immudb: A Lightweight, Performant Immutable Database

CodeNotary Inc., Houston, TX

Michael Paik
mpaik@codenotary.com

Jerónimo Irazábal
jeronimo@codenotary.com

Dennis Zimmer
dennis@codenotary.com

Michele Meloni
michele@codenotary.com

Valentin Padurean
valentin@codenotary.com

Abstract

More of the world's activity is being recorded by digital services, which has resulted both in an increasingly stringent compliance and regulatory environment for data storage and attacks on that storage that are growing in sophistication and subtlety. This paper describes immudb, an append-only general purpose database that, in concert with other security best practices, provides tamper evidence and irrepudiable transactions while maintaining performance appropriate for high-volume applications.

immudb uses Merkle Hash Trees (MHTs) to create digests that represent the state of the entire database at any given time. Together with cryptographic proofs that demonstrate that 1) a given element has been successfully inserted into the database and 2) a database is consistent between two points in time, these provide robust guarantees about the validity of the state of the database.

Simultaneously, immudb provides data read access using versioned key-value and insertion order APIs that enable immudb to serve in a broad range of capacities that would otherwise use less secure alternatives.

1 Introduction and Motivation

In July 2020, Mandiant Solutions, a cybersecurity research company, released a report describing a Russian hacking campaign against targets in Poland, Lithuania, and Latvia that had been ongoing since at least March 2017 [27]. The hacking group(s) associated with this campaign, which Mandiant entitled "Ghostwriter", had, among other attack vectors, penetrated the content management systems that support various news outlets' web publications in the target nations. Subsequently, the attackers had posted fabricated news articles intended to sway public opinion against the North Atlantic Treaty Organization (NATO), in an apparent bid to weaken its geopolitical influence in three nations that were either former Soviet Socialist Republics or, in the case of Poland, a member of the Warsaw Pact. Rather than simply post new fraudulent articles, which would have been detected quickly due to the recency bias of the news cycle, the hackers replaced older articles which would not appear on the front pages, but would be returned in searches for, e.g. "NATO."

While this attack is significant in that it features a state-sponsored actor, it belongs to a larger group of threats in which attackers, whether internal or external to an organization, gain access to a data store and modify or delete data, instead of or in addition to exfiltrating it for other purposes. This can take myriad forms: modifying transaction histories to remove debits, tweaking clinical trial research data, deleting receipts or inflating invoices in support of theft, altering navigation waypoints, changing grades, etc.

Simultaneously, the burden of regulation for data retention is heavy. Major sets of regulations that impose data security rules include but are certainly not limited to HIPAA [5] (healthcare data), FERPA [12] (student records), FCRA [16] and its amendment FACTA [17] (consumer credit), SOX [13] (institutional accounting), and 21 CFR § 11 (pharmaceutical trial data), and so forth. The burden is also continuously increasing, with recent years seeing the European Union enact the General Data Protection Act [7], New York enact the SHIELD act [14], and California enact the California Consumer Privacy Act [4]. Many of these regulations levy specific rules requiring audit trails for data modifications.

Together, these two coinciding forces strongly indicate the need to create a new class of database that is auditable, irrepudiable, and tamper-evident *by design* rather than attempting to retrofit existing database solutions with additional instrumentation[1].

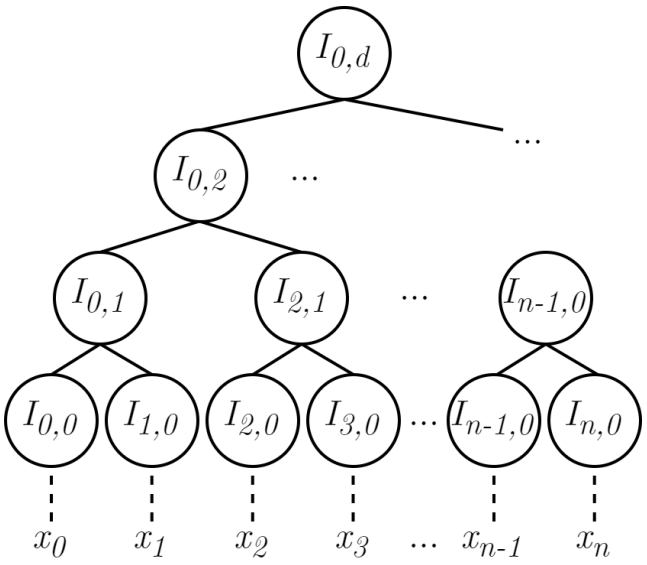

Figure 1: An example of a classic Merkle Hash Tree. Each data element x_i is hashed and put into leaf node $I_{i,0} = \mathrm{H}(x_i)$. Each pair $I_{i,0}, I_{i+1,0}$, $\frac{2}{i} \in \mathbb{Z}$ is concatenated, hashed, and placed in interior tree node $I_{i,1}$. Each successive interior tree node consists of the hash of the left-to-right concatenation of the value of its children until a single root node $I_{0,d}, d = \log_2 n$ is created, representing a digest of the entire tree's state.

2 Background

2.1 Merkle Hash Trees

In his 1979 thesis [28] and again in a 1987 paper [29], Ralph C. Merkle described what have come to be called Merkle Trees or Merkle Hash Trees (MHTs), as illustrated in Figure 1. These trees were conceived as digital signatures, attested to in a public shared file, to some underlying data. The primary motivation of the signature scheme at the time was to provide cryptographically secure signatures under certain circumstances while avoiding the computational expense (significant at the time) of using asymmetric key encryption by using computationally cheap hash functions instead. A contemporary understanding of the operation of these trees is as follows[2]:

1. Some n data elements x_0 through x_n is presented for signature. In Merkle's original conception, these are groups of message bits. For simplicity we assume that $n \bmod 2 = 0$.
2. Each of these n elements is digested using a hash function H, and we refer to each $\mathrm{H}(x_i)$, $0 \leq i \leq n$ as X_i for notational convenience. These X_i values form the leaves of the MHT, each X_i contained in a leaf node $I_{i,0}$ denoting its index i and its depth at the leaf layer, 0. More generally, for each node $I_{i,j}$, i is the index of the leftmost element in the subtree rooted at $I_{i,j}$, and j is the node's depth measured in path traversals up (i.e toward the root) from the leaf layer.
3. Assuming a binary tree, each pair of hashes at the leaf layer are concatenated and hashed, so $I_{0,1} = \mathrm{H}(I_{0,0}||I_{1,0})$. We thus assemble $I_{0,1}$, $I_{2,1}$, ... $I_{n-1,1}$ into $\frac{n}{2}$ trees, each of which is a hash of the concatenation of the hashes of two original elements. We repeat this process $I_{0,2} = \mathrm{H}(I_{0,1}||I_{2,1})$ and so on until we create a single tree with root node $I_{0,\log_2 n}$.

The resultant tree has several properties that will prove to be important. First, the root node $I_{0,\log_2 n}$ is a digest of the entire tree, including all of the original data elements $x_0 ... x_n$; thus the alteration of any data element will alter the values at the roots of any subtree containing that value, including the overall tree. Second, the path from the root to any given leaf $I_{i,0}$ is unique. Third, given a data element x_i, and a tree root value, it is possible to construct a proof that x_i is in the tree using a series of interior node values.

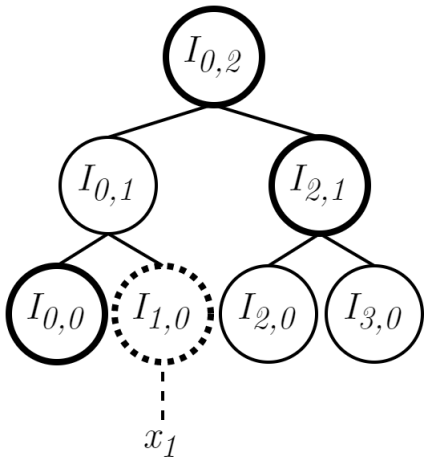

Figure 2: An inclusion proof for some tree with root $I_{0,2}$ and data element x_i. The dashed circle indicates a derivable value; heavy circles are the values that must be provided to prove that x_i was used in the construction of the tree (specifically as the ith element).

2.2 Inclusion Proofs

Figure 2 illustrates the procedure for generating a proof of inclusion. The value of an MHT's root is the hash of the left-

[1]The evaluation of such approaches is outside the scope of this paper, but see [30, 31] for some examples. Generally, these are not performant and require an external trusted party.

[2]The notation here is not perfectly self-documenting, but follows [25] for consistency.

to-right concatenation of its children's values, and each of its children's values are the left-to-right concatenation of their children's, and so forth. In the case of a complete binary tree with depth 3 and 4 data elements, if we wish to prove the inclusion of x_1 we need to provide the missing information to allow someone to generate $I_{0,2}$ given x_1.

$$\begin{aligned} I_{0,2} &= \mathrm{H}(I_{0,1}||I_{2,1}) \\ &= \mathrm{H}(\mathrm{H}(I_{0,0})||I_{1,0})||I_{2,1}) \\ &= \mathrm{H}(\mathrm{H}(I_{0,0})||\mathrm{H}(x_1)||I_{2,1}) \end{aligned} \tag{1}$$

The elements needed to show that $I_{0,1}$ (and by extension x_1) was used at its indicated position to generate the tree rooted at $I_{0,2}$ are thus sibling $I_{0,0}$, its parent's sibling $I_{2,1}$, and the root. Thus the inclusion proof is the set $\{I_{0,0}, I_{2,1}, I_{0,2}\}$. More generally, for a given x_i, the inclusion proof set is its sibling and the sibling of each parent moving towards the root, which suffice to calculate the root from x_i. Proof validation is merely the calculation of the analog to Equation 1 above. This demonstrates an additional important property of the tree - inclusion proofs do not need to contain any data elements, preserving privacy of those data.

2.3 Mutable Merkle Hash Trees

Merkle's original description of MHTs was intended for use as part of a signature scheme, with a static message agreed upon by two parties out of band. Crosby and Wallach [25] and, independently, Laurie et al. [6], proposed procedures to extend MHTs to append-only data structures that maintain MHTs' desirable security invariants. In these systems, as some element x_n is added to a tree of n elements, it is appended to the tree and hashes toward the root recalculated, as illustrated in Figure 3.

Thus, rather than a complete recalculation of the tree, resulting in $O(n)$ hashes, only $O(\log_2 n)$ hashes are recalculated on insertion to a tree of size n.

Both Crosby and Laurie make the modification of prepending a byte indicator to the hash input function of `0x00` if the hash result is a leaf and `0x01` if it is an internal node. This serves to fend off second preimage attacks in which an attacker could present a tree in which nodes at some depth>0 are presented as leaves, with their values being the concatenated hashes of their actual children, producing the same root value as the original valid tree, but rendering the data garbled.

2.4 Consistency Proofs

The introduction of growing MHTs also introduces a new class of incremental consistency proofs that demonstrate that some tree root value I_q is a digest of a tree built from a valid superset of some other tree whose digest is I_p. That is, I_q contains I_p plus additional data elements. Such a proof consists

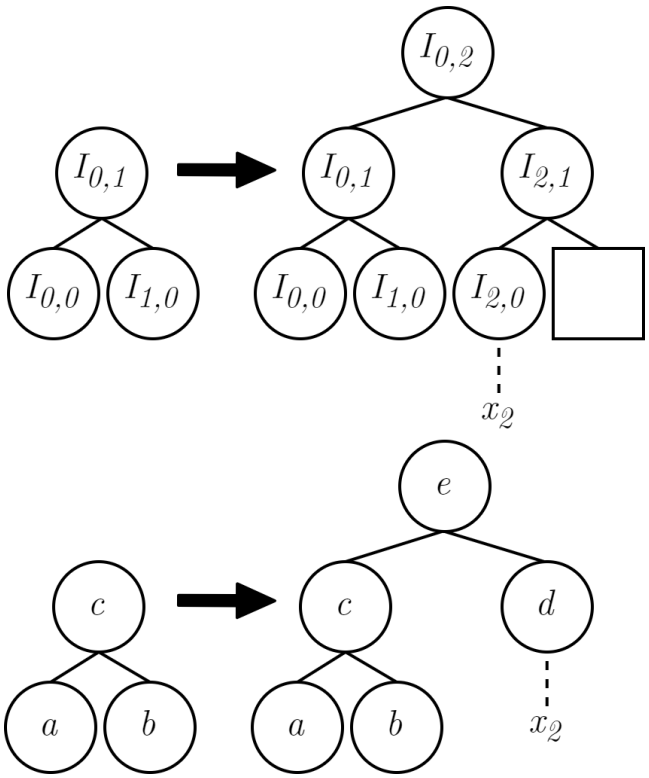

Figure 3: Mutation strategies for MHTs by Crosby (top) and Laurie (bottom). Crosby mutates MHTs by having all new insertions at depth 0 and fills in intermediate nodes, representing empty subtrees with □. Laurie adds leaf nodes parsimoniously, not creating any intermediate nodes but appending leaves as near the root as possible. Thus, the left and right subtrees of a root in Laurie's scheme may have significantly different depths.

of the interior values in I_p nearest the root that also exist in I_q sufficient to calculate I_p, as seen in Figure 4.

2.5 Root Signing

The MHT regime described above assumes a client/server/auditor model in which one or more clients submit their data to a server holding an authoritative copy of a mutable MHT and optionally request proof that their data was successfully inserted. One or more auditors periodically test the consistency of the MHT by demanding proofs.

If a client or auditor (collectively, "agents") receives a proof that does not correctly compute, it can immediately surmise that the server is misbehaving. However, said agent cannot necessarily *prove* this misbehavior to any other agent; depending on what updates and proofs have been seen by another agent, the server could claim that the reporting agent is misbehaving by falsely claiming that the server is misbehaving

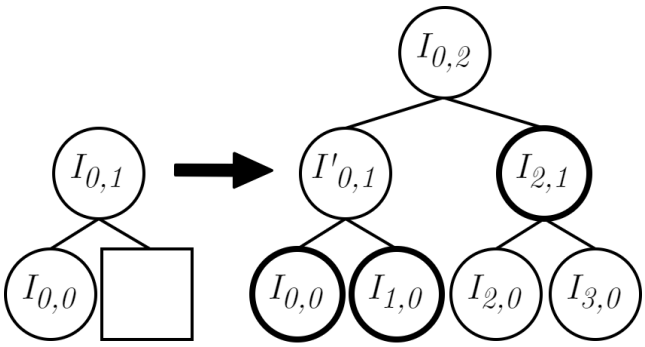

Figure 4: Using Crosby's scheme, the one-element tree at the left can be proved to be consistent with the tree on the right by providing the minimal set of leaves and nodes that proves the left tree can be calculated from the right ($I_{0,0}$ in this case, which can be used to calculate $I_{0,1}$, which is different from $I'_{0,1}$), plus $I_{1,0}$ which can be used to calculate $I'_{0,1}$, and $I_{2,1}$ which, with the computed $I'_{0,1}$, can be used to calculate the root of the right tree. The necessary nodes are in heavy circles.

by offering a forged proof.

To avoid this situation, both Crosby and Wallach and Laurie et al. specify that the server should sign the root values it advertises, and that such signed roots should be published broadly to deter deniability. If root values are signed, any agent can demonstrate that any proof was legitimately offered by the server.

2.6 Security of Mutable Merkle Trees

Given the components above, the resulting mutable MHT has strong security properties:

1. The root digest represents the state of the full MHT, and if any underlying data element is changed, the associated hashes will also change, making MHTs tamper-evident.

2. The append-only structure of the tree means that an attempted deletion of a data element will cause significant and obvious changes to the tree. Thus any consistent MHT guarantees irrepudiability of previously-entered data in a given position.

3. The server and its underlying data do not have to be trusted; each client can verify that her data was inserted and audit that the current state of the MHT still contains that data at any time. Signed roots represent an attestation by the server that can be used to prove misbehavior to other agents.

Proofs and further formal examination of these and other properties can be found in [6, 23, 25, 26].

2.7 Asymptotic Performance of Mutable Merkle Trees

In addition to their security properties, mutable binary MHTs have well-characterized properties with respect to worst-case performance in space and time.

Space Such MHTs grow as $O(n)$. Specifically, an MHT's space consumption for storage of n items is $2n \times 32$ bytes (the number of nodes $\times$ the size of a SHA-256 hash), plus necessary implementation-specific overhead. This does not include the space necessary to store the data elements, which will depend on the size of those data.

Depth The length from a given leaf to the root of an MHT grows as $O(\log_2 n)$.

Insertion Time The time consumed to append an item to an MHT grows as $O(\log_2 n)$.

Lookup Time The time consumed to find some item inserted ith in an MHT grows as $O(\log_2 n)$.

Proof Time The time consumed to generate or verify a proof of inclusion or consistency grows as $O(\log_2 n)$.

Each of these asymptotic limits is for a basic implementation of mutable binary MHTs, and can be improved with additional optimizations, some of which are discussed in Sections 3.2 and 5.

3 immudb

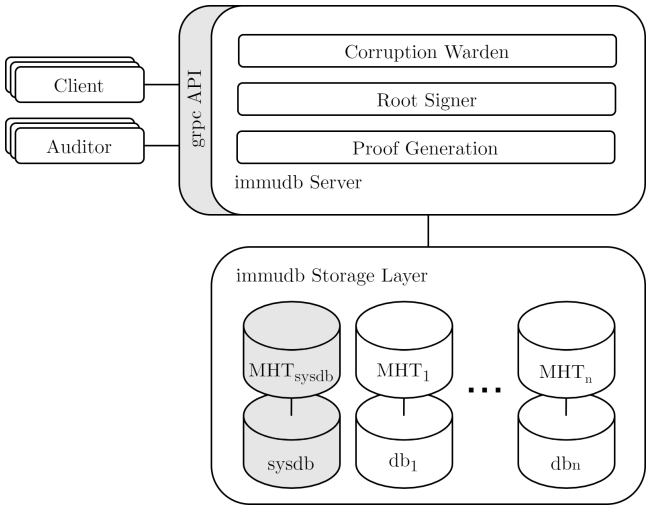

Figure 5: An overview of the immudb architecture.

immudb is an extension of mutable MHTs as described above which presents a general-purpose database backed by a mutable MHT as seen in Figure 5.

3.1 The immudb Server Process

The immudb server process performs the overall management of immudb's MHTs and manipulation of associated data. The server process contains a warden thread that continuously recomputes constituent MHT values from the underlying data to guard against corruption. It also contains the proof generation and root signing mechanisms.

The server exposes an API via grpc [10] to which clients and auditors connect in order to authenticate, insert and fetch data, perform monitoring, and demand inclusion and consistency proofs. The grpc API is optionally secured at the transport layer using mutual TLS [18].

immudb uses FIPS certified SHA-256 [8] and ECDSA [9] for MHT hashing and root signing, respectively.

The server process connects to a storage layer which manages the physical persistence of inserted data and the associated mutable MHTs.

3.2 The immudb Storage Layer

immudb's storage layer includes one or more application databases and the special `sysdb` database and MHT which store the server's own metadata. Each database is comprised of the store, containing the various data elements x_i, and a supporting MHT protecting the store against undetected tampering.

immudb's storage layer is designed as a key-value store, where each key-value pair (k, v) is stored as a data element x_i. The pair, rather than just the value, is hashed in the MHT.

The storage layer maintains indexes to aid in accessing data:

- An insertion order index $i \rightarrow x_i$ is maintained to provide random access by insertion order. This index is also used by the server's proof generation task and corruption warden. This index improves lookup of a data element from $O(\log_2 n)$ for a basic binary MHT to $O(1)$.
- A key index $k \rightarrow \{v_x, v_y, ...\}$is maintained to provide access to the current and historical values of given key k. As immudb is an append-only database, updates to a value associated with a given key are modeled as insertions of new values for that key. Clients may retrieve the most recent value or the list of historical values. This index improves the lookup time of all data elements with the key k from a complete table scan requiring $O(n \log_2 n)$ for the naive case or $O(n)$ by using the index above to $O(|\{v|k \rightarrow v\}|)$, i.e. growing as the number of values v mapped to by key k.
- An MHT node index $i, j \rightarrow I_{i,j}$ providing expedient access to the MHT tree nodes and leaves[3]. This index tightens the constant factor of various $O(\log_2 n)$ tree algorithm runtimes including insertion time and proof generation and verification by eliminating tree traversal operations.

At present the immudb storage layer uses Badger [3] as its storage engine, but the immudb server and storage layer can be considered engine-agnostic. A lighter-weight engine is currently under development which changes none of immudb's properties except to improve performance.

3.3 The immudb API

The immudb API presents various methods for setting and fetching data. Core methods include:

`get(Key k)` Returns the most recent key-value pair (k, v) associated with key k.

`safeGet(Key k, int i)` As above, but also demands inclusion proof of (k, v) and consistency proof between the latest MHT known to the server and the tree as it appeared after the insertion of x_i.

`getByIndex(int i)` Returns the key-value pair (k, v) inserted ith.

`history(Key k)` Returns a list of all key-value pairs (k, v) associated with k.

`set(Key k, Value v)` Adds the key-value pair (k, v) and adds its hash at the next available location in the MHT.

`safeSet(Key k, Value v, int i)` As above, but also demands inclusion proof of (k, v) and consistency proof between the MHT as it stands after the insertion of the requested element and the tree as it appeared after the insertion of x_i.

These API calls provide flexibility when integrating immudb into new or legacy applications. Software that stores key-value pairs works without further effort. Applications making use of time-series data, e.g. tick data, can store these data by "overwriting" the key and then can retrieve the tick-stream by using `history()`. Document-oriented data operations can store documents as the value v in the (k, v) pair, though filtering and manipulation of the documents in the store will have to be done at the application layer at present.

immudb also supports further methods in support of proofs:

`consistencyProof(int i)` Returns the consistency proof between the version of the MHT immediately following the insertion of x_i and the current tree

`inclusionProof(int i)` Returns the inclusion proof for the element x_i.

[3] immudb's MHT implementation generally follows [6], except that non-full trees maintain all their leaves at depth 0. This leads to the nuance that

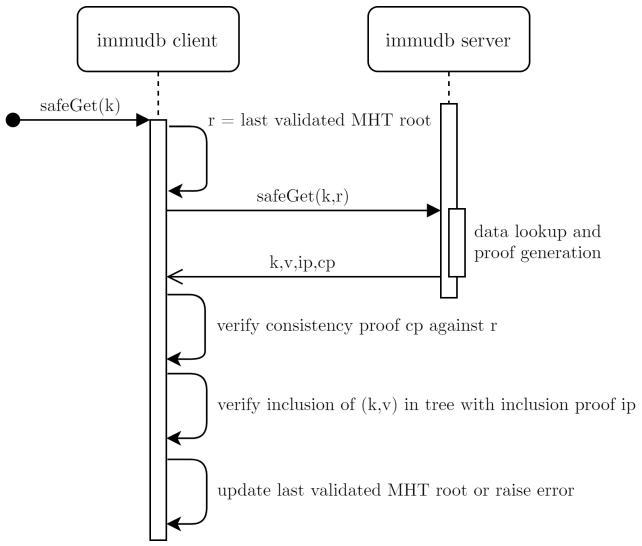

Figure 6: `safeGet()` fetches key-value data together with cryptographic proofs.

Figure 6 provides an illustration of a client's use of `safeGet()`. A client calls `safeGet(k)`, which is syntactic sugar for `safeGet(k,r)` where `r` is the most recently verified root known to the client. The server then returns the key, value, and inclusion and consistency proofs. The client checks the inclusion and consistency proofs and, if the proofs compute correctly, the client updates its last validated MHT root with that from the proofs.

Each auditor and client is responsible for maintaining its own copy of the last valid MHT state it has seen and periodically validating the server's MHT against its own and updating its internal state or announcing a failure accordingly.

Inclusion and consistency proofs have the desirable property that said proofs do not include data elements from the database, but only their hashes. As such, although multiple users may use the same database, under an appropriate access control regime, no user will be sent data to which she does not have access.

3.4 immudb Defense in Depth

As with any service, security best practices are necessary in order to ensure safe operation. While the mutable MHTs described in this paper are robust against silent tampering, they provide no guarantees beyond standard databases with respect to data destruction, corruption, or pilfering that is not meant to be covert.

Appropriate network segmentation and access control should be practiced. Further, it is strongly recommended that the immudb server run on a separate machine (physical or virtual) from any clients to ensure that the only interaction the clients have with the server or its data is through the exposed grpc API.

Root signing provides stronger guarantees of irrepudiability, but are only useful in applications in which agents may be assumed to work adversarially. As ECDSA signing is computationally expensive, using immudb without signed roots may offer greater throughput in cases where clients who detect a fault in the database may be taken at their word.

Section 5 provides a look ahead at mechanisms under development that may further harden immudb and applications that interface with it against attack.

4 Applied immudb

The example in Section 1 may appear trivial and the result of poor security on the part of small media operators. However, the threat of data manipulation is in the wild, often as part of long-term breaches resulting in significant losses.

FireEye reported in October 2018 on a campaign that uses data manipulation as part of an attack's critical path. Since at least 2014, the North Korean state sponsored APT38 threat group [2] has, based on public reporting, stolen some 1.1 billion USD from various banks using the SWIFT global financial transaction network [19].

In a well-orchestrated series of attacks, APT38 used a combination of spearfishing and other social engineering attacks, malware, and abuse of various security vulnerabilities to enter the networks of financial institutions. Once inside, APT38 would spend significant amounts of time (155 days on average, up to 678 days, per FireEye) gathering intelligence about the network and computing environment, and then insert large transactions into the SWIFT network bound for accounts under APT38's control at banks in other nations with poor fiscal oversight.

As part of a robust strategy for evading detection until funds had been delivered, the DYEPACK malware toolkit that APT38 deployed would directly execute SQL commands on the Oracle database backing the SWIFT transaction server and delete or modify fraudulent transactions made, as verified by forensic analysis of live copies of the malware [19, 21, 22].

This simple step prevented the examination of outgoing SWIFT transactions that would have immediately flagged the activity for review.

In such a system, immudb could have been deployed in one of two ways. First, an immudb client listening to the Oracle database's Change Data Capture (CDC) facility [1] to record all database activity could have piped the CDC output to an immudb server, itself secured against corruption by multiple auditors distributed throughout the network. The immudb database would then have an irrepudiable and tamper-evident record of DYEPACK's updates and deletes, and action could be taken. This solution, however, while necessary in

leaves can have parents at depths > 1, but this is abstracted from the end user and does not change any security properties.

many scenarios with legacy applications, is far from ideal. In the specific case of DYEPACK's attack against SWIFT servers, DYEPACK could simply have dropped the relevant CDC tables from the Oracle database, disabling CDC entirely, and then re-created them when the deletions and edits were complete.

The more secure if higher-effort mitigation would be to replace the Oracle instance with immudb. immudb provides no APIs for modification and deletion, so deletion of records from a command-line interface as was done with Oracle is not possible. Modification of data directly on disk is possible, but such modification would be quickly detected by the continuously-running server corruption warden, as well as by deployed auditors.

Work described in the following section will serve both to harden immudb deployments in such situations and reduce the burden of application development.

5 Future Work

immudb today represents a significant leap forward in irrepudiable, tamper-evident, auditable data storage. Additional features that are planned or in development to make immudb both more usable, useful, and performant include, but are not limited to:

Drivers - Drivers written for the most-used languages to allow the native use of immudb with more new and legacy applications.

SQL-Like Querying - Support for a SQL-like query language to allow more sophisticated server-side fetching of results and an easier transition to immudb from less secure legacy datastores.

Improved Storage Engine - Creation of a engine specifically designed to efficiently support the data structures underlying immudb.

Caching - Support for configurable caching based on application needs. As past entries and hashes within frozen subtrees are functionally immutable in immudb, these lend themselves to mature techniques for replication and caching of static data.

High Availability and Sharding - Support for configurable replication and sharding in support of load balancing, fault tolerance, and availability. The uniformly random output of the SHA-256 hashes used by immudb in particular make auto-sharding based on hash prefix ranges straightforward and probabilistically load-balanced.

External Security Keys - Support for hardware devices containing cryptographic keys, e.g. Universal 2nd Factor (U2F) [15] devices from Nitrokey [11] or Yubico [20].

Encryption at Rest - Following on from the immudb-specific storage engine, support for data encryption at rest at the database server level to complement immudb's mutual TLS encryption in transit.

Gossip Protocol - Support for a gossip protocol among auditors and clients to transparently and continuously detect and flag suspected tampering.

GPU Acceleration - Support for acceleration of immudb's SHA-256 hashing and ECDSA signing using commodity GPU hardware. Acceleration of cryptographic primitives makes use of the wide data bus of GPUs compared to CPUs and can, in cases of transactions that trigger large numbers of simultaneous hash calculations or signatures, provide significant improvements in throughput.

GDPR and CCPA Compliance - Compliance with the European Union's General Data Protection Regulation (GDPR) [7] and the California Consumer Privacy Act [4]. The GDPR mandates certain circumstances in which data **must** be deleted. While this might appear challenging for append-only databases such as immudb, because the coherence of the underlying MHT depends upon the hash of a given value rather than the value itself, the underlying data may itself be deleted under a regime that appropriately audits the data and records (e.g. in a companion immudb metadatabase not subject to deletion or redaction regulations) which entries have been removed under what authority.

6 Conclusion

In this paper we have described the design and implementation of immudb, a performant append-only general-purpose database that provides strong tamper-evidence and irrepudiability of inserted data.

Starting with Merkle Hash Trees and subsequent work by Crosby and Wallach as well as the Certificate Transparency standard proposal by Laurie et al., we expand upon the narrowly defined prior art by providing a practical implementation with the novel extension to a general-purpose database usage metaphor.

Correctly deployed in conjunction with standard network and application security best practices, immudb provides defense against undetected insertion, mutation, or deletion of sensitive data. The cryptographic digest of the database state allows regular and continuous auditing without the computational expense of continuous scans of the state of the database, and optional server-side signing of these digests allow auditors to demonstrate authoritatively when the server's data has been tampered with.

The properties of immudb make it appropriate for use in a variety of industry siloes in which the correctness of data and resistance against undetected *post hoc* modifications is crucial. The gamut of use cases is extensive, including financial ledger compliance, electronic health records, passport and document control systems, scientific data records, source code validation, and so forth. We believe that the broad-scale application of immudb has the potential to shift the entire security landscape.

7 Acknowledgments

The authors gratefully acknowledge Ralph C. Merkle, Scott A. Crosby, and Dan S. Wallach for their foundational contributions to the data structures and algorithms that underpin immudb.

The immudb Project

immudb is open source software under an Apache 2.0 license, and can be found at:

https://github.com/codenotary/immudb

More information about immudb is available at:

https://immudb.io

References

[1] 16 Change Data Capture. https://docs.oracle.com/cd/B28359_01/server.111/b28313/cdc.htm.

[2] APT38. https://attack.mitre.org/groups/G0082/.

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

[4] Code Display Text. http://leginfo.legislature.ca.gov/faces/codes_displayText.xhtml?division=3.&part=4.&lawCode=CIV&title=1.81.5.

[5] Combined Regulation Text of All Rules. https://www.hhs.gov/hipaa/for-professionals/privacy/laws-regulations/combined-regulation-text/index.html.

[6] draft-ietf-trans-rfc6962-bis-34 - Certificate Transparency Version 2.0. https://datatracker.ietf.org/doc/draft-ietf-trans-rfc6962-bis/34/.

[7] EUR-Lex - 32016R0679 - EN - EUR-Lex. https://eur-lex.europa.eu/eli/reg/2016/679/oj.

[8] FIPS PUB 180-4 Secure Hash Standard. https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.180-4.pdf.

[9] FIPS PUB 186-4 Digital Signature Standard (DSS). https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.186-4.pdf.

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

[11] Nitrokey | Secure your digital life. https://www.nitrokey.com/.

[12] Part 99–FAMILY EDUCATIONAL RIGHTS AND PRIVACY. https://www.ecfr.gov/cgi-bin/text-idx?node=34:1.1.1.1.33.

[13] SARBANES-OXLEY ACT OF 2002. https://www.govinfo.gov/content/pkg/COMPS-1883/pdf/COMPS-1883.pdf.

[14] Senate Bill S5575B. https://www.nysenate.gov/legislation/bills/2019/s5575.

[15] Specifications Overview - FIDO Alliance. https://fidoalliance.org/specifications/.

[16] SUBCHAPTER F–FAIR CREDIT REPORTING ACT. https://www.ecfr.gov/cgi-bin/text-idx?tpl=/ecfrbrowse/Title16/16CIsubchapF.tpl.

[17] SUBCHAPTER III–CREDIT REPORTING AGENCIES. https://uscode.house.gov/view.xhtml?req=granuleid%3AUSC-prelim-title15-chapter41-subchapter3&edition=prelim.

[18] The TLS Protocol Version 1.0. https://datatracker.ietf.org/doc/rfc2246/.

[19] Un-usual Suspects. https://content.fireeye.com/apt/rpt-apt38.

[20] Yubico | YubiKey Strong Two Factor Authentication. https://www.yubico.com/.

[21] 649. APT38 DYEPACK FRAMEWORK. https://github.com/649/APT38-DYEPACK.

[22] AKM. APT38 DYEPACK Framework used to Steal over $1.1 BILLION from SWIFT Banking Servers. https://spuz.me/blog/zine/4P738DY3P4CK.html.

[23] BECKER, G., AND UNIVERSITÄT BOCHUM, R. Merkle signature schemes, merkle trees and their cryptanalysis.

[24] BULDAS, A., LAUD, P., LIPMAA, H., AND WILLEMSON, J. Time-stamping with binary linking schemes. In *Advances in Cryptology - CRYPTO '98, 18th Annual International Cryptology Conference, Santa Barbara, California, USA, August 23-27, 1998, Proceedings* (1998), vol. 1462 of *Lecture Notes in Computer Science*, Springer, pp. 486–501.

[25] CROSBY, S. A., AND WALLACH, D. S. Efficient data structures for tamper-evident logging. In *Proceedings of the 18th Conference on USENIX Security Symposium* (USA, 2009), SSYM'09, USENIX Association, p. 317–334.

[26] DOWLING, B., GÜNTHER, F., HERATH, U., AND STEBILA, D. Secure logging schemes and certificate transparency. In *Computer Security – ESORICS 2016* (Cham, 2016), I. Askoxylakis, S. Ioannidis, S. Katsikas, and C. Meadows, Eds., Springer International Publishing, pp. 140–158.

[27] FOSTER, L., RIDDELL, S., MAINOR, D., AND RONCONE, G. Ghostwriter Influence Campaign. https://www.fireeye.com/content/dam/fireeye-www/blog/pdfs/Ghostwriter-Influence-Campaign.pdf.

[28] MERKLE, R. C. *Security, Authentication, and Public Key Systems*. PhD thesis, Stanford University, 1979.

[29] 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.

[30] PAVLOU, K. E., AND SNODGRASS, R. T. Forensic analysis of database tampering. *ACM Trans. Database Syst. 33*, 4 (Dec. 2008).

[31] SNODGRASS, R. T., YAO, S. S., AND COLLBERG, C. Tamper detection in audit logs. In *Proceedings of the Thirtieth International Conference on Very Large Data Bases - Volume 30* (2004), VLDB '04, VLDB Endowment, p. 504–515.