Literature Review

Heyi Wang, Rongbin Cui, Yuhao Wu

Introduction

This paper shows making data integrity a commodity by using FastVer.

Data Integrity

Data Integrity refers to the accuracy and reliability of data. It is proposed to prevent the existence of data that does not conform to the semantics in the database and to prevent invalid operations or wrong information caused by the input and output of wrong information. Data integrity is divided into four categories: Entity Integrity, Domain Integrity, Referential Integrity, and User-defined Integrity.

Existing Technical Background

A database system would maintain a hash table containing usernames and user passwords. Suppose the system administrator can tamper with the login information and table information at will without being discovered. In that case, it is possible to pretend to be another user to log in to the service. To implement data integrity, we can store an encrypted hash of data in a secure and trusted location. And we can use Merkle trees to manage the hashes.

Merkle tree

In the Hash Tree (Merkle Tree), each node is marked with a data block encrypted hash. The hash tree can be used to verify any data stored, handled, and transmitted between the computer and the computer. They ensure that the speed of data transmission in the point-to-point network is not affected, the data spans the unrestricted use of any medium, and there is no change in damage and has not changed.

Deferred memory verification

Use the method of delayed verification to verify the integrity of the operation in batches.

Limitations of Merkle tree and deferred memory verification

1. Merkle tree only has the performance of tens of thousands of times per second in the Enclave.

2. Deferred memory verification scans the entire database in trusted locations, and verifying operations in batches may take tens of seconds to complete, resulting in too high a time and performance cost.

A Combination of two simple but effective ideas

1. Intra-enclave caching: Validate data by leveraging the cache within the Enclave. It not only stores the root of the Merkle tree but also caches any nodes in the Enclave. The benefit of introducing a cache is that only the path from one record to the first cached ancestor needs to be validated, reducing costs. In addition, concurrency is improved because each operation no longer starts from the root node, but the root is removed as an individual contention through the cache.

2. Hybridizing Merkle and deferred memory verification: The core idea is to perform integrity checks on database records and Merkle tree nodes. This scheme handles cold data at a lower cost because of the flexibility to defer verifying the integrity of the scan.

FastVer

By integrating these data integrity technologies into a system named FastVer. FastVer can handle tens of millions of data and operate key-values per second.

After the author's test, the performance of FastVer outperforms Faster than Redis, and FastVer outperforms Merkle trees by two times.

Problem Statement

This section describes the architecture of the algorithm, some problems, and the threat mod.

Architecture:

get(k) //To determine whether the value of the key matches, if it matches, it returns the current value, otherwise it returns null

put(k, v) //To update the database and log them to another external storage. Thus making the record sustainable and permanent

if we add the enclave, which means the host database system transfers the result to the verifier first, then it returns the result of a client request, If it is validity then digitally signs the result and returns successfully so that the client only receives result which has the signature of validation.

Then extend the get/put request like:

get(k, t)→<v,$s_v(k, v, t)$>
put(k, v, t, $s_c(k, v, t)$)→<$s_v(t)$>

Add a parameter t in get request so that the validation signature covers t to prevent returning a previously verified output from an untrusted host.We can see that put request has other two parameters t and $s_c(k, v, t)$. $s_c(k, v, t)$ denotes a client signature, which used to protect other parameters in put request. And parameter t in put request is used to avoid the rollback attacks.

Threat Model and Integrity Guarantees

The author builds a threat model with some assumptions and Integrity Guarantees.

Assumption 1: The attacker runs the server in Byzantine form, so the attacker can take control of the host database system in a way that deviates from the protocol.

Assumption 2: An attacker cannot break the cryptographic difficulty assumptions required to implement primitives.

Assumption 3: The Enclave is protected by an attacker, which means the attacker cannot change the calculations and the status of the host database system.

Assumption 4: The validator API can be used by an attacker.

Assumption 5: An attacker can restart the Enclave and reset the validator to its original state.

Assumption 6: To prevent rollback attacks, the verifier maintains a small number of persistent states to hold individual hashes.

Assumption 7: The client is trusted and inaccessible to the adversary.

Assumption 8: The communication between the client and the validator takes place through an untrusted server. In other words, the communications between the two parties are under the attacker's control.

Assumption 9: The system does not guarantee schedule or availability.

Integrity Guarantee 1: The clients and verifiers can share cryptographic secrets by using public key infrastructure to establish symmetric keys.

Integrity Guarantee 2: The output seen by the client reflects some order of historical updates.

Reference author describes the verified code from TCB.

Performance Goals

The approaches should satisfy the following performance goal:

P1: Size of the Verifier State: A solution should not store the whole database to get good performance because of the limitation of storage.

P2: Verification Complexity: The solution should decrease the additional computation as lower as possible which means we need an high throughput method to let the verification complexity be O(1).

P3: Verification Latency: The database should not be used as a factor to control client setup latency.

P4: Concurrency Bottlenecks: The concurrency bottleneck of verified databases should not beyond the load of the customer because we should consider the different performances of different client.

Simple Approach

Run the entire system in one Enclave. Untrusted hosts only relay requests and responses from clients to the enclave database system. This approach is reliable and can achieve good performance when the entire database can fit in the enclave memory.

Problem of A1: This solution can not meet P1 because it runs the entire system in one Enclave.

Update: Adding "page out" validator records to untrusted storage outside the Enclave instead of storing the entire database in one Enclave. If some of them need verification, "page out" them back.

Problem of Update: Untrusted hosts can tamper with the records being paged out.

Solution: Add a data integrity check that verifiers can use to ensure that when called in, what is logged is the same as when it was last called out by identifying the keys so that we can just check the key-values.

Merkle Tree Approach

A Merkle tree (also called a hash tree) is a typical binary tree structure consisting of a root node, a set of intermediate nodes, and a set of leaf nodes. First proposed by Merkle Ralf in 1980, Merkle trees have been widely used in file systems and P2P systems.

Its main features are that the bottommost leaf node contains the stored data or its hash value. Non-leaf nodes (including intermediate and root nodes) are hashes of the contents of their two child nodes. Further, the Merkle tree can be extended to the case of a multinomial tree when the content of a non-leaf node is a hash of the content of all its children nodes. The characteristic of the Merkle tree to record hash values layer by layer gives it some unique properties. For example, any changes in the underlying data are passed to its parent nodes, layer by layer, along the path to the tree's root. This means that the value at the root of the tree represents a "digital summary" of all the data at the bottom. However, the traditional Merkle tree doesn't meet the verification complexity requirement(P2) because every operation incurs a logarithmic validation cost. Also, the Merkle tree's feature that touches the root when each operation is made doesn't meet the Concurrency Bottlenecks(P4) requirement. And since the traditional Merkle Tree can't the null for a get request, they used a variant of the Traditional Merkle Tree called Sparse Merkle tree. And it solves this problem by using a unique value NULL for non-existent keys. Based on this data structure, they also tried to make two big enhancements to improve the performance.

The first enhancement is to encode the Merkle tree with two kinds of records. The record is responsible for external purposes such as being created by the users and being the reference for the operations. And the other record is called Merkle records, designed for verification purposes. The second enhancement is adding verifier caching feature. How do they achieve this enhancement by containing the Merkle records into verifier caching, which makes any records that could be added and removed from the cache. One feature of this enhancement is called lazy update. The propagation to ancestors won't happen during the regular record update unless the parent record is evicted.

Those enhancements improve the performance of the Merkle tree from 10k ops/sec to around 100K ops/sec, which is a huge alleviation. However, it's still the Merkle tree; it still doesn't completely meet the performance goal of P2 and P4.

The Deferred Memory Verification Approach

Deferred memory verification relies on collision-resistant multiset hashing, which could hash two multisets with properties to two different hash values. By making this approach, we don't need to use any data structure. There is a read-set and a write-set. Read-set catches the pre-image of the records for the operations, and the write-set catches the related operation's reflection. And after adding these records, read-set and write-set will check if they are equal. Otherwise, they will fail the check. Based on this mechanism they come up with an enhancement with caching. The verifier in the root will add the newly added record cache to its read-set. When the record is being evicted, it will be added to its write-set, and it will verify those to check if the check the content after the record is added to the catch is identical to when it was removed.

Deferred memory verification meets the P1, P2, and P4 performance goals. It is very efficient with high concurrency. However, due to the thread during the verification scan caused by all the records being routed to

the verifier bringing linear verification latency, it doesn't meet the performance goal of P3.

Hybrid Approach

Hybrid approach mentions it combine the advantages of Merkle trees and Deferred memory verification to take the tradeoff of throughput and latency. In this approach need ensure any three options to integrity of record. First option, the record needs be kept and protected in a verifier cache, the second and the third options are storing the record in untrusted storage. And either of them protects records that storing the value in a cryptographic hash or storing its value in write-set hash.

In hybrid approach, it has verification hierarchy, caching is at the top that store hot records, and the deferred memory verification is follow by cache, but it has limitation of the numbers of records in order to control the verification latency. If record in the memory is full, Merkle hashing would store the rest records since it doesn't have this limitation, it's similar to secondary storage. If the cache is full, storing records in deferred memory priorly since the computation in deferred verification is less expensive than in Merkle hashing. The verification can access record in aggregate while the Merkle hashing access it individually. The other function in hybrid approach is parallelizing Merkle-Hashing, cache in the verifier thread has the root record and it only adds the records into verifier thread without adding its parents. Hence, the merkle hash computations would be process in verifier thread. Using deferred verification to protect Merkle records unshackled from their parent records and the records can be added to the cache of any verifier thread. It also uses F* proof to proven the approach correct.

Implementation

FASTVER is an application based unmodified FASTER that has an api which can customize the specified logic for key-value operations by ourselves, and the project uses C++ language to implement it. In the implementation, every value has a 64-bit aux filed used in the internal bookkeeping and managing the database. In thread model, every worker ensures the operation of records are in verifier cache, it would get result from verifier and return it to client. In addition, every worker has its own log stream that means it has no log contention in each thread, once the buffer in verification log is full, the corresponding thread would access in the enclave and addresses the entries that update, delete or something. In the thread inner loop, the value in aux filed is related to encode the latest protection mechanism in verification, it's related to timestamp in deferred as well as related to thread and slot in cache. The special thing in timestamp is that the value can be computed exactly in prediction without actual operation, in verification due to its clock. Even though it would have subtle different in actual step, the overall structure is the same that updating the atomic which is contended and recording uncontended data in the log at background, those atomic is related to the value in aux fields. Cryptography is used C language of a specified techniques called Blake3 for Merkel multiset hashing and the construction was built by using Intel AES-NI instructions to execute the vectorized assembly of AES, besides durability use standard CPR logging techniques to align with epoch-based verification.

Experimental Evaluation

The system environment of simulated enclaves was used Ubuntu 18.04 without Intel SGX enclaves, another one of true enclaves has Intel SGX. In the experiments, simulated enclaves use 32 workers and true enclaves use 8 worker and a cache has 512 entries for each verifier thread. In comparison, there have four variants of

the YSCB, the first workload is 50% read and write respectively, the second one is 95% read, the third one is read-only and the last one contains scanning function. In most of the experiment, read-only is the benchmark, and the theta of Zipfian distribution is equal to 0.9 in default.

There're four experiments, first one is about performance of throughput and latency, trading off throughput and latency to present a well performance according to setting two main parameter that batching and the count of merkle records. Comparing the performance of different size of records and comparing the performances in different workloads. The results are that the curve of throughput-latency is flatter at low latencies that means merkle records are protected by using deferred verification in this region, the other thing is that scan-based workload would make the cached merkle record higher than those without scanning. The second type of experiment is SGX Enclaves that using 50%read and write respectively workload, comparing true enclaves and simulate enclaves, the observation is that the performance in true enclaves is slight slowdown than simulation that the reason is no extra model in the simulated enclaves likes a model of increasing cost of memory accesses. The third type is comparing the FASTER and FASTVER in two workloads, 50%read and only-read. In FASTVER, it has two types to compare, one is no constraints on verification latency and the other has sub-second verification latency. The performances show that FASTVER still has competitiveness since the proportion of FASTVER performance divided by FASTER performance is more than 50% in all database sizes. Even though, the performance of sub-second verification latencies is several times slower than FASTER, this program still tolerate verification latencies in a second. The fourth one is about scalability that varying the number of worker threads in different sizes of databased with 50% read and shows that FASTVER scales well with the number of worker threads that 32 workers can perform slightly more than double of those in 16 workers.

The last experiment is about the performance of Drill-Down. In these experiments, it uses 64M size of dataset and perform throughput performance of 5 different verification techniques that has different size of cache, using Merkle tree without verifier caching as baseline to compare these five verifiers and different size of cache presents different degree of throughput. With non-sequential Merkle variant in single thread, caching can improve the verification cost that show in the experiments and still slight improve its performances. In sequential workload, caching has a significant improvement of its performance according to observing its throughput. In multithreads, it displays the throughput in different number of threads and observing whether it affect the memory access cost.

Conclusions

The design of this paper presents the state-of-the-art integrity solution to achieve appreciable throughput when it optimizes FASTER key-values store and it's a milestone in data integrity solutions.

Division of labour

Name	Report for Paper
Heyi Wang	Responsible for Paper (6-9) reporting hybrid approach, implementation, evaluation and conclusion.
Rongbin Cui	Responsible for Paper (4-5) reporting two methods, Merkle Tress and enhancements
Yuhao Wu	Responsible for Paper (1-3) reporting introduction, background and trust database approach