LSM Trees in Adversarial Environments

Hayder Tirmazi
City College of New York
hayder.research@gmail.com

ABSTRACT

The Log Structured Merge (LSM) Tree is a popular choice for key-value stores focusing on optimized write throughput while maintaining performant, production-ready read latencies. LSM stores rely on a probabilistic data structure called the Bloom Filter (BF) to optimize read performance. In this paper, we focus on adversarial workloads that lead to a sharp degradation in read performance by impacting the accuracy of BFs used within the LSM store. Our evaluation shows up to 800% increase in the read latency of lookups for popular LSM stores. We define adversarial models and security definitions for LSM stores. We implement adversary resilience into two popular LSM stores, LevelDB and RocksDB. We use our implementations to demonstrate how performance degradation under adversarial workloads can be mitigated.

PVLDB Reference Format:
Hayder Tirmazi. LSM Trees in Adversarial Environments. PVLDB, 14(1): XXX-XXX, 2020.
doi:XX.XX/XXX.XX

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/jadidbourbaki/db_attacks

1 INTRODUCTION

A large number of modern key-value stores are based on Log-structured Merge Trees (LSM Trees) [8, 19]. LSM trees allow for write-optimized [8], highly configurable [28] storage while being relatively simple to implement. These qualities have made them common as the storage engine for a large number of commercial and open-source database systems including LevelDB [16] from Google, bLSM [27] and cLSM [15] from Yahoo, RocksDB [21] and Cassandra [1] from Meta, WiredTiger [22] from MongoDB, Monkey [8] from Harvard's Data Systems Lab, Apache HBase [2], [11] from DGraph and many others [9, 19].

LSM stores use Bloom Filters [4, 5] to reduce unnecessary disk access. This strategy depends on Bloom Filters maintaining a low false positive rate (FPR). In normal workloads, this works well as Bloom Filters filter non-existent keys, so the LSM store does not have to look on disk [8]. An adversary can strategically insert keys into an LSM store that saturate its Bloom Filters, drastically raising their FPR. In this case, lookups for non-existing keys (called zero-result lookups [8]) require multiple disk accesses, increasing query latency by up to 800% according to our experiments. An adversary only needs a modest number of well-chosen insertions to render Bloom Filters in LSM stores ineffective. This motivates the need for adversarially resilient mechanisms in LSM store.

We make the following contributions in this paper.

(1) We define rigorous adversarial models for LSM store and reason about their performance under computationally bound adversaries.
(2) We demonstrate significant performance degradation under adversarial workloads, showing an increase in lookup latency by up to 800%. Our experiments are conducted on two popular LSM stores, LevelDB and RocksDB.
(3) We introduce a lightweight and provably secure mitigation strategy that employs keyed pseudorandom permutations (PRPs) to obfuscate key placement.
(4) We implement our mitigation in LevelDB and RocksDB, and demonstrate that it reduces the impact of adversarial workloads while maintaining performance.

2 PRELIMINARIES

In this section, we first go over notation and rigorously define an LSM store. We then define a set of axioms that hold for a large number of popular LSM store implementations. Lastly, we use this axiomatic framework to introduce multiple aspects of LSM store design and performance include the performance of zero-result lookups and the use of auxiliary data structures.

Notation. We review notation common in adversarial data structure literature [12, 13, 24, 29]. Given set S, we write $x \xleftarrow{\$} S$ to mean that x is sampled uniformly randomly from S. For set S, we denote by $|S|$ the number of elements in S. The same notation is used for a list $\mathcal{L}$. We write variable assignments using $\leftarrow$. If the output is the value of a randomized algorithm, we use $\xleftarrow{\$}$ instead. For a randomized algorithm A, we write output $\leftarrow \mathrm{A}_r(\mathrm{input}_1, \mathrm{input}_2, \cdots, \mathrm{input}_l)$, where $r \in \mathcal{R}$ are the random coins used by A and $\mathcal{R}$ is the set of possible coins. For natural number n, we denote the set $\{1, \cdots, n\}$ by $[n]$. Table 1 contains a summary of all the notation used in this paper.

2.1 LSM stores

An LSM Tree consists of $\mathcal{N}$ levels [8]. L_i indicates level i in the LSM Tree. L_0 is typically an in-memory buffer, while the remaining levels $[L_1, \cdots, L_{\mathcal{N}}]$ consist of data in secondary storage including SSDs, Hard Disks, and external storage nodes connected over the network. We formalize the general syntax and behavior of an LSM store. Given an LSM store, Λ, we denote the set of public parameters of an LSM store by Φ. The public parameters contain all the knobs relevant to the LSM store implementation including the number of levels in the LSM Tree as well as the knobs of any auxiliary data structures (discussed below) used by the LSM store. We denote the elements stored in Λ by $\mathcal{L}_\Lambda$ (a list). We denote the state of Λ

Proceedings of the VLDB Endowment, Vol. 14, No. 1 ISSN 2150-8097.
doi:XX.XX/XXX.XX

by $\sigma \in \Sigma$ where Σ is the space of all possible states of Λ. Λ can store elements from any finite domain $\mathfrak{D}$, where $\mathfrak{D} = \cup_{l=0}^{L}\{0,1\}^{l}$ for any natural number $L \in \mathbf{N}$. An LSM store, Λ consists of three algorithms.

Construction. $\sigma \leftarrow\!\!\$\; C_r(\Phi)$ sets up the initial state of an empty LSM store with public parameters Φ.

Insertion $\sigma' \leftarrow\!\!\$\; I_r((k,v),\sigma)$, given a tuple $(k,v) \in \mathfrak{D} \times (\mathfrak{D} \cup \{\perp\})$, returns the state σ' after insertion. After insertion, the key k is an element of $\mathcal{L}_\Lambda$. The value $\perp$, referred to as a tombstone [8], is used to indicate a deletion. Deletions have the same control flow as insertions in LSM stores [8].

Query. $v \leftarrow Q(k,\sigma)$, given an element $x \in \mathfrak{D}$ returns a value $v \in \mathfrak{D} \cup \{\perp\}$. The return value is either the value inserted by I_r for key k, or $\perp$ if no such key was inserted. In the case of a deletion, a $\perp$ is returned as the tuple $(k,\perp)$ was inserted by I_r.

Note that, unlike C_r and I_r, the query algorithm Q is not allowed to change the value of the state. While C_r and I_r are allowed to use random coins, Q is deterministic. A class of LSM stores can be uniquely identified by its algorithms: $\Lambda = (C_r, I_r, Q)$. We also assume all three algorithms always succeed and their outputs are correct with probability 1.

2.2 LSM Axioms

We define two axioms to reason about the performance of an LSM store, the Axiom of Cost (Axiom 1) and the Axiom of Recency (Axiom 2). Our axioms hold for many popular LSM store implementations including LevelDB [16], RocksDB [21], Monkey [8], WiredTiger [22], HBase [2] and Cassandra [1].

Axiom 1 (Axiom of Cost)**.** *If $i > j$, then $E_C(L_i) \geq E_C(L_j)$ where $E_C(L_x)$ is the expected cost of accessing an entry in L_x.*

LSM stores optimize for writes (inserts, deletes, and updates) using Axiom 1. The LSM store immediately stores written entries in the in-memory buffer in L_0 without accessing slower storage in higher levels [8]. When the L_0 buffer reaches capacity, it is sorted (by key) and flushed to L_1. These sorted arrays are called *runs* [8, 10, 19]. We refer to a run x within a level L_i as $R_{i,x}$. We denote the number of runs within a level L_i by $\mathcal{N}_R(L_i)$. The value of $\mathcal{N}_R(L_i)$ for a given LSM Tree depends on the merge policy of the LSM store [8].

Axiom 2 (Axiom of Recency)**.** *For an entry (k,v) in $R_{i,x}$ and an entry (k,v') in $R_{j,y}$ (with the same key k) if one of the following two conditions holds, then value v was written before value v': $i > j$ (Case 1), or $i = j$ and $x > y$ (Case 2).*

Using these axioms, we can explain LSM store design and performance.

2.3 Zero-result Lookups.

In an LSM Tree with $L_\mathcal{N}$ levels, when any key k is queried, the LSM store begins at the in-memory buffer at L_0 and traverses from L_0 to $L_\mathcal{N}$ in ascending order. If the LSM store finds a matching key at level L_i, the query returns early [8]. Entries at higher levels are superseded by more recent entries at lower levels thanks to Axiom 2, so looking further is unnecessary. For a key whose most recent entry is present in run $R_{i,x}$, the number of runs an LSM store needs to probe is at most $\sum_{l=0}^{i}\sum_{r=0}^{N_R(L_l)} E_C(R_{l,r})$ where $E_C(r)$ is the expected I/O cost of probing run r.

A query on a key k that is not stored in the LSM Tree is called a zero-result lookup. Such queries have a high worst-case I/O cost

Symbol	Description
General Notation	
$\mathfrak{D}$	Finite domain of elements
S	A set of elements
$x \leftarrow\!\!\$\; S$	x sampled randomly from S
$\|S\|$	Cardinality of set S
$[n]$	Set $\{1,\ldots,n\}$
LSM Store Notation	
$\sigma \in \Sigma$	State of LSM store
Λ	An LSM store
Φ	Public parameters
L_i	Level i in LSM Tree
$R_{i,x}$	Run x in level L_i
$N_R(L_i)$	Number of runs in L_i
$E_C(L_i)$	Expected access cost in L_i
(k,v)	Key-value pair
$C_r(\Phi)$	Construction algorithm
$Q(k,\sigma)$	Query operation for key k
$I_r((k,v),\sigma)$	Insert operation
$\perp$	Tombstone (deletion marker)
Bloom Filter Notation	
$\Pi = (C_{\dagger r}, Q_\dagger)$	Bloom Filter (BF) construction/query
$\sigma_\dagger$	Internal state of BF
$m_\dagger$	BF bit array size
$k_\dagger$	Number of BF hash functions
$h_i(k)$	i^{th} BF hash function
$P_B(R_{i,x})$	BF false positive probability
Adversarial Model Notation	
$\mathcal{A} \in \mathbb{A}$	Computationally-bound adversary
$\mathcal{A} \in \mathbb{A}_{BF}$	BF-targeting adversary
λ	Security parameter
t	Number of adversary queries
Smash-Lsm	Adversarial game for LSM stores
Smash-Bloom	Adversarial game for Bloom Filters
F_κ	Keyed pseudorandom permutation
ϵ	False positive/security bound
Oracle Notation	
$\mathcal{O}_Q(k)$	returns $\top$ if $\exists$ a BF Π in LSM Λ s.t $\Pi(k) = \top$
$\mathcal{O}_R$	Returns state $\sigma_\dagger$ of each BF Π_i in LSM Λ
$\mathcal{O}_I(k,v)$	Inserts (k,v) into LSM Λ
$\mathcal{O}_C(\Phi)$	Constructs LSM Λ with parameters Φ
$\mathcal{O}_{\dagger Q}(k)$	Queries BF Π for k
$\mathcal{O}_{\dagger R}$	Returns state $\sigma_\dagger$ of BF Π

Table 1: Mathematical notation used in the paper

because the LSM store must probe every run within every level before the LSM store can be sure the key does not exist. Zero-result lookups are very common in practice [8, 27]. They have been the focus of LSM store analysis and new LSM store designs proposed by prior work [8, 9]. A zero-result lookup is the worst-case lookup time [8] because the number of runs the LSM store must prove now is $\sum_{l=0}^{N}\sum_{r-0}^{N_R(L_l)} R_{l,r}$. This is the worst case for a point query because this is a probe of every run present in the LSM store.

2.4 Auxiliary Data Structures

For each run $R_{i,x}$ in an LSM Tree, modern LSM stores store two auxiliary data structures in main memory [8]: a Bloom Filter (BF) [4] and an array of fence pointers [8, 17]. The fence pointers contain min/max information for the disk pages that store run $R_{i,x}$ [16, 21]. In a point query, the LSM store uses this array of fence pointers when traversing a level to figure out which disk page to read when searching for a key [8]. The LSM store only has to read one disk page for each run.

For a point query, the LSM store first probes a run's Bloom Filter. The LSM store only accesses the run in secondary storage if the corresponding BF returns positive. If the run indicated by the BF does indeed have the key, the LSM store returns the query early following Axiom 1. However, the BF is a probabilistic data structure with one-sided errors, so it can have false positives with some probability [4, 5]. In the case of a false positive, the query continues to run [8]. Having a Bloom Filter in main memory for each run reduces the expected I/O cost of the LSM store for a point query, depending on the false positive probability of the BF. For a key whose most recent entry is present in run $R_{i,x}$, the LSM store needs to probe, in expectation:

$$E_C(R_{0,0})\cdot\left(\sum_{l=0}^{i}\sum_{r=0}^{N_R(L_{i-1})} P_B(R_{l,r})E_C(R_{l,r})+\sum_{r=0}^{x-1}P_B(R_{i,r})R_{i,r}+R_{i,x}\right)$$

where $P_B(r)$ is the false positive probability of the BF corresponding to run r. The extra $E_C(R_{0,0})$ factor comes from the fact that each BF resides in the main memory, which involves the cost of accessing a main memory page. Since $R_{0,0}$ (the run at L_0) is also a main memory page. Therefore, the expected cost of accessing a main memory page is the same as that of accessing run $R_{0,0}$.

3 ADVERSARIAL MODEL

In this section, we first discuss the performance degradation caused by adversarial attacks and the feasibility of such attacks. We then introduce a game-based adversarial model for LSM stores and propose a security definition for LSM stores.

3.1 Performance Degradation

The key idea for our adversarial workloads is to target the false positive rate (FPR) of the per-run Bloom Filters implemented in an LSM store. Since most popular LSM stores [8, 16, 21] use BFs with non-cryptographic hashes, such attacks are feasible and inexpensive. Even if an adversary is allowed a very small bound on insertions, they can still fully insert enough elements to saturate the BFs. For a Standard Bloom Filter implementation with a memory budget of $m_\dagger$ bits and $k_\dagger$ hashes, only $\frac{m_\dagger}{k_\dagger}$ adversarial insertions are required

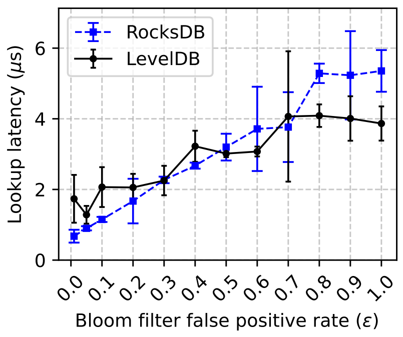

Figure 1: Performance degradation of zero-result lookups on a uniformly random query workload

to make the BF's FPR become 1. Figure 1 shows the impact of Bloom Filter FPR on the zero-result lookup latency for LevelDB [16] and RocksDB [21]. We insert 10M keys in batch sizes of 10K. We benchmark 50K uniformly randomly sampled zero-result look-ups for each experiment. Between the insertion stage and the lookup stage, we save and re-open the store to mitigate the effects of caching. As BF FPR increases, the lookup latency of LevelDB and RocksDB becomes 2x and 8x respectively.

3.2 Feasibility of Attacks

We first give an overview of how the standard implementation of a Bloom Filter first suggested in [4] and used in LevelDB [16] and RocksDB [21] works. A Standard Bloom Filter (SBF) construction is a zero-initialized array of $m_\dagger$ bits [5, 14] and requires a family of $k_\dagger$ independent hash functions, $h_{i,m} : \mathfrak{D} \mapsto [m_\dagger]$ for all $i \in [k_\dagger]$[5, 29]. Upon setup, For each element $x \in S$ (S is the set being encoded by the Bloom Filter), the bits $h_i(x)$ are set to 1 for $i \in [k_\dagger]$. When querying an element x, we return true if all $h_i(x)$ map to bits that are set to 1. If there exists an $h_i(x)$ that maps to a bit that is 0, we return false.

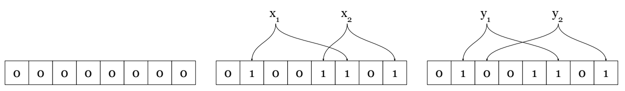

Figure 2: Bloom Filter example ($m_\dagger$ = 8, $k_\dagger$ = 2) adapted from [3]. Inserted x_i is hashed $k_\dagger$ times, setting mapped bits. Queried y_i is hashed k times. If a mapped bit is unset, $y_i \notin S$. Otherwise y_i is in S or a false positive

The expected number of entries to fully saturate an SBF is $\lfloor \frac{m_\dagger \log m_\dagger}{k_\dagger} \rfloor$ [14]. An adversary can pick well-chosen items to insert such that sets $k_\dagger$ previously unset bits (one new bit for each hash function). This reduces the number of entries to fully saturate an SBF down to $\lfloor \frac{m_\dagger}{k_\dagger} \rfloor$. [14] show many practical saturation attacks for real-world Standard Bloom Filter deployments. This is particularly easy to do when a non-cryptographic hash is invertible. However, even a brute force strategy where the adversary crafts well-chosen inputs offline does not take a large amount of time if the memory budget of a Bloom Filter is small. Since the entire point of a

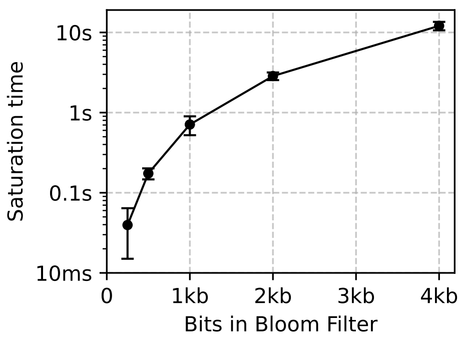

Figure 3: Time taken by a brute force algorithm running sequentially on a local machine to saturate LevelDB's Bloom Filter implementation with various memory budgets.

Bloom Filter is for it to use a small number of bits (otherwise we can simply replace it with a non-probabilistic data structure such as a hash-table-based set), this is commonly the case. Our inexpensive experimental setup (Section 4) running a sequential brute force algorithm can fully saturate LevelDB's Bloom Filter implementation with a memory budget of up to 4 kilobits in about 11 seconds.

3.3 Game-based Model

We define a self-contained game-based [3, 23, 24] adversarial model for LSM stores that is restricted to adversaries that target an LSM store's Bloom Filters. For cryptography-focused readers, we also define a more general simulator-based [20] adversarial model for any class of adversaries in Appendix A. We will be assuming our adversary is *efficient.* We will assume that the adversary works in non-uniform probabilistic polynomial time (n.u. p.p.t). This standard notion is used to model efficient adversaries in cryptography literature [25]. Moreover, we will restrict our model to $\mathbb{A}_{\text{BF}}$, the set of efficient adversaries that target the Bloom Filters in the LSM store. We discuss other adversarial targets in Appendix **??**. To capture the notion of adversary resilience in an LSM store setting, we propose a game inspired by prior work on game-based adversary models for other probabilistic data structures [23, 24]. We first define the notion of a Bloom Filter in our adversarial game.

Definition 3.1 (Bloom Filter). *Let $\Pi = (C_{\dagger_r}, Q_\dagger)$ be a pair of polynomial time algorithms. $C_{\dagger_r}$ is randomized, it takes a set $S_\dagger \subseteq \mathcal{D}$ as input and outputs a state $\sigma_\dagger$. $Q_\dagger$ is deterministic, it takes as a state $\sigma_\dagger$ and a key $k \in \mathcal{D}$ as input and returns y in$\{\top, \bot\}$. Π is an (n, ϵ)-BF if for all sets $S^\Pi \subseteq \mathcal{D}$ of cardinality n and suitable public parameters $\Phi_\dagger$, the following two properties hold.*

- *Completeness:* $\forall x \in S : P[Q_\dagger(x, C_{\dagger_r}(\Phi_\dagger, S_\dagger)) = \top] = 1$
- *Soundness:* $\forall x \notin S : P[Q_\dagger(x, C_{\dagger_r}(\Phi_\dagger, S_\dagger)) = \top] \leq \epsilon$

where the probabilities are over the coins of $C_{\dagger_r}$.

In our game, the adversary $\mathcal{A} = (\mathcal{A}_C, \mathcal{A}_Q)$ consists of two parts: $\mathcal{A}_C$ chooses a list $\mathcal{L} \subset \mathcal{D} \times (\mathcal{D} \cup \{\bot\}$. $\mathcal{A}_Q$ gets $\mathcal{L}$ as input and attempts to find a false positive key k given only oracle access to the LSM store Λ initialized with $\mathcal{L}$. $\mathcal{A}$ succeeds if key k is not among the queried elements and is a false positive. We measure the success probability of $\mathcal{A}$ for the random coins in Λ and in $\mathcal{A}$. For computationally bound adversaries, our game includes a security parameter λ which is given to the adversary $\mathcal{A}$ in unary 1^λ and given to the LSM store implicitly as part of the public parameters Φ. For LSM store $\Lambda = (C_r, I_r, Q)$, the adversary $\mathcal{A}$ is allowed access to two oracles. The first oracle, $\mathcal{O}_Q(k)$, returns $\top$ if there exists a Bloom Filter Π in LSM store Λ such that $\Pi(k) = \top$. Otherwise, it returns $\bot$. The second oracle, $\mathcal{O}_R$ returns a list $\sigma_{\dagger_i}$ of internal states for each Bloom Filter Π_i in the LSM store Λ.

Game 3.1 (Smash-Lsm). *We have a challenger Υ, security parameter λ, and a n.u p.p.t adversary $\mathcal{A} = (\mathcal{A}_C, \mathcal{A}_Q)$. $\mathcal{A}$ is a Bloom Filter targeting adversary, $\mathcal{A} \in \mathbb{A}_{BF}$. We have an LSM store Λ with public parameters Φ. We define the game Smash-Lsm($\mathcal{A}, t, \lambda$) as follows.*

Step 1 $\mathcal{L} \leftarrow\$ A_C(1^\lambda)$ *where* $\mathcal{L} \subset \mathcal{D} \times (\mathcal{D} \cup \{\bot\})$ *and* $|\mathcal{L}| = n$.
Step 2 Υ *initializes* Λ *with* $\sigma \leftarrow\$ C_r(\Phi)$ *and invokes* $\sigma \leftarrow\$ I_r((k, v), \sigma)$ *for each* $(k, v) \in \mathcal{L}$.
Step 3 $k_\mathcal{A} \leftarrow\$ \mathcal{A}_Q(1^\lambda, \mathcal{L})$. $\mathcal{A}_Q$ *can make at most t queries* $k_1, \cdots, k_t$ *to* $\mathcal{O}_Q$ *and unbounded queries to* $\mathcal{O}_R$.
Step 4 *We denote the set of keys in* $\mathcal{L}$ *by* L_k . *If* $k_\mathcal{A} \notin \mathcal{L}_k \cup \{k_1, \cdot, k_t\}$ *and* $\mathcal{O}_Q(k_\mathcal{A}) = \top$, $\mathcal{A}$ *wins (Smash-Lsm returns* $\top$*). Otherwise,* $\mathcal{A}$ *loses (Smash-Lsm returns* $\bot$*).*

Clearly, if adversary $\mathcal{A}$ wins the Smash-Lsm game, $\mathcal{A}$ has *forced an unnecessary run access* by exploiting Bloom Filter false positives in the LSM store.

3.4 Security Definition

We now define our notion of the security of an LSM store Λ against Bloom Filter targeting adversaries $\mathbb{A}_{\text{BF}}$ using the Smash-LSM game.

Definition 3.2. *For an LSM store Λ, we say that Λ is (n, t, ε)-secure against Bloom Filter targeting adversaries if for all n.u p.p.t adversaries $\mathcal{A} \in \mathbb{A}_{BF}$ and for all lists of cardinality n, for all large enough $\lambda \in \mathbb{N}$, it holds that*

$$Pr[\textsc{Smash-Lsm}(\mathcal{A}, t, \lambda) = \top] \leq \varepsilon$$

where the probabilities are taken over the random coins of Λ and $\mathcal{A}$.

The definition essentially states a requirement for declaring that an LSM store is secure against computationally bound adversaries that are looking to sabotage its Bloom Filters. We have to prove that no such adversary can, with high probability, find a previously unused key that results in a false positive for any of the Bloom Filters being used by the LSM store. This must be proven, even if the adversary chooses the initial list of keys to be inserted into the LSM store, even if the adversary makes t queries to the LSM store's Bloom Filters before answering, and even if the adversary can look at the internal state of the Bloom Filters.

4 ADVERSARY RESILIENCE

We construct an adversary-resilient implementation of LevelDB [16] and RocksDB [21]. Our secure construction is simple and easily pluggable in any popular LSM store. Instead of writing and reading keys directly, we simply patch the LSM store API to maintain a keyed pseudorandom permutation (PRP) and read/write the permuted value of the key. The values are kept unchanged. This does not affect the correctness of the LSM store. As an example, assume we use a PRP that permutes a to x and b to y. Inserting two key-value pairs (a, v_1) and (b, v_2) will instead insert (x, v_1) and (y, v_2).

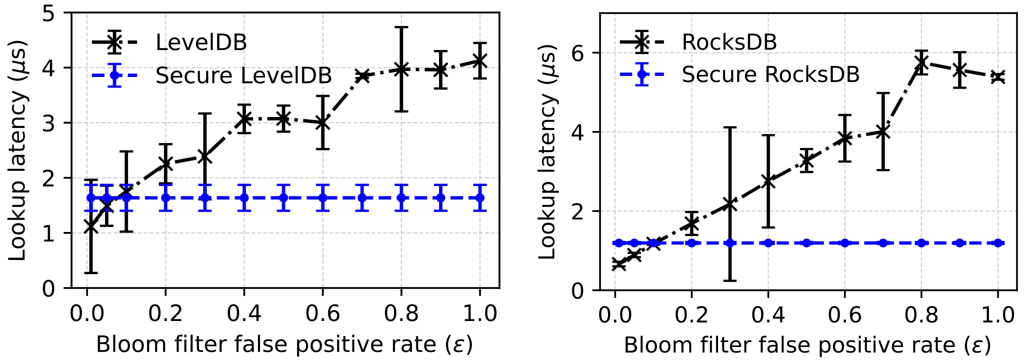

Figure 4: Zero-result lookup performance on a uniformly random query workload for LevelDB (left) and RocksDB (right) with adversarial resilience.

When key a is queries, since our PRP is consistent, it is mapped again to x and Λ returns the correct corresponding value v_1. PRPs are bijections therefore the keys themselves can also be recovered by running the inverse permutation. We informally summarize our result here and then rigorously prove it below in Section 5.

Theorem 4.1. *Let* $\Lambda = (C_r, I_r, Q)$ *be an LSM store using* m *bits of memory for its Bloom Filters. If pseudo-random permutations exist, then there exists a negligible function* $\text{negl}(\cdot)$ *such that for security parameter* λ*, there exists an LSM engine* Λ' *that is* $(n, t, \epsilon + \text{negl}(\lambda))$*-secure against Bloom Filter targeting adversaries* $\mathbb{A}_{BF}$ *using* $m' = m+\lambda$ *bits of memory for its Bloom Filters.*

Proof Sketch: Applying a keyed PRP to LSM store keys before insertion and query ensures that adversaries cannot control key placements to force high false positive rates. Correctness is due to PRP bijectivity: queries map consistently to transformed keys, retrieving the correct values. Security follows from PRP indistinguishability. If an adversary could significantly increase false positives, they could distinguish the PRP from random, which contradicts the existence of PRPs. We need extra λ bits of memory to store the PRP's secret key.

4.1 Implementation & Experimental Setup

We use the experimental setup discussed here for all experiments in this work. We use an Apple M2 processor with 8 cores, 8 GB of Memory, a 128 KB L1 cache, a 1 MB L2 cache, and 256 GB of SSD storage out of which 245 GB is usable storage. We evaluated the most recent versions of RocksDB (9.10) and LevelDB (1.23). We run each experiment 5 times and display the median, the error bars indicate standard deviations. We implement security as an easily pluggable module written in C++ 17 for both LevelDB and RocksDB. Our implementation relies on the hardness of AES. Our implementation uses the AES-128 implementation provided by OpenSSL v3.4.0 in CTR mode. We use a 16 byte (λ = 128 bits) AES secret key to parameterize the PRP. To prevent secret key leakage, production-scale deployments can rely on secret stores and ephemeral key rotation mechanisms. We have released our implementation and evaluation as open-source software.

4.2 Performance

We compare our adversary-resilient implementations to untouched releases of LevelDB and RocksDB under adversarial workloads. Figures4 and 5 show performance under the same workload as Section 3. For zero-result lookups, our implementations reduce latency by 60% and 78% for LevelDB and RocksDB respectively. Our implementations are slower for workloads with existing keys (where not all the runs need to be probed, in expectation) by 30% and 6% for LevelDB and RocksDB respectively. We also measured the latency for 50 K uniformly random inserts with our implementation and saw a median latency increase of 39% and 49% for LevelDB and RocksDB respectively. These increases are due to the extra compute overhead of AES for every key. This is a constant compute cost that can potentially be reduced by using a hardware-implemented version of AES such as AES-NI [23] for Intel processors. This overhead also is not a major concern when I/O dominates lookup performance, which is frequently the case for LSM stores [8].

5 SECURITY PROOFS

We first define the notion of secure Bloom Filters (Def. 3.1). We use known definitions [13, 29] for Bloom Filter (BF) constructions. A BF, Π consists of two algorithms.

Construction. $\sigma_\dagger \leftarrow C_{\dagger_r}(\Phi_\dagger, S_\dagger)$ sets up the initial state of a BF with public parameters $\Phi_\dagger$ and a given set $S_\dagger \subseteq \mathfrak{D}$.

Query. $b \leftarrow Q_\dagger(x, \sigma_\dagger)$, given an element $x \in \mathfrak{D}$ returns a boolean $b \in \{\bot, \top\}$. The return value approximately answers whether $x \in S_\dagger$ ($b = \top$) or $x \notin S_\dagger$ ($b \neq \bot$).

The construction algorithm C_r is called first to initialize Π. The query algorithm Q is not allowed to change the value of the state. While C_r is randomized, Q is deterministic. Both algorithms always succeed. A class of BFs can be uniquely identified by its algorithms: $\Pi = (C_r, Q)$.

We now define a well-known [23, 24] security game for Bloom Filters. Our adversary, $\mathcal{A}_\dagger = (\mathcal{A}_{\dagger_C}, \mathcal{A}_{\dagger_Q}$ consists of two parts: $\mathcal{A}_{\dagger_C}$ chooses a set $S_\dagger \subset \mathcal{D}$. $\mathcal{A}_{\dagger_Q}$ gets $S_\dagger$ as input and attempts to find a false positive key k given only oracle access to the BF Π initialized with $S_\dagger$. We measure the success probability of $\mathcal{A}_\dagger$ for the random coins in Π and in $\mathcal{A}_\dagger$. For computationally bound

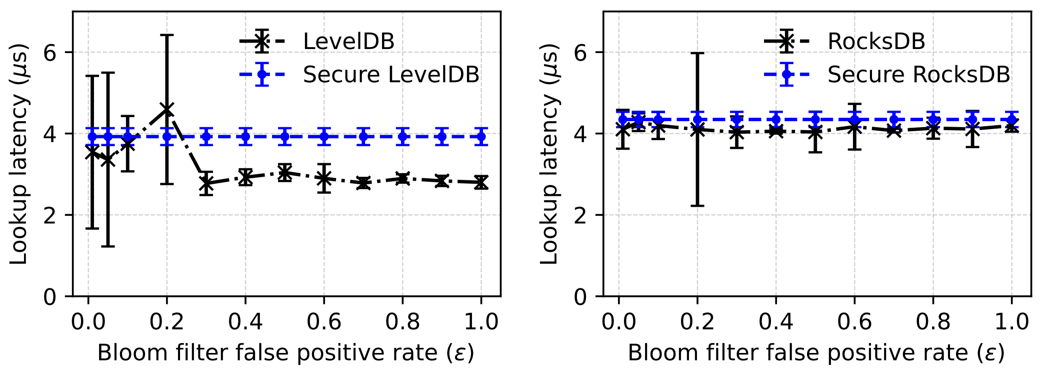

Figure 5: Existing query lookup performance on a uniformly random query workload for LevelDB (left) and RocksDB (right) with adversarial resilience.

adversaries, our game includes a security parameter λ which is given to the adversary $\mathcal{A}_\dagger$ in unary 1^λ and given to the BF implicitly as part of the public parameters $\Phi_\dagger$.

For a BF $\Pi = (C_{\dagger r}, Q_\dagger)$, the adversary $\mathcal{A}_\dagger$ is allowed access to two oracles. The first oracle, $\mathcal{O}_{Q_\dagger}(k)$, returns $C_{\dagger r}(k)$. The second oracle, $\mathcal{O}_{R_\dagger}$ returns $\sigma_\dagger$, the internal states for the Bloom Filter Π. Note that the internal state returned does **not** include any secret keys used in the construction. This assumption is consistent with prior work [12, 13, 23, 24, 29].

Game 5.1 (Smash-Bloom). *We have a challenger Υ, security parameter λ, and a n.u p.p.t adversary $\mathcal{A}_\dagger = (\mathcal{A}_{\dagger C}, \mathcal{A}_{\dagger Q})$. We have a Bloom Filter Π with public parameters $\Phi_\dagger$. We define the game Smash-Bloom($\mathcal{A}_\dagger, t, \lambda$) as follows.*

Step 1 $S_\dagger \xleftarrow{\$} A_{C_\dagger}(1^\lambda)$
Step 2 Υ *initializes* Π *with* $\sigma_\dagger \xleftarrow{\$} C_\dagger r(\Phi_\dagger, S_\dagger)$.
Step 3 $k_\mathcal{A} \xleftarrow{\$} \mathcal{A}_{Q_\dagger}(1^\lambda, S_\dagger)$. $\mathcal{A}_{Q_\dagger}$ *can make at most t queries* $k_1, \cdots, k_t$ *to* $\mathcal{O}_{Q_\dagger}$ *and unbounded queries to* $\mathcal{O}_{R_\dagger}$.
Step 4 *If* $k_\mathcal{A} \notin S_\dagger \cup \{k_1, \cdot, k_t\}$ *and* $\mathcal{O}_{Q_\dagger}(k_\mathcal{A}) = \top$, $\mathcal{A}$ *wins (Smash-Bloom returns* $\top$*). Otherwise,* $\mathcal{A}$ *loses (Smash-Bloom returns* $\bot$*).*

Definition 5.1. *For an Bloom Filter Π, we say that Π is (n, t, ε)-secure if for all n.u p.p.t adversaries $\mathcal{A}_\dagger$ and for all sets of cardinality n, for all large enough $\lambda \in \mathbb{N}$, it holds that*

$$Pr[\textsc{Smash-Bloom}(\mathcal{A}_\dagger, t, \lambda) = \top] \leq \varepsilon$$

where probabilities are over the random coins of Π and $\mathcal{A}_\dagger$.

Pseudo-random Permutations. We provide a brief self-contained treatment of a cryptographic construction called pseudorandom permutations adapted from [18] that will allow the construction of secure Bloom Filters. Let Perm_n be the set of all permutations on $\{0, 1\}^n$.

Definition 5.2. *Let an efficient permutation F be any permutation for which there exists a polynomial time algorithm to compute $F_k(x)$ given k and x, and there also exists a polynomial time algorithm to compute $F_k^{-1}(x)$ given k and x.*

Definition 5.3. *Let $F : \{0, 1\}^* \times \{0, 1\}^* \mapsto \{0, 1\}^*$ be an efficient, length-preserving, keyed function. F is a keyed permutation if $\forall k$, $F_k(\cdot)$ is one-to-one.*

Definition 5.4. *Let $F : \{0, 1\}^* \times \{0, 1\}^* \mapsto \{0, 1\}^*$ be an efficient keyed permutation. F is a pseudo-random permutation if for all probabilistic polynomial time distinguishers D, there exists a negligible function* $\mathtt{negl}$, *such that*

$$|\Pr[D^{F_k(\cdot)F_k^{-1}(\cdot)}(1^n) = 1] - \Pr[D^{f_n(\cdot)f_n^{-1}(\cdot)}(1^n) = 1]| \leq \mathtt{negl}(n)$$

where the first probability is taken over uniform choice of $k \in \{0, 1\}^n$ and the randomness of D, and the second probability is taken over uniform choice of $f \in \mathrm{Perm}_n$ and the randomness of D.

Secure Bloom Filters. We show a well-known result for Bloom Filters initially proved by [23]. The proof was expanded by [3] to make it clearer that it holds even for the case where an adversary has access to the internal state of a Bloom Filter. We include a self-contained proof here using our notation based on proofs in the two cited works.

Theorem 5.1. *Let $\Pi = (C_{\dagger r}, Q_\dagger)$ be a Bloom Filter using $m_\dagger$ bits of memory. If pseudo-random permutations exist, then there exists a negligible function* $\mathrm{negl}(\cdot)$ *such that for security parameter λ, there exists an $(n, t, \epsilon + \mathrm{negl}(\lambda))$-secure BF using $m_\dagger{}' = m_\dagger + \lambda$ bits of memory.*

Proof. We first demonstrate a secure construction. We then prove its security and correctness.

Construction: Choose a key $\kappa \in \{0, 1\}^\lambda$ for a pseudorandom permutation F_κ. Let $\Pi' = (C_\dagger{}'_r, Q'_\dagger)$ where

(1) $C_\dagger{}'_r(\Phi_\dagger, S_\dagger) = C_{\dagger r}(\Phi_\dagger, S'_\dagger)$ where $S'_\dagger$ is the permuted set $S_\dagger$ i.e. $S'_\dagger = \{F_\kappa(x) : x \in S_\dagger\}$.
(2) $Q'_\dagger(x, \sigma_\dagger) = Q_\dagger(F_\kappa(x), \sigma_\dagger)$.

$C'_{\dagger r}$ initializes Bloom Filter Π' with $S'_\dagger$. $Q'_\dagger$ on input x queries for $x' = F_\kappa(x)$. The only additional memory required is for storing κ which is λ bits long.

Security Proof: The security of Π' follows from a hybrid argument. Consider an experiment where F_κ in Π' is replaced by a truly random oracle $\mathcal{R}(\cdot)$. Since x has not been queried, $\mathcal{R}(x)$ is a truly random element that was not queried before, and we may think of it as chosen before the initialization of Π'. No n.u. p.p.t adversary $\mathcal{A}_\dagger$ can distinguish between the Π' we constructed using $\mathcal{R}(\cdot)$ and the Π' construction that uses the pseudo-random permutation F_κ by more than a negligible advantage. We can prove this by contradiction. Suppose that there does exist a non-negligible function $\delta(\lambda)$ such that $\mathcal{A}_\dagger$ can attack Π' and find a false positive with probability $\epsilon + \delta(\lambda)$. We can run $A_\dagger$ on Π' where the oracle is replaced by an oracle that is either random or pseudo-random, and return 1 if $\mathcal{A}_\dagger$ finds a false positive. This allows us to distinguish between $\mathcal{R}(\cdot)$ and $F_\kappa(\cdot)$ with probability $\geq \delta(\lambda)$. This contradicts the indistinguishability of pseudo-random permutations.

Correctness proof: Π' is still a valid Bloom Filter as per Def. 3.1. Π''s completeness follows from the completeness of the original BF Π. From the soundness of Pi, we get that the probability of x being a false positive in Π' is at most ϵ. Therefore, the probability of $\mathcal{A}_\dagger$ winning the Smash-Bloom game is $\Pr[\text{Smash-Bloom}(\mathcal{A}_\dagger, t, \lambda) = \top] \leq \varepsilon + \text{negl}(\lambda)$. □

We can now prove our main result regarding the security of LSM stores.

Theorem 5.2. *Let $\Lambda = (C_r, I_r, Q)$ be an LSM store using m bits of memory for its Bloom Filters. If pseudo-random permutations exist, then there exists a negligible function* negl$(\cdot)$ *such that for security parameter λ, there exists an LSM engine Λ' that is $(n, t, \epsilon + \text{negl}(\lambda))$-secure against Bloom Filter targeting adversaries $\mathbb{A}_{BF}$ using $m' = m+\lambda$ bits of memory for its Bloom Filters.*

Proof. We first demonstrate a secure construction. We then prove its security and correctness.

Construction: Choose a key $\kappa \in \{0,1\}^\lambda$ for a pseudorandom permutation F_κ. Let $\Lambda' = (C_r, I'_r, Q')$ where

(1) $I'_r((k, v), \sigma) = C_r((F_\kappa(k), v), \sigma)$
(2) $Q'(x, \sigma) = Q(F_\kappa(x), \sigma)$.

The only additional memory required is for storing κ which is λ bits long.

Security Proof: We prove this by contradiction. Suppose there does exist an n.u. p.p.t. adversary $\mathcal{A}$ that can with the Smash-Lsm game with probability greater than ε+negl(λ). Then by definition of oracle $\mathcal{O}_Q$, there exists a Bloom Filter Π' with the construction from Thm 5.1 for which $\mathcal{A}$ can generate false positives with probability higher than ε+negl(λ). So $\mathcal{A}$ can also win the Smash-Bloom game with a probability higher than ε + negl(λ). This is a contradiction as Thm 5.1 proves that no n.u p.p.t adversary can win the Smash-Bloom game against Bloom Filter constructions of type Π' with probability higher than ε + negl(λ).

Correctness Proof: The correctness of our construction follows from the fact that F_κ is a bijection so it does not affect the correctness of the internal Bloom Filters, fence pointers, or the algorithms called on the LSM tree. □

Since only the keys are permuted, not the values stored in the LSM store, we only need to do a forward permutation F_κ but not an inverse permutation F_κ^{-1} for a point query.

6 RELATED WORK

We discuss two areas of research relevant to our work: adversarial correctness of probabilistic data structures, and LSM store benchmarking and optimization.

Adversarial Data Structures. There have been many recent efforts to rigorously define a game-based [3, 6, 23, 24] and simulator-based [12, 13, 29] adversarial model for probabilistic data structures such as the Bloom Filter and the Learned Bloom Filter. In particular, the work of Naor et. al. [23, 24] was the first to show provably secure constructions for the Bloom Filter. There is also prior work showing feasible attacks on probabilistic data structures including the Bloom Filter [14] and the Learned Bloom Filter [26]. Our work builds upon these insights by applying them to LSM stores and designing countermeasures tailored to real-world storage systems.

LSM Store Benchmarking / Optimization. KVBench [30] is one of many works that focus on benchmarking workloads for LSM stores. Prior work on LSM store optimizations includes [8, 10, 17, 28]. Monkey [8] focuses on optimizing the memory budget allocation of the Bloom Filters used in LSM stores for better query performance. Huynch et. al. [17] use Large Language Models (LLMs) to tune the design knobs of LSM stores. All prior work discussed primarily focuses on benchmarking and improving performance under non-adversarial workloads. To the best of our knowledge, this is the first paper to rigorously define an adversarial model for LSM stores and propose concrete, provably secure LSM store constructions.

7 OPEN PROBLEMS

We leave the community with open problems in four categories that emerge from our work.

7.1 Adversarial Targets.

The experiments in our work focus on adversaries that target Bloom Filters in LSM stores via well-chosen insertions. We conduct our experiments on, LevelDB and RocksDB, both of which use a merge policy called leveling [8, 17]. There is a vast universe of adversarial targets we leave as open problems that emerge from our work. Empirical evidence showing performance degradation from these targets would help greatly in the design of future LSM stores.

Deleted Insertions. A direct follow-up for our attacks comes from the observation that Bloom Filters in LSM trees cannot perform deletions. This means that if an adversary $\mathcal{A}$ sabotages Bloom Filters via well-chosen insertions and then deletes the keys it inserted, the false positive rate of the Bloom Filters remains high unless the Bloom Filter is reconstructed from existing keys.

Range Queries. Range queries [8, 19] look for a larger number of keys in multiple levels of an LSM tree compared to point queries. Therefore, they are potentially more vulnerable to adversarial workloads. Adversary $\mathcal{A}$ can insert data in a manner that forces range queries to access an excessive number of runs, increasing read latency. Studying adversarial range query complexity and designing more efficient prefetching and merging strategies could help mitigate these attacks.

Merge Policy. There are different merge policies within LSM stores including leveling and tiering [8]. Prior work [17] has shown that leveling is more robust to uncertain (but not explicitly adversarial) workloads than tiering. We conducted our evaluations on LevelDB and RocksDB, both of which use leveling. We might potentially see higher performance degradation on LSM stores that rely on tiering.

7.2 General Adversary-resilience

In our work, we have shown a construction that provides adversary-resilience in an LSM store against $\mathbb{A}_{\text{BF}}$, the set of computationally bound Bloom Filter targeting adversaries. We leave answering the following follow-up questions as open problems:

(1) Does our construction provide adversary resilience against other classes of adversaries?
(2) Is there a construction, perhaps using our simulator-based model (*Appendix A*) guaranteeing reasonable adversary resilience against all computationally bound adversaries?
(3) Is a construction possible (either only for $\mathbb{A}_{\text{BF}}$ or for the general set $\mathbb{A}$) that provides adversary-resilience if the adversary is computationally-unbounded?

Helpful starting points for this work include [23, 24] who provide an adversary-resilient construction of the Bloom Filter against computationally unbounded adversaries, and the simulator-based security constructions of [12, 13].

7.3 Learned Bloom Filters.

We conjecture that replacing Bloom Filters used by an LSM store with Learned Bloom Filters [3] with a better false positive rate may lead to better performance. Adversary resilience for Learned Bloom Filters can be solved using the construction of [3]. We leave experimental validation of this as an open problem. Similar research for Adaptive Bloom Filters (or Learned Adaptive Bloom Filters [7]) will also be interesting. A helpful starting point for this work is [3] who provide an adversary-resilient construction of the Learned Bloom Filter, called the Downtown Bodega Filter, in the same game-based setting as [23].

7.4 Existence of an Ideal-World Simulator

In our simulator-based adversarial model (Appendix A), we have not proved the existence of an ideal-world simulator. We leave proving the existence of and providing a construction for an ideal-world simulator for LSM stores as an open problem. The constructions of an ideal-world simulator for probabilistic data structures by [12, 13] may be a good starting point. However, their constructions require two properties in a data structure: function decomposability and reinsertion variance. Informally, reinsertion invariance requires the internal state of a probabilistic data structure to remain unchanged when the same key is reinserted. This does not necessarily apply to LSM stores.

8 CONCLUSION

In this paper, we investigate the performance of LSM stores under adversarial workloads. Our analysis shows that adversaries can significantly increase zero-result lookup latencies. We introduce a lightweight, provably secure mitigation strategy based on keyed pseudorandom permutations. Our implementation in LevelDB and RocksDB shows that this approach effectively reduces adversarial impact while maintaining overall system performance. Our work demonstrates the importance of adversarial resilience in storage systems. We introduce the community to several important open problems calling for both theoretical and experimental research into broader classes of attacks and secure designs for LSM stores.

REFERENCES

[1] Apache. 2025. Cassandra. https://cassandra.apache.org/.
[2] Apache. 2025. HBase. http://hbase.apache.org/.
[3] Allison Bishop and Hayder Tirmazi. 2024. Adversary Resilient Learned Bloom Filters. Cryptology ePrint Archive, Paper 2024/754. https://eprint.iacr.org/2024/754
[4] Burton H. Bloom. 1970. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM* 13, 7 (July 1970), 422–426. https://doi.org/10.1145/362686.362692
[5] Andrei Broder and Michael Mitzenmacher. 2003. Network Applications of Bloom Filters: A Survey. *Internet Mathematics* 1, 4 (2003), 485 – 509.
[6] David Clayton, Christopher Patton, and Thomas Shrimpton. 2019. Probabilistic Data Structures in Adversarial Environments. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security* (London, United Kingdom) *(CCS '19)*. Association for Computing Machinery, New York, NY, USA, 1317–1334. https://doi.org/10.1145/3319535.3354235
[7] Zhenwei Dai and Anshumali Shrivastava. 2020. Adaptive Learned Bloom Filter (Ada-BF): Efficient Utilization of the Classifier with Application to Real-Time Information Filtering on the Web. In *Advances in Neural Information Processing Systems*, H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin (Eds.), Vol. 33. Curran Associates, Inc., 11700–11710. https://proceedings.neurips.cc/paper_files/paper/2020/file/86b94dae7c6517ec1ac767fd2c136580-Paper.pdf
[8] Niv Dayan, Manos Athanassoulis, and Stratos Idreos. 2017. Monkey: Optimal Navigable Key-Value Store. In *Proceedings of the 2017 ACM International Conference on Management of Data* (Chicago, Illinois, USA) *(SIGMOD '17)*. Association for Computing Machinery, New York, NY, USA, 79–94. https://doi.org/10.1145/3035918.3064054
[9] Niv Dayan and Stratos Idreos. 2019. The Log-Structured Merge-Bush & the Wacky Continuum. In *Proceedings of the 2019 International Conference on Management of Data* (Amsterdam, Netherlands) *(SIGMOD '19)*. Association for Computing Machinery, New York, NY, USA, 449–466. https://doi.org/10.1145/3299869.3319903
[10] Niv Dayan, Tamar Weiss, Shmuel Dashevsky, Michael Pan, Edward Bortnikov, and Moshe Twitto. 2022. Spooky: granulating LSM-tree compactions correctly. *Proc. VLDB Endow.* 15, 11 (July 2022), 3071–3084. https://doi.org/10.14778/3551793.3551853
[11] DGraph. 2025. BadgerDB. https://github.com/hypermodeinc/badger.
[12] Mia Filić, Keran Kocher, Ella Kummer, and Anupama Unnikrishnan. 2024. Deletions and Dishonesty: Probabilistic Data Structures in Adversarial Settings. In *Asiacrypt '24*. IACR.
[13] Mia Filić, Kenneth Paterson, Anupama Unnikrishnan, and Fernando Virdia. 2022. Adversarial Correctness and Privacy for Probabilistic Data Structures. In *CCS '22*. Association for Computing Machinery, 1037–1050.
[14] Thomas Gerbet, Amrit Kumar, and Cédric Lauradoux. 2015. The Power of Evil Choices in Bloom Filters. In *2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. IEEE, 101–112. https://doi.org/10.1109/DSN.2015.21
[15] Guy Golan-Gueta, Edward Bortnikov, Eshcar Hillel, and Idit Keidar. 2015. Scaling concurrent log-structured data stores. In *Proceedings of the Tenth European Conference on Computer Systems* (Bordeaux, France) *(EuroSys '15)*. Association for Computing Machinery, New York, NY, USA, Article 32, 14 pages. https://doi.org/10.1145/2741948.2741973
[16] Google. 2025. LevelDB. https://github.com/google/leveldb/.
[17] Andy Huynh, Harshal A. Chaudhari, Evimaria Terzi, and Manos Athanassoulis. 2022. Endure: a robust tuning paradigm for LSM trees under workload uncertainty. *Proc. VLDB Endow.* 15, 8 (April 2022), 1605–1618. https://doi.org/10.14778/3529337.3529345
[18] Jonathan Katz and Yehuda Lindell. 2014. *Introduction to Modern Cryptography, Second Edition* (2nd ed.). Chapman & Hall/CRC, n/a.
[19] Shubham Kaushik and Subhadeep Sarkar. 2024. Anatomy of the LSM Memory Buffer: Insights & Implications. In *Proceedings of the Tenth International Workshop on Testing Database Systems* (Santiago, AA, Chile) *(DBTest '24)*. Association for Computing Machinery, New York, NY, USA, 23–29. https://doi.org/10.1145/3662165.3662766
[20] Yehuda Lindell. 2016. How To Simulate It - A Tutorial on the Simulation Proof Technique. Cryptology ePrint Archive, Paper 2016/046. https://eprint.iacr.org/2016/046
[21] Meta. 2025. RocksDB. https://github.com/facebook/rocksdb.
[22] MongoDB. 2025. Wired Tiger. https://github.com/wiredtiger/wiredtiger.

[23] Moni Naor and Yogev Eylon. 2019. Bloom Filters in Adversarial Environments. *ACM Trans. Algorithms* 15, 3, Article 35 (jun 2019), 30 pages. https://doi.org/10.1145/3306193

[24] Moni Naor and Noa Oved. 2022. Bet-or-Pass: Adversarially Robust Bloom Filters. In *Theory of Cryptography*, Eike Kiltz and Vinod Vaikuntanathan (Eds.). Springer Nature Switzerland, Cham, 777–808.

[25] Rafael Pass and Abhi Shelat. 2010. A Course in Cryptography. Lecture Notes. Available at: https://www.cs.cornell.edu/courses/cs4830/2010fa/lecnotes.pdf.

[26] Pedro Reviriego, José Alberto Hernández, Zhenwei Dai, and Anshumali Shrivastava. 2021. Learned Bloom Filters in Adversarial Environments: A Malicious URL Detection Use-Case. In *2021 IEEE 22nd International Conference on High Performance Switching and Routing (HPSR)*. 1–6. https://doi.org/10.1109/HPSR52026.2021.9481857

[27] Russell Sears and Raghu Ramakrishnan. 2012. bLSM: a general purpose log structured merge tree. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data* (Scottsdale, Arizona, USA) *(SIGMOD '12)*. Association for Computing Machinery, New York, NY, USA, 217–228. https://doi.org/10.1145/2213836.2213862

[28] Viraj Thakkar, Madhumitha Sukumar, Jiaxin Dai, Kaushiki Singh, and Zhichao Cao. 2024. Can Modern LLMs Tune and Configure LSM-based Key-Value Stores?. In *Proceedings of the 16th ACM Workshop on Hot Topics in Storage and File Systems* (Santa Clara, CA, USA) *(HotStorage '24)*. Association for Computing Machinery, New York, NY, USA, 116–123. https://doi.org/10.1145/3655038.3665954

[29] Hayder Tirmazi. 2025. A Privacy Model for Classical & Learned Bloom Filters. Cryptology ePrint Archive, Paper 2025/125. https://eprint.iacr.org/2025/125

[30] Zichen Zhu, Arpita Saha, Manos Athanassoulis, and Subhadeep Sarkar. 2024. KVBench: A Key-Value Benchmarking Suite. In *Proceedings of the Tenth International Workshop on Testing Database Systems* (Santiago, AA, Chile) *(DBTest '24)*. Association for Computing Machinery, New York, NY, USA, 9–15. https://doi.org/10.1145/3662165.3662765

A SIMULATOR-BASED MODEL

In this section, we define a simulator-based [20] adversarial model for LSM stores inspired by the simulator-based model of [13] for Bloom Filters. The game-based adversarial model of Section 3 is restricted to adversaries that target an LSM store's Bloom Filters. The simulator-based adversarial model, on the other hand, applies to any class of adversaries. We reuse the notation defined in Section 2.

Adversarial Setting. Let $\Lambda = (C_r, I_r, Q)$ be an LSM store, with public parameters Φ. Let $\mathbb{A}$ be any given set of adversaries playing Game A.1 that are bound by time $t_{\mathbb{A}}$ and are allowed at most $\psi i, \psi q, \psi_r$ queries to oracles $\mathcal{O}_I, \mathcal{O}_Q, \mathcal{O}_R$ respectively. Let $\mathbb{D}$ be the set of all distinguishers bound by time $t_{\mathbb{D}}$. To establish any results regarding the behavior of LSM stores in the presence of adversaries, we need to compare it to what the behavior of an LSM store without the presence of an adversary is expected to be. We call any simulator that provides a view of the LSM store without adversarial interference an *ideal-world* simulator. Let $\mathcal{S}$ be an ideal-world simulator bound by time $t_{\mathcal{S}}$ that provides a *non-adversarially-influenced view* of Λ to adversaries in set $\mathbb{A}$. For simpler notation, we denote $\Psi = (\psi_i, \psi_q, \psi_r)$ and $T = (t_{\mathbb{A}}, t_{\mathcal{S}}, t_{\mathbb{D}})$.

Definition A.1 (Adversary Resilient LSM store)**.**
We say that Λ is an (Ψ, T, ε)-resilient LSM store if for all adversaries $\mathcal{A} \in \mathbb{A}$ and for all distinguishers $\mathcal{D} \in \mathbb{D}$ in Game A.1, it holds that:

$$|Pr[Real(\mathcal{A},\mathcal{D})=1] - Pr[Ideal(\mathcal{A},\mathcal{D},\mathcal{S})=1]| \leq \varepsilon.$$

where the constructions $\Lambda, \mathbb{A}, \mathbb{D}, \Psi,$ and T are defined in the Adversarial Setting section above. The probabilities are taken over the random coins used by I_r and $_Cr$ within $\mathcal{O}_I$ and $\mathcal{O}_C$, as well as any random coins used by $\mathcal{A}, \mathcal{S}$, or $\mathcal{D}$.

Game A.1. *We have an adversary $\mathcal{A}$, a simulator $\mathcal{S}$, a distinguisher $\mathcal{D}$, and public parameters Φ.*

```
Real-Or-Ideal($\mathcal{A}, \mathcal{S}, \mathcal{D}, \Phi$)
1: $d \leftarrow\$ \{0,1\}$
2: if $d = 0$   // Real
3:   $\sigma \leftarrow\$ C_r(\Phi)$
4:   $y \leftarrow\$ \mathcal{A}^{\mathcal{O}_I, \mathcal{O}_Q, \mathcal{O}_R}$
5: else   // Ideal
6:   $y \leftarrow\$ \mathcal{S}(\mathcal{A}, \Phi)$
7: return $d' \leftarrow\$ \mathcal{D}(y)$

Oracle $\mathcal{O}_I(k, v)$
$\sigma \leftarrow\$ I_r(k, v, \sigma)$

Oracle $\mathcal{O}_Q(k)$
return $Q(k, \sigma)$

Oracle $\mathcal{O}_R()$
return $\sigma$
```