, TX

Jerónimo Irazábal *jeronimo@vchain.us*
Dennis Zimmer *dennis@vchain.us*
Michele Meloni *michele@vchain.us*
Valentin Padurean *valentin@vchain.us*

Abstract

immudb is a lightweight yet highly scalable immutable database. Providing low-latency and high-throughput comparable to a raw key-value store but ensuring any tampering is not only properly identified but scoped.
A well-defined update protocol between clients and server provides an on-demand tampering detection while built-in corruption and consistency checkers give rise to the first-ever proactive tampering awareness database.
This article includes immudb architecture, a brief description of main components, an explanation of the protocol for ensuring immutability and future work.

1 Introduction

Dealing with limitless data, processing, storing and querying is challenging, especially for sensitive data, where any tampering could lead to significative loss. Traditional approaches merely relying on Public Key Infrastructure in enterprise-grade contexts may be a problem on its own and it may be impractical in highly scalable scenarios. Attempts of using Blockchain failed because those platforms were not defined for tampering detection in enterprise-grade solutions and they introduce significant scalability and operational issues due to their complexity. Thus a more scalable and comprehensible solution is required. This is where immudb shines, with the simplicity of a traditional cloud native database but built from the ground up with tampering awareness capabilities, makes it possible to deal with sensitive data at large.
immudb security is not about role-based access, it's not about being an open source project, it's about the mathematical algorithms and state update protocol it employs. Whenever immudb is able to provide a satisfactory proof against the returned data, it means data is origin and was not tampered by any means.
Internally, immudb stores data together with a hash data-structure used to generate cryptographic proofs. The additional data structure that is generated can be thought of as a mutable merkle tree [7] and an update protocol run by clients and server is always made against the current state of immudb.
While immudb shares some technical similarities with blockchain platforms, immudb is not meant to provide smart-contracts or de-centralized governance capabilities. But provides fully-automated tools to accomplish enterprise-grade immutability, ensuring any tampering on sensitive data is not only detected but scoped.
Immutability is ensured by following a well-defined update protocol. Clients and auditors continously evaluate the state update provided by the server. Such evaluation is based on cryptographic proofs, meaning validation is matematically determined, making it impossible to fake or circunvent, resulting in any tampering being inmediately detected.

2 immudb Architecture

immudb server 1 provides a secured and authenticated multi-database support. Grpc [3] services are exposed for all its functionality, built-in metrics are served for Prometheus server collection [4] and proactive corruption checking.
At the storage layer, immudb implements an extended key-value model in which data is stored, key-indexed and bounded with a mutable merkle tree which root denotes the current and accumulative state of the database. The state is accumulative because its value depends on every change ever made within the database.
immudb server itself employs an internal immutable database to provide support user management and other capabilities (*sysdb*). Meaning that tampering detection features are also provided for immudb server as well.
Following section describes the internals of immudb storage layer, which determines not only the data model but also the immutability and tamper-awareness capabilities.

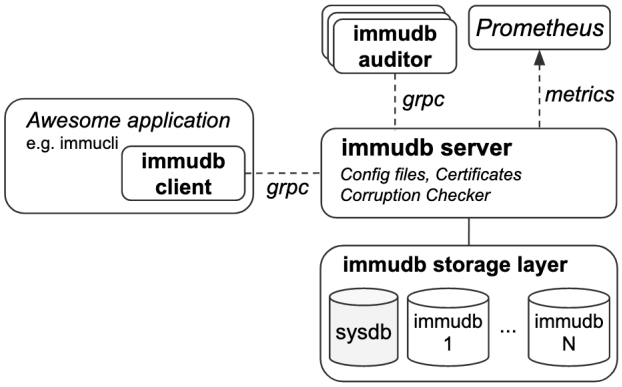

Figure 1: immudb Architecture: a single immudb server supports multiple databases.

3 Immutable Storage

At the lowest layer, where the actual data and indexing information resides, [1]. Indexing in immudb is twofold, to provide fast write and query operations and to guarantee any tampering is detected.

At this layer, input data consists of key-value pairs and each key-value pair is augmented with a numeric value denoting the insertion order 4.

While tampering detection can be as simple and efficient as calculating a hash function over the entire data, it does not provide any practical way for a client to actually verify that some specific key-value pair was tampered or not. To overcome this limitation, the hash calculation is done by using a hierarchical data-structure.

immudb implements a mutable merkle tree 2 in which only the latest tree is needed to construct proofs regarding any data ever written. The merkle tree grows from the bottom up to the root with any key-value addition, the order of insertion associated to the key-value pair having a one-to-one mapping with the leaves of the tree.

Given the merkle tree is kept on sync with any change, the root is determined by the entire history of changes. Therefore the root of the mutable merkle tree is considered the accumulative state of the database. It's worth to mention that subtrees in the mutable merkle tree get eventually fixed and not further update is applied to any of its nodes. In fact, the exact number of updates a given node will have is determined by the maximal number of leaves any of its childs could hold. Which is equivalent to state that any subtree gets fixed once it becomes a binary complete tree.

The advantage of constructing a hierarchical data-structure by using unpredictable, deterministic and efficient functions i.e. cryptographic hash function from the actual data (or a digest

[1] immudb currently uses Badger [1] for both data and indexing

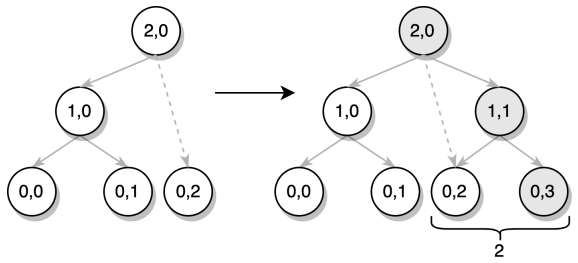

Figure 2: Mutable Merkle Tree: Appending a leaf may generate or update up to $log(N)$ nodes.

of it), is that for re-calculating any node in the tree, the exact values of its children has to be known. Recursively, it means the exact value of a leaf must be known in order to recompute the root of the merkle tree, but only the sibling values are required.

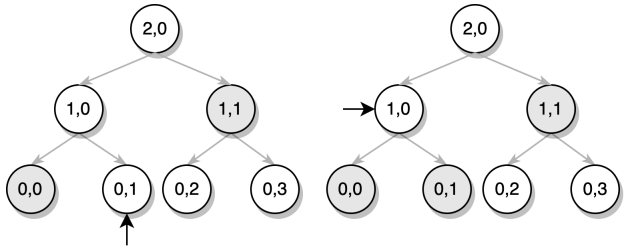

Figure 3: Cryptographic Proofs: Inclusion and Consistency proofs consist of providing the required data to calculate the root of the tree.

While the explanation above provides a quite general description of what a cryptographic proof means 3. There are two algorithms which are used to construct proofs in different contexts[2]:

Inclusion proof minimal data, including a designated leaf, required to re-compute the root of a containing mutable merkle tree.

Consistency proof minimal data required to re-compute a given subtree root and the root of a containing mutable merkle tree.

In practical terms it means that clients receive a subset of nodes to calculate the current root of the tree from any key-value entry (*Inclusion proof*) and a subset of nodes to calculate the current root from any previous root (*Consistency proof*).

Following section describes how immudb makes use of its

[2] Cryptographic proofs implemented in immudb are based on the works [2, 6].

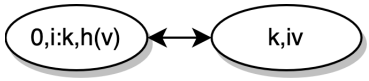

Figure 4: Indexes: key-indexed value and order of insertion, order-index key and value digest.

immutable storage layer to ensure any tampering is properly detected.

4 Immutability guarantees

Immutability goes beyond data corruption detection due to unexpected software or hardware failures. It's about generation of irrefutable proofs to demonstrate any subset of data was not accidentally or bad-intentionally tampered or erased.
immudb ensures any change produces an unforgettable mutation which gets carried over any future database state.
immudb accumulative state, cryptographic signatures and independent verifications are the fundamental pieces to ensure immutability for enterprise-grade applications.
immudb fulfills:

- Generation of irrefutable proofs to demonstrate inclusion of any ever written data.
- Any deliberate change or tampering produces an unforgettable mutation in the database accumulative state.

Above described capabilities conforms a solid ground for the implementation of on-demand and proactive methods of verification to ensure any tampering attempt is detected.

4.1 State update protocol

Immutability as described in previous section is guaranteed by immudb clients and auditors following a well-defined protocol. The objective of the protocol is to ensure every change in the database is unforgettable.
immudb clients are responsible of safe-guarding the newest database state validated so far. Every time a newer state is seen, the verification of the newer state can be done by re-calculating from the previous one. For this calculation to be possible, a cryptographic proof is provided by immudb

Algorithm 1 SafeGet (*Key*)

1: $KV_i \leftarrow Get(Key)$
2: $ReadLock()$
3: $Root_n \leftarrow GetCurrentRoot()$
4: $IProof \leftarrow InclusionProof(i,n)$
5: $CProof \leftarrow ConsistencyProof(i,n)$
6: $ReadUnlock()$
7: **return** $(KV_i, Root_n, IProof, CProof)$

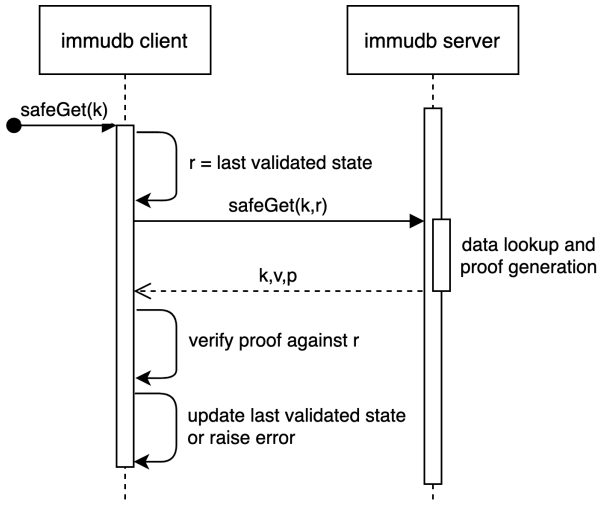

Figure 5: SafeGet fetches key-value data together with cryptographic proofs.

in the form of a consistency proof. immudb clients run an on-demand validation as part of the execution of any secured operation e.g. *safeGet*, *safeSet*. Figure 5 depicts the steps executed by an immudb client during a *safeGet* operation. If an immudb client is not provided with a satisfactory proof, it means the database has to be carefully analyzed.
immudb auditors behavior is as specific as useful. They periodically validate that the current database state can be re-calculated from the latest already validated one.
The number of auditors and deployment procedures may change depending on trust requirements. Auditors are a special type of light-weight immudb clients which can run anywhere with minimal requirements.
It's even possible to leverage the state update protocol with application specific business logic e.g. clients can request the changes between any two database states and execute application-specific validations before updating to the newly seen database state.
The combination of cryptographically signed accumulative states together with independent verifications makes it possible for any single client or auditor to detect and report any tampering.

4.2 Corruption Checker

immudb server has build-in corruption verification. Similar to the state update protocol and ensuring actual key-value pairs are in consistency with the mutable merkle tree. This additional verification may be useful when the actual storage layer is managed independently from the server process or for the server to self-detect tampering based on previously validated root values.

Corruption detection can be interpreted as a generalization of inclusion proof, by which non-tampering is demonstrated for a range of entries.

4.3 Accumulative database state

As already described, immudb appends a new leaf to the merkle tree with every change. Thus the root represents the complete history of changes ever made and given the tree calculation relies on *SHA-256* as cryptographic hash function, the root will differ even with the smallest tampering.

5 Data Model and Query capabilities

immudb foundational data-model consist of a totally ordered double-indexed set of key-value pairs. Data may be retrieved by key (or key prefixes) and by insertion order.
Basic exposed API is analogous to a key-value store excepts for the possibility to retrieve data from a numeric position. immudb supports batching operations, sequential scanning and sorted-sets similar to the ones provided by Redis [5].
List below describes a basic set of operations provided by immudb:

```
Set(Key, Value) Pos
Get(Key) (Key, Pos, Value)
Get(Pos) (Key, Pos, Value)
History(Key)
```

In addition to basic store operations, immudb incorporates new type operations reflecting the underneath immutability capabilities.

```
Root() Root
ConsistencyProof(Root) Proof

SafeGet(Key) (Key, Pos, Value, Proof)
SafeSet(Key, Value) (Key, Pos, Value, Proof)
```

Auditors rely on *Consistency Proof* operation to demonstrate the immutable history of the database.
Clients may fetch data with *SafeGet* operation, which in addition to provide the data associated to a key, includes cryptographic proofs to demonstrate the returned data was not tampered.

6 Ongoing and future work

Lot of new capabilities are already in progress or planned for the near future:

- Language bindings for the most used languages today
- Significantly reduce key-value store footprint
- Improve and enhance the corruption checker
- High availability and sharding
- Richer query support

7 Acknowledgments

The authors gratefully acknowledge researchers at Rice University, Scott A. Crosby and Dan S. Wallach, for their contributions on the data structures and algorithms highly relevant for immudb project.

Project

immudb is an open source software, available on

https://github.com/codenotary/immudb

More information about immudb is available on

https://immudb.io

References

[1] Badger: Fast key-value db in go. https://github.com/dgraph-io/badger.

[2] Certificate transparency. https://tools.ietf.org/html/rfc6962.

[3] grpc: A high performance, open-source universal rpc framework https://grpc.io/.

[4] Prometheus: Monitoring system and time series database. https://github.com/prometheus.

[5] Redis: in-memory data structure store. https://github.com/redis.

[6] CROSBY, S., AND WALLACH, D. Efficient data structures for tamper-evident logging. pp. 317–334.

[7] MERKLE, R. C. A digital signature based on a conventional encryption function. In *Advances in Cryptology - CRYPTO '87, A Conference on the Theory and Applications of Cryptographic Techniques, Santa Barbara, California, USA, August 16-20, 1987, Proceedings* (1987), vol. 293 of *Lecture Notes in Computer Science*, Springer, pp. 369–378.