Setsum: A Checksum for Set Membership

Robert Escriva
Seeking Employment

Abstract

This paper introduces the concept of a *setsum* for set membership. A setsum is a checksum that operates on sets of byte strings with the property that when two sets contain the same strings, they have the same setsum regardless of the order in which strings are considered. This paper walks through the setsum construction and gives an end-to-end example of a database protected by setsums.

1 Introduction

Checksums are probably the most ubiquitous method of data integrity assurance. From network packets to database pages, checksums generally take in a byte stream of data and produce a fixed-size output with the crucial property that when the byte stream changes, the checksum changes with high probability. Some checksums are cryptographic in nature because they make it a near-certainty that small changes in the input—or any changes in the input—lead to big changes in the output with low risk of collision.

It is common to build tooling on top of the raw checksums that verifies the checksummed data. For example, it is common to publish cryptographic checksums of software distributions alongside the distributions themselves. Any user can verify that the checksum holds by computing the checksum and comparing it to the published values. Another example would be Percona toolkit's table checksum (pt-tc) tool. The pt-tc tool passes hashes of data through the replication stream allowing the main replica to dictate what the data should be and the replicas to compare the data to the replicated checksum. Fundamentally, both tools established an agreed-upon convention that both the party creating the checksum and the party verifying the checksum, checksum the data in a particular order. Specifying the same bytes in a different order, or specifying a different set of bytes changes the checksum.

This paper introduces the idea of a *setsum*, which is a fundamentally new type checksum that operates on a set of byte streams. The setsum provides operators for computing set union and set difference, which allows for efficient addition and removal of items from a setsum to correspond to changes in a set's membership. The core idea at play is to maintain membership of the setsum in parallel to maintenance of some other set of data. When an item is added to the set, it gets added to the setsum. The setsum for the union of two sets is the union of their setsums. At all times the setsum corresponds to the current membership of the set, matching the value that would be computed if the items were to be hashed from scratch. Consequently, it is possible to use setsum to compare the maintained hash against the data to verify that the data matches what's expected.

For a concrete example, we could construct a database that maintains for each transaction the setsum corresponding to the database when the transaction committed. The set union operator used to merge in the transaction's delta takes $O(1)$ time, so each transaction can efficiently compute its local checksum and then compute the overall database's checksum with a single merge. The database could then verify that backups match their respective checksums as a way of verifying backup integrity without doing more than iterating over the backup to compute a setsum.

The core contributions of this paper are:

- We provide a specification of the setsum and contrast it to existing checksum APIs.
- We evaluate a Rust implementation of the setsum and discuss performance considerations.
- We describe three example use cases within a database and show how a setsum can be extended to cover all aspects of database integrity.

The rest of this paper is as follows. Section 2 gives an overview of the math behind our implementation and goes over some related work. Section 3 goes over the design and implementation considerations. Section 4 describes how to use setsum to provide end-to-end data integrity protection for a database. Section 5 evaluates our implementation, and Section 6 concludes.

2 Background

Conceptually, a setsum builds on checksums. Where checksums operate on byte streams, setsums operate on sets of byte streams. To relate the two: In a checksum, there is just one element that gets hashed while a setsum hashes the members of a set in a way that preserves set structure. Setsums support set union and set difference so that two setsums representing two sets can be union'd or diff'd in setsum-space with $O(1)$ operations. Figure 1 gives a concise illustration of the difference between a checksum and a setsum in pseudo-Rust.

The way any setsum works starts with a mapping from the input space into setsum space. This mapping gives the setsum for the one-element set associated with each input. Setsum places no requirements on this mapping, beyond asking that it provide a strong enough hash to make the risk of collision negligible. A hash like SHA-256 would work well.

Once an element is hashed to its one-element setsum, an associative and commutative merge function we'll call $\cdot$ directly provides the associative and commutative properties of the setsum. The choice of $\cdot$ affects the setsum greatly, but without knowing its definition we can say that if $\cdot$ provides set-union, then $\cdot$ of the inverse will provide set-difference.

A setsum, then, is defined by the hash function of its inputs that relate byte streams to element's hashes, the merge operator $\cdot$ that computes set union over the elements, and an inverse function that is defined for all elements.

One straw-man way to achieve this would be to use a checksum like SHA-256 to get 256-bit checksums that XOR together. The inverse function for XOR just so happens to be the identity function so we have a trivial setsum with XOR.

For example, let's consider the set of elements A, B, C. Let's say these elements hash to the values $h(A) = 0b1101$, $h(B) = 0b0110$ and $h(C) = 0b1100$. The setsum of these three elements is $0b0111$, the XOR across all the values. Adding an element a second time is the same as removing it. We can subtract B from the set by adding $inverse(B)$, which is the same as adding B again to get $0b0001$.

The optical illusion that makes addition look like deletion is a massive problem here: One of the problems checksums detect is corrupt data and when duplicate data looks like missing data—as it must with XOR—we cannot reliably determine what went wrong. Ideally we would have to add an element many times over before it begins to act as its own inverse. We can think of this as saying something about the strength of the setsum. Our goal, then, is to develop the necessary tools to make a strong setsum.

The necessary observation to make here is that even though there are 2^{256} different hashes, it only takes a cycle of length two to encounter a problem because that is the periodicity of an individual bit. A 256-bit number is 256 groups of size two. The periodicity is two because all the groups cycle together regardless of initial configuration. In general it is the least-common multiple of the groups' periodicities.

```
fn checksum(bytes: String)
    -> ChecksumDigest;

fn setsum(set_of_bytes: Iterator<Item=String>)
    -> SetsumDigest;
fn union(lhs: SetsumDigest, rhs: SetsumDigest)
    -> SetsumDigest;
fn diff(lhs: SetsumDigest, rhs: SetsumDigest)
    -> SetsumDigest;
```

Figure 1: A checksum takes a string and returns a digest. A setsum takes an iterator over strings and returns a digest. The iterator is free to return strings in any order; for a given set of strings the setsum is order-invariant.

We can make the periodicity large by making a single, large group. Our next straw-man way to do this would be to use large numbers—on the order of as many bits as there are in the hash of an element—and do modular arithmetic over these large numbers. This requires integer division over numbers with many hundreds of bits. It's infeasible fast, even for small objects, but it would give us the strongest of checksums if we chose our modulus right.

For our straw men, the fast group is not strong and the strong group is not fast. What we want is some group over which we can perform operations with a constant-time, fast merge operator that's free from weakness. The two weaknesses in the checksum are the mapping from elements to their one-element-set setsums and the merge operator. We correct the first weakness by using a cryptographic checksum to generate the bytes used as a one-element-set setsum. We correct the second weakness by using some group theory.

We need the $\cdot$ operator to be associative and commutative; that is, we need an Abelian group [6]. This means that for any two elements e_i, e_j, the result of $e_i \cdot e_j$ will be in the group too, and we can compute a set in any arbitrary order so long as we eventually add all elements.

The critical property we ask of our Abelian group is that there should be no subgroups that capture the elements in a cycle. For example, when working with numbers mod 12, the values 4 and 8 capture the value. There's no way to transition to something other than a 0, 4, or 8 once you land on either. Similarly 3, 6, and 9 capture the value. And once on zero there's no transitioning away from zero.

To ensure there are no bad subgroups, it is sufficient to ensure that for every element, every other element is exactly one value's merge away. This stems from the definition of a sub group in which the $\cdot$ of any two sub group elements will also be in the sub group. If it is possible to reach all possible values the bad subgroup will not exist. Constructively for an element e_i to reach e_j it suffices to take $e_i^{-1} \cdot e_j$ to transition from e_i to e_j because $e_i \cdot e_i^{-1}$ must necessarily give the identity element, and cannot change e_j. Thus we can guarantee there will be no bad subgroups by ensuring that every element has an inverse.

3 Design and Implementation

The design and implementation of setsum centers around the Abelian group used for merge operations and the inverse operator used for difference operations. In this section we pick a group that is as strong as those discussed in the previous sections and is possible to implement efficiently.

3.1 Picking a Group

Our implementation leverages an insight we can make about XOR to provide a strong setsum. A 256-bit number can be cast as 256 Abelian groups consisting of the numbers mod 2. Our previous insight was that the least-common multiple of the groups' sizes will determine the strength of the setsum. What if we instead of 256, 1-bit groups we used fewer, bigger groups? And, rather than mod 2, let's have the modulus of each group be different and co-prime to every other modulus so that the least-common multiple is $\prod_i p_i$ for the broader setsum. The resulting group would be uniquely defined by the number and value of primes, p_i, specified.

It seems natural, then, to move from 1-bit modular arithmetic to some other number of bits. There's a tension here: Too big, like the largest of 64-bit numbers, and it becomes hard to do modular arithmetic without first overflowing the type—something that would lead to incorrect values. Too small and there's too many modular arithmetic computations.

Our approach is to do 32-bit arithmetic with 64-bit numbers. In this way values can temporarily overflow 32-bits and addition modulo a prime number is sufficiently efficient. To make the groups' sizes co-prime, we do arithmetic modulo the largest 32-bit primes. It's easy to see why this group works and modular arithmetic is well supported out of the box in every language. Figure 2 shows the exact arithmetic.

Formally this construction provides an Abelian group over the elements 0 to $\prod_i p_i$ where p_i is the i'th largest 32-bit prime. We can see that the, "No bad subgroups," assumption holds because we can construct an inverse element for every valid setsum using subtraction from the provided primes, and thus we can construct the transformation that takes e_i to e_j by $e_i^{-1} \cdot e_j$ for all e_i, e_j.

3.2 Syntactic Sugar

In our implementation we augment the code from Figure 2 with syntactic sugar that allows for more normalized usage of the library. First, we create an accumulating `Setsum` type. Then, we implement insert and remove operators on the type to allow single element insertion and deletion. Addition and subtraction operators allow for efficient merge and different operations. There are no batched APIs because they are akin to a parallel computations brought together via a merge and should be implemented as such. Figure 3 shows prototypes of the syntactic sugar.

```
pub const SETSUM_COLUMNS: usize = 8; (1)

type Setsum = [u32; SETSUM_COLUMNS];

const PRIMES: Setsum = [ (2)
    4294967291, 4294967279,
    4294967231, 4294967197,
    4294967189, 4294967161,
    4294967143, 4294967111,
];

fn merge(lhs, rhs) -> Setsum
{
    let mut ret = Setsum::default();
    for i in 0..SETSUM_COLUMNS {
        let lc = lhs[i] as u64; (3)
        let rc = rhs[i] as u64;
        let sum = (lc + rc)
            % PRIMES[i] as u64;
        ret[i] = sum as u32;
    }
    ret
}

fn inverse(state) -> Setsum {
    for i in 0..SETSUM_COLUMNS {
        state[i] = PRIMES[i] - state[i]; (4)
    }
    state
}
```

Figure 2: The core constants and algorithm. (1) The groups consist of eight 32-bit numbers, for approximately 2^{256} strength. (2) The primes chosen are part of the specification. (3) Math is performed in 64-bit space to avoid 32-bit overflows when adding two numbers. (4) Inverse is the value that will take us to zero. It's subtraction with no need for division.

```
pub struct Setsum {
    state: [u32; SETSUM_COLUMNS],
}

impl Setsum {
    fn insert(&mut self, item: &[u8]);
    fn insert_vectored(&mut self, item: &[&[u8]]);
    fn remove(&mut self, item: &[u8]);
    fn remove_vectored(&mut self, item: &[&[u8]]);
}

impl std::ops::Add<Setsum> for Setsum {
    type Output = Setsum;
    fn add(self, rhs: Setsum) -> Setsum;
}

impl std::ops::Sub<Setsum> for Setsum {
    type Output = Setsum;
    fn sub(self, rhs: Setsum) -> Setsum;
}
```

Figure 3: Syntactic sugar. Insert and remove update the internal digest to reflect the added value. The vectored variants allow for inserting elements like key-value pairs without having to perform concatenation prior to the insertion in much the same way as vectored I/O.

4 An End-to-End Example

This section walks through an end-to-end application of setsum to a proposed replicated database backed by a log-structured merge-tree [7]. We will generate a setsum in the replication stream and logically cover maintenance aspects of the database with the provided setsum. In this way, the replication stream always dictates the contents of the database. Once by replicating the data and once by specifying the data's checksum.

First, we'll look at the replication log itself. This is the point at which data gets ingested into the system. We will maintain a checksum over the replication stream so that the state of the database for each transaction is known. Each transaction builds on the immediately previous transaction's state to provide its own setsum digest. The setsum at all times represents the data added and removed by the replication stream.

We'll then extend the checksum to cover a log-structured merge-tree and show how to preserve the database's checksum while compacting away data to reclaim space. In this manner the setsum acts like double-entry accounting to make sure that the only data that disappears is data to be intentionally thrown away. The data must be thrown away the same in both the output file and the setsum.

Finally, we'll look at how to keep a checksum over the backups so that backups can be incrementally verified regularly without performing a full restore.

4.1 Replication Log

Database systems like MySQL [3] replicate a log that dictates the changes to the database. There are two primary methods by which replication happens. In the first mode of operation, queries themselves are transmitted through the replication stream and execute downstream on replicas. This mode of operation assumes the data should be the same so the query will execute deterministically and lead to the same outcome on secondary replicas as was achieved on the main replica. The second method is to execute the query on the main replica, watching what data it touches so that it can replicate the pre- and post-images of the transaction. These are the locks held, and rows removed and added. Propagating the query is convenient because a compact query can have large impact on the database. Propagating the pre- and post-images guarantees the replica will execute the transaction the same as the primary replica.

We can augment either replication method with setsums over the data to keep an up-to-date checksum over the database. The database can, at any time, take a snapshot of the data and compare it to the setsum observed at the time of the snapshot. For systems without a snapshot primitive, the snapshot can be achieved by taking nodes out of the replication stream to hash data at rest. It is common in database operations to take nodes out of the replication stream for such actions. An alternative would be to restore from backup and compare against the setsum of the latest transaction included in the backup.

```
acc = Setsum::default();
for row in rows_removed() {
    acc.remove(row);
}
for row in rows_added() {
    acc.insert(row);
}
acc + previous_transaction_setsum
```

Figure 4: The code run for each transaction in the replication stream. The result of `acc.digest()` represents at all times the set of item held by the database. Note that while we initialize `acc` to the previous transaction's value, it would be just as natural and acceptable to initialize it to zero and then use `merge` to merge it with the previous transaction's value.

Figure 4 shows an example of how to maintain a rolling checksum over the replication log with only local computation. Items get removed from the set when removed by the replication stream, and added to the set when added to the replication stream. The accumulator will capture the state of the database so that a fresh scan of the database would yield the same value.

Propagating a setsum alongside a transaction provides a stronger form of replicated transaction than queries or pre-/post-images by themselves. The replica can compute a setsum of its behavior and verify that the construction of the setsum is the same as was used to create the setsum. In this way a setsum can be used to detect if a replica diverges when replicating database queries. The same protection extends to replicating the pre- and post-images, but with considerably less sparkle as the transaction's side effects are defined by the pre- and post-images.

4.2 Log-Structured Merge Tree Compaction

Log-structured merge trees [7] are a data structure that writes data to the root of a tree and moves it upwards through the tree via a process known as *compaction*. In log structured merge trees like LevelDB [2] or RocksDB [4], the tree itself comprises files on disk in sorted runs, and compaction merges from one sorted run into the next higher sorted run by writing new files and erasing old files. Each time compaction runs, it runs the risk of dropping too much data.

The primary reason for compaction in an LSM-tree is to reclaim space from deleted and overwritten data. Because data is always ingested at the root, deletes happen as a write of a *tombstone* that masks the value to readers and new values are written in parallel to old values. When multiple versions of a key exist—whether by tombstone or multiple writes to the same key—readers pay a performance penalty. Compaction

```
inputs = Setsum::default();
compact = Setsum::default();
dropped = Setsum::default();
outputs = Setsum::default();
for file in provided_inputs() {
    inputs.insert(file.setsum());
}
for kv_pair in input {
    compact.insert(kv_pair);
    if !compacted_away(kv_pair) {
        write_to_output(kv_pair);
    } else {
        dropped.insert(kv_pair);
    }
}
for file in written_outputs() {
    outputs.insert(file.setsum());
}
assert_eq!(inputs, compact);
assert_eq!(inputs, merge(outputs, dropped));
```

Figure 5: The code for checking that compaction does not drop data extraneously. Note that while its shown as a single process here, it could be the case that multiple processes, representing multiple releases of software can verify the compaction by operating on setsums.

removes tombstones and the data they mask to reclaim space and re-organize the tree. A bug in compaction that spuriously drops data would be disastrous as it is hard to detect such data loss as a user of the system.

We can augment compaction with a setsum that tracks what data gets eliminated. A verification process, colloquially called a *verifier* can check that what it expects compaction to drop is all that gets dropped, and it can do this solely via inspection of the inputs to compaction and the setsums generated during compaction. This allows two distinct processes to communicate about what they observe in the compaction. One process can perform the compaction, while another, perhaps from an older and more well-tested release, verifies the compaction.

In this way, we can have multiple versions of software cross-check each other and communicate solely via the checksums of the data they see. A verifier can then check that checksums of like data match. Only when the verifier succeeds will the two processes apply the compaction and make its state become canonical.

Figure 5 demonstrates the compaction algorithm. We work with four checksums to compute the input. The first checksum is over the inputs. It can be computed by looking at the setsum embedded within each file's metadata. The second checksum is over the read data. This checksum should match the input checksum because it reads the same data. The third checksum is over data that gets dropped. When data isn't written to the output it gets "written" to the dropped setsum. The fourth checksum is the union of the output's metadata setsums. The four checksums capture all aspects of compaction.

The equation for balancing compaction's setsums is:

$$c_{input} = c_{compact} = merge(c_{outputs}, c_{dropped})$$

We can see that the way these setsums interrelate allows for accounting of what was dropped. Two processes can communicate the setsums for their outputs and dropped values and be confident that when the setsums match, the compaction was performed the same in both processes.

4.3 Backup Verification

Setsums present a unique way to verify backups for systems like LevelDB and RocksDB. Both systems create immutable files, called *sorted-string-tables* (or SSTs), on disk that change rarely and only with compaction. Because these files are immutable, it suffices to store alongside each file its setsum. The setsums of files included in the manifest can be used to check the manifest's overall setsum, and then be used to check each individual file on disk produces the same checksum. Thus, a backup can be verified without restoring completely from backup and taking a fresh hash; instead, the process verifying the backup can get away with resources comparable to a single SST.

Verifying a LevelDB or RocksDB backup, then, becomes a game of capturing the setsums securely for later reference, ensuring that their union matches the replication stream. When streaming the backup, each SST is compared to its setsum and the total is maintained for comparison to the manifest. If a file diverges, it will be detected by the individual file setsums and the reason for the checksum failure can be investigated.

5 Evaluation

Setsum is provided in Rust via the `setsum` crate, version 0.3.0. In all benchmarks using the sugared API, we link the crate directly. In microbenchmarks justifying implementation choices, code was selected from the crate for presentation alongside the alternative choices.

All benchmarks are run on a Lenovo Thinkpad X280 with Core i7-8550U CPU, 8 GB of RAM, and Samsung SSD 980 PRO 2TB hard drive running Ubuntu 22.04. The A/C adapter was connected and power governor set to performance mode for all trials.

5.1 Random

All random strings used in testing were generated using the `guacamole` and `armnod` crates. Guacamole generates a linearly-seekable stream of 2^{70} bytes. This differs from a random number generator because movement from i to $i+x$ in

Test Name	/dev/urandom	Guacamole
sts_serial 4	PASSED	WEAK
sts_serial 9	PASSED	WEAK
sts_serial 13	WEAK	PASSED
rgb_bitdist 4	WEAK	PASSED
rgb_permutations 2	WEAK	PASSED
rgb_lagged_sum 24	WEAK	PASSED
rgb_lagged_sum 25	PASSED	WEAK
*	PASSED	PASSED

Figure 6: DieHarder results for `/dev/urandom` and Guacamole. All tests pass with weak results or better. Any test not listed passed for both `/dev/urandom` and Guacamole.

```
armnod --number 1000 --chooser-mode set-once
armnod --number 1000
    --chooser-mode set-zipf --zipf-theta 0.999
```

Figure 7: Generate 1000 strings and then select from them according to a zipf distribution.

the input seeks proportionately far in the output stream. Consequently, the seed is predictable. Moving from seed i to seed $i+1$ moves exactly 64 B in the output stream. This allows us to take indices into the randomness that are equidistant from each other—a property that follows directly from taking equidistant inputs to the guacamole.

The guacamole algorithm was formed by taking the Salsa ciper [5] and optimizing for a constant key and plaintext. The algorithm was unrolled by hand to propagate the changes through. Consequently, guacamole functions like a faster, less strong salsa as far as speed and entry go. Comparisons to the Linux kernel using the dieharder [1] tool show that guacamole is comparable in strength to `/dev/urandom`. Figure 6 shows the results of tests that passed with weak results.

The `armnod` crate provides tools for generating UTF-8 strings from guacamole. It provides command-line tools for generating a fixed set of strings from `guacamole`. Figure 7 shows two example command lines for armnod. The first generates strings from a fixed set, returning each string exactly once. The second command draws from the set of strings according to a Zipf distribution with a theta of 0.999.

5.2 Hash Function Selection

The strength of the setsum comes in-part from the strength of the checksum that maps elements to their one-element-setsum. We will choose by assumption a cryptographic-strength hash as the hash for the setsum. The API permits alternate implementations of sugar using the same 256-bit group operators by exposing the group operators directly. In this section we will compare two options of 256-bit checksums and evaluate their fitness to power setsum.

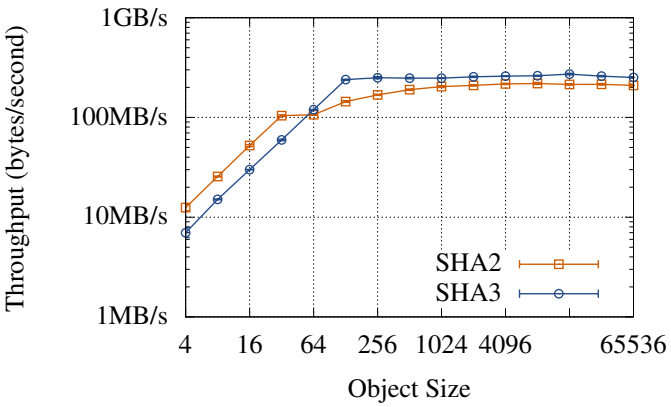

Figure 8: Throughput for hashing one megabyte of data consisting of elements varying in size between 8 and 65536 bytes. Maximum throughput of SHA2 is 219.1 MB/s and maximum of SHA3 is 271.8 MB/s.

For this experiment, we used guacamole to generate 1 MB of data consisting of strings of different lengths. The overall data remained the same size to ensure that all data fits in the CPU cache, so that we report the efficiency of the operations, not effects of the memory hierarchy. For each set of parameters, we measure the latency taken to hash the 1 MB and report the reciprocal as throughput measured in megabytes per second. For 64 B strings and longer the SHA3 algorithm performs better by approximately 24%. All results report the mean across 1000 trials and were measured to have a standard deviation tight enough to appear as zero on the graph.

The SHA2 algorithm was chosen by the setsum crate prior to these results. A future version will consider adopting SHA3.

5.3 Group Operator

The group operator we chose to implement in Figure 2 can be improved upon. We start with the observation that the group operator uses division to implement modular arithmetic. Because it is division modulo prime numbers there are not compiler nor hardware tricks that can be used to turn the division into something other than division. The compiler and hardware aren't privy to an insight that we can make using modular arithmetic. Any valid setsum digest has 8 32-bit integers, each of which is less than a specified prime. When adding two setsums together it is easy to see that $s_1 + s_2 < 2p$ because $s_1 < p$ and $s_2 < p$. We will never create a setsum that's more than $2p$. Consequently we can do a conditional subtraction to turn a value in the range $[p, 2p)$ into one that's $[0, p)$.

To decide the implementation of the group operator, we compared the difference between using modular arithmetic and a conditional subtraction of p. For this experiment, we generated $1,000,000$ setsums using guacamole and timed

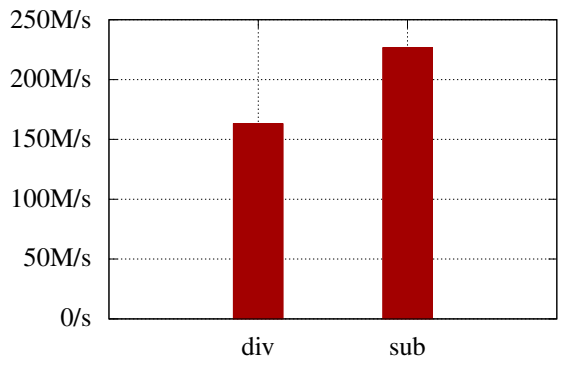

Figure 9: Throughput of the Abelian group operator in operations per second. When using division the group operator is able to sustain 163 million operations per second. Changing to a conditional subtraction is able to sustain 227 million operations per second.

how long it took to take the union across all setsums. We ran the benchmark as a warm-up for five seconds and then collected the next 1000 measurements. Figure 9 reports the mean throughput of both methods. We can see that the conditional subtraction approach achieves a throughput that is 39% higher than the approach based upon division.

5.4 Inverse

The inverse operator is the final part of our setsum. This operator is well defined for all values and is required to remove an element from the setsum. Figure 10 shows the performance of the inverse operator in operations per second. This value was measured by constructing one million random setsums using guacamole, then measuring the time taken to take the inverse of all one million. The reported value is converted to operations per second.

5.5 Database

As part of our evaluation, we wrote a sorted-string-table library in Rust so that we could test the ingest and verify paths of the replicated database design. This library supports the ingestion and verify paths for SSTs and is available as the `sst` crate. To test our design, we generated four different data sets of data with object size ranging from 72 B to 16 kB. Each data set is a 1 GB data set that is pre-generated in advance using guacamole. The first data set has 8 B keys and 64 B values. The other sets range in size up to 16 kB.

Figure 11 shows the throughput of ingest for all four workloads. The reported throughput is the average throughput for ingesting the pre-generated 1 GB of data as a series of new sorted string tables. We can see that the throughput scales

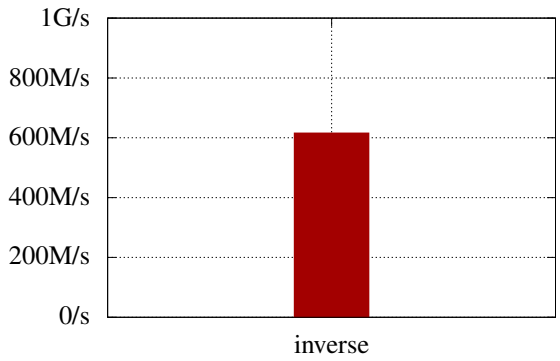

Figure 10: The group inverse operator's speed in operations per second. This is the computation necessary to remove the inverse of a set from a setsum. The setsum can perform 615 million setsum inverse operations per second.

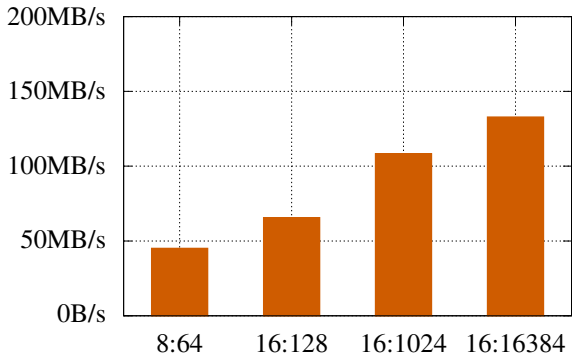

Figure 11: Throughput for four different key-value store workloads that ingest data into a key-value store.

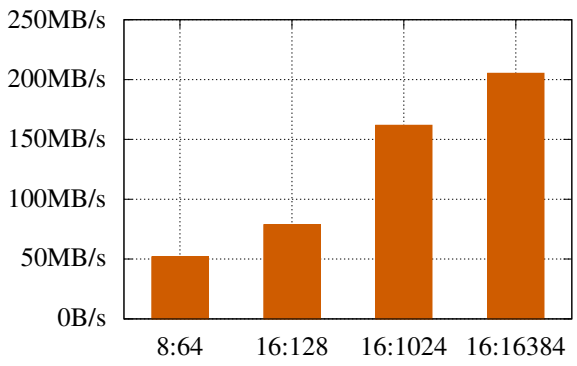

Figure 12: Throughput for four different key-value store workloads that verify the data in a key-value store.

with size of the data and ranges from 45 MB/s for the small objects to 133 MB/s for the large objects.

6 Conclusion

This paper introduced the concept of a *setsum* over an unordered set of byte strings. This is a new class of checksum, the likes of which is not present within the literature. To demonstrate the value of setsum, we walked through an example database that extends setsum coverage from where data is ingested all the way to the backups.

We hypothesize that other structures like setsum exist, with the defining characteristic being the relationship between a checksum and the data checksummed: As computation moves in the data space, so too does parallel computation in the checksum space. We are actively exploring ways to use cryptography to power more verifiers in the future.

Availability

Setsum is available in Rust via the `setsum` crate. The sorted string table library is available via the `sst` crate. The `guacamole` and `armnod` crates are likewise available. All code is Apache 2.0 licensed for free distribution.

References

[1] dieharder. https://webhome.phy.duke.edu/~rgb/General/dieharder.php.

[2] LevelDB. https://github.com/google/leveldb.

[3] MySQL. https://www.mysql.com.

[4] RocksDB. https://rocksdb.org.

[5] The salsa20 core. https://cr.yp.to/salsa20.html.

[6] Lara Alcock. *How to think about Abstract Algebra*. Oxford University Press, New York, 2021.

[7] Patrick E. O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth J. O'Neil. The log-structured merge-tree (lsm-tree). *Acta Informatica*, 33(4):351–385, 1996.