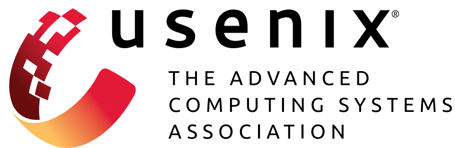


Prefix Siphoning: Exploiting LSM-Tree Range Filters For Information Disclosure

Adi Kaufman, *Tel Aviv University;* Moshik Hershcovitch, *Tel Aviv University & IBM Research;* Adam Morrison, *Tel Aviv University*

https://www.usenix.org/conference/atc23/presentation/kaufman

This paper is included in the Proceedings of the
2023 USENIX Annual Technical Conference.

July 10–12, 2023 • Boston, MA, USA

978-1-939133-35-9

Open access to the Proceedings of the
2023 USENIX Annual Technical Conference
is sponsored by

Prefix Siphoning: Exploiting LSM-Tree Range Filters For Information Disclosure

Adi Kaufman* Tel Aviv University

Moshik Hershcovitch* Tel Aviv University & IBM Research

Adam Morrison Tel Aviv University

Abstract

Key-value stores typically leave access control to the systems for which they act as storage engines. Unfortunately, attackers may circumvent such read access controls via *timing attacks* on the key-value store, which use differences in query response times to glean information about stored data.

To date, key-value store timing attacks have aimed to disclose stored values and have exploited external mechanisms that can be disabled for protection. In this paper, we point out that *key* disclosure is also a security threat—and demonstrate key disclosure timing attacks that exploit mechanisms of the key-value store itself.

We target LSM-tree based key-value stores utilizing *range filters*, which have been recently proposed to optimize LSM-tree range queries. We analyze the impact of the range filters SuRF and prefix Bloom filter on LSM-trees through a security lens, and show that they enable a key disclosure timing attack, which we call *prefix siphoning*. Prefix siphoning successfully leverages benign queries for non-present keys to identify prefixes of actual keys—and in some cases, full keys—in scenarios where brute force searching for keys (via exhaustive enumeration or random guesses) is infeasible.

1 Introduction

Key-value stores serve as the storage engines of many cloud and enterprise systems, from object caches [44,46,47] through stream processing [6,14,54] to database systems [2,29,31,41]. Performance of these modern data intensive systems often depends on their key-value storage engine's performance [51]. Consequently, research on key-value stores overwhelmingly focuses on *efficiency*: from I/O efficiency of writes [20,21], point queries [18,19], and range queries [45,65] to memory efficiency [24,43], energy efficiency [5], multi-core scalability [37,58], and reducing I/O write amplification [51].

But systems also depend on their key-value storage engine for the *security* of stored data. This dependency is not obvious, because key-value stores typically provide only a dictionary abstraction without access control mechanisms [16,30,38,40], leaving access control to the system. Systems enforce access control by mediating user accesses to the key-value store, often based on access control lists (ACLs) stored as value metadata in the key-value store. While this approach blocks users from directly making unauthorized queries, users may still be able to indirectly glean information about restricted data if the key-value store is vulnerable to *timing attacks* [11].

A timing attack exploits differences in query response times to glean information about stored data. A system using a key-value store that is vulnerable to timing attacks can itself become vulnerable to such attacks, because the system's query response time depends on the storage engine's response time, making differences in key-value query response times manifest as differences in the system's response times.

To date, key-value store timing attacks [55,56] have aimed to disclose stored values. We point out, however, that *key* disclosure is also a security threat. In some systems, keys can *explicitly* contain secret data. For example, database systems that use key-value storage engines (e.g., CockroachDB, YugabyteDB, or MyRocks) encode rows (or subsets of rows) onto keys [7,26,28,32]. This makes key disclosure equivalent to database data disclosure. Keys may also be *implicitly* secret, with users expecting them to be hard to obtain. For instance, in object storage systems, such as Amazon S3, identifying valid keys may create an insecure direct object reference vulnerability [48], which enables attackers to probe access to the objects associated with the disclosed keys.

Unfortunately, resilience to timing attacks is not a goal in existing key-value efficiency work—in fact, such resiliency can be at odds with improved performance. In this paper, we demonstrate this trade-off: we analyze key-value store performance mechanisms through a security lens and show that they enable a key disclosure timing attack.

We focus on write-optimized key-value stores based on log-structured merge (LSM) trees [49], which are in widespread use [12,13,15,18,20,22,30,37,42,51,57,60]. In these designs, data in secondary storage consists of multiple immutable files called *SSTables*. LSM-trees can efficiently sustain write-intensive workloads, but queries may require multiple I/Os to search the many SSTables [49,57]. LSM-trees minimize unnecessary I/Os by issuing the I/O only if the queried key is likely to be in the SSTable. Likelihood is determined by querying an in-memory *filter* [10], which space-efficiently *approximately* represents the SSTable's contents. Specifically, filter queries can make "one-sided" errors: if the queried key is present in the SSTable, then the filter always returns true; but for a small fraction of non-present keys, the filter might return a false positive response.

Standard filters can answer point (single-key) queries [8,10,33], but do not support range queries of the form "does the

*Both authors contributed equally to this research.

SSTable contain a key in range $[X,Y]$." Consequently, LSM-tree range queries must search the many SSTables, performing multiple superfluous I/Os [65]. To address this problem, recent work has proposed *range filters*, which are filters that support range queries in addition to point queries. Range filters such as SuRF [65] and RocksDB's prefix Bloom filter (PBF) [25] compactly store some or all prefixes of each of the SSTable's keys, and leverage this information to answer range and point queries.

From a security perspective, however, we show that *certain range filters enable a key disclosure timing attack on LSM-trees*. We describe an attack framework, called *prefix siphoning*, which exploits general range filter characteristics present in both SuRF and PBF. Prefix siphoning successfully leverages benign point queries for non-present keys to identify prefixes of actual keys—and in some cases, full keys—in scenarios where brute force searching for keys (via exhaustive enumeration or random guesses) is infeasible.

Prefix siphoning targets systems with the common design paradigm of storing a key's ACLs as part of its value [1, 4], which means that to check access permissions, the system's query handling always tries to read the queried key's value from the key-value store. Prefix siphoning exploits this property to determine if a random key is one on which the LSM-tree's filter returns a false positive. This is possible because whether the filter returns true or false can be determined by the attacker observing the query's response time, as the filter's response decides whether the LSM-tree performs I/Os. For range filters meeting our characterization, finding a false-positive key implies that the false-positive key shares a prefix with some stored key. Prefix siphoning then performs further point queries—tweaking the queried key—to maximize the length of the disclosed prefix. Prefix siphoning can sometimes subsequently perform a limited enumeration search to fully identify the stored key. Our prefix siphoning implementation performs multiple such steps concurrently, ultimately extracting multiple keys or prefixes.

We evaluate prefix siphoning against SuRF and PBF analytically as well as empirically and demonstrate its feasibility in practice. For example, we successfully use prefix siphoning to extract 64-bit stored keys from a RocksDB [30] datastore employing SuRF in minutes, whereas brute force search of this key space is infeasible. Our analysis and evaluation also quantify the cost of prefix siphoning, showing that it effectively reduces the key search space size by multiple orders of magnitude. For instance, SuRF prefix siphoning requires ≈ 10 M queries to disclose a key from a 50 M 64-bit key dataset—implying a 40992$\times$ reduction of the key search space size.

Our results draw attention to the security vs. performance trade-offs in key-value store design, and encourage practitioners and researchers to evaluate the security impact of their work. We hope that our characterization of vulnerable range filters will spur research on more secure filters.

2 Background

This section provides background on key-values stores (§ 2.1), LSM-trees (§ 2.2), and filters (§ 2.3).

2.1 Key-value stores

A key-value store exposes a dictionary-like abstraction with the following operations.

- *put*(k,v). A *put* stores a mapping from key k to value v. If key is already present in the store, its value is updated.
- *get*(k). The *get()* (or point query) returns the value associated with the requested key.
- *range_query*(k_1,k_2). A range query returns all key-value pairs falling within the given range.

Due to their simple and general abstraction as well as high performance, key-value stores serve as the storage engines for many, more complex systems. Examples of such systems include database systems (e.g., Cassandra [42], MongoDB [2], and MySQL [3]) and storage systems (e.g., CEPH [1]).

2.2 LSM-based data stores

The log-structured merge (LSM) tree [49] is a popular choice as the core storage structure for write-optimized key-value stores, which must sustain write-intensive workloads. An LSM-tree consists of levels, each of which contains multiple immutable static sorted table (SSTable) files storing key/value pairs. Two SSTs at the same level never overlap in the key range they store, but SSTables at different levels may overlap.

A *put* request inserts the key-value pair into an in-memory buffer called the Memtable, which is the LSM-tree's only mutable storage object. Once the Memtable fills up, its data is flushed to secondary storage as an SSTable file. The LSM-tree periodically performs *compaction*, where it unifies SSTs between levels to eliminate duplicate (stale) key-value pairs.

A *get* query searches for the target key in a top-down manner: first in the Memtable and subsequently in the relevant SSTable (if it exists) in each level. Searching an SSTable requires I/Os to read it from secondary storage. Once the key is found, its value is returned and the query completes.

However, this design penalizes queries, which may require multiple I/Os to search many SSTables [49, 57]. In particular, a *get()* for a *non-present* key (not associated with any value) searches every level before failing. This not only increases the query response time, but may "thrash" the page cache by reading in many SSTables which will not be accessed later.

LSM-trees minimize unnecessary I/Os by issuing the I/O only if the queried key is likely to be in the SSTable. Likelihood is determined by querying an in-memory filter (described in § 2.3), which space-efficiently *approximately* represents the SSTable's contents. The LSM-tree only reads an SSTable from secondary storage if its filter returns true for

the queried key. As a result, most non-present key queries can respond without performing I/Os.

Likewise, a range filter (§ 2.3.1) can answer both point and range queries with one-sided errors. Using a range filters instead of a standard filter enables an LSM-tree to avoid superfluous I/Os also for range queries, which can improve range query throughput by orders of magnitude [45].

2.3 Filters

A *filter* [10] is a data structure used to approximately represent a set D a of keys. A filter can be *immutable* or *dynamic*. An immutable filter is provided D upon its creation and can subsequently only be queried. A dynamic filter learns D dynamically, via *insert* operations.

Responses for filter queries allow "one-sided" errors: if $k \in D$, then a query for k returns true; but for a fraction of keys $k \notin D$, a query for k might answer true instead of false. We say k is a *positive/negative key* if a filter a query for k answers true or false, respectively. A positive key k is a *false positive* if $k \notin D$. We also say that the filter *passes* positive keys and *rejects* negative keys.

Filters are compared by their space efficiency and false-positive rates. Space efficiency is measured in bits per key. The *false-positive rate* (FPR) of a filter is the probability over keys not in D of being a false positive. I.e., $FPR = FP/(FP+NK)$, where FP is the number of false-positive keys and NK is the number of negative keys. Filters typically have configurable FPRs, with lower FPRs requiring more bits per key for increased accuracy [8, 10, 33].

Bloom filters A Bloom filter [10] is a widely-used dynamic filter (e.g., the default filter of RocksDB). It consists of an m-bit array and j hash functions $H_1, \ldots, H_j$. The parameters m and j determine the filter's FPR and space. Insertion of key k sets the bits indexes $H_1(k), \ldots, H_j(k)$. A query for k returns true if and only if all bit indexes $H_1(k), \ldots, H_j(k)$ are set.

2.3.1 Range filters

A *range filter* is a filter that also supports range queries with one-sided error: a query for $[a,b]$ returns true if there exists $k \in D \cap [a,b]$, but might also return true if $D \cap [a,b]$ is empty.

3 Motivation: avoiding key disclosure

We observe that keys stored in a key-value storage engine can contain sensitive data. It is therefore desirable that users are not able to efficiently discover stored keys that they are not authorized to access. Of course, users can always guess such keys and check if their queries return an authorization error, but such brute force searches are infeasible on large key spaces. The goal is for brute force search to be the only attack option, i.e., to block more efficient key extraction attacks.

Explicitly secret keys Some systems encode secret data in stored keys, which makes key disclosure equivalent to disclosure of the encoded data. For example, database systems such as CockroachDB, YugabyteDB, and MyRocks store table rows as values in a key-value storage engine, with the associated key consisting of the table's id and the row's primary key (one of the cell values). The motivation for this technique is that it enables the database system to perform efficient primary key lookups using key-value store range queries [7, 26, 28, 32].

Implicitly secret keys In many cases, keys are tacitly assumed to be secret or, at least, hard to guess. One example of implicitly secret keys are *object identifiers*. Many web applications and object storage systems maintain object id-to-value mappings in a key-value store. Key disclosure thus allows attackers to probe access to the associated objects, resulting in an insecure direct object reference vulnerability [48]. While objects typically have ACLs, users often neglect to configure these ACLs. This is not a hypothetical concern: for instance, there are numerous scanning tools for "open" (unprotected) Amazon S3 objects [9, 23, 50, 53, 61, 62], and open S3 objects have led to exfiltration of employee information, personal identification information, and other sensitive data [27].

4 Threat model

We consider a *high-level system*, such as a database system or object store, that utilizes a key-value storage engine to respond to user queries. Key ACLs are stored as part of the value associated with the key. As the high-level system performs key-value queries to satisfy a user's query, it checks the ACL of each key it accesses by inspecting the key's value. If the user is not authorized to read a key, the system returns a failure response to the user.

The attacker's goal is to identify keys stored in the system's key-value storage engine. The attacker cannot compromise the system (e.g., to run attack code) and cannot eavesdrop on requests performed by other users and/or on their responses. The attacker can only interact with the system by making requests via its interfaces, such as a representational state transfer (REST) API [34, Chapter 5].

We assume that the attacker can craft their requests in a way that causes the high-level system to make key-value store point queries for arbitrary keys (i.e., chosen by the attacker) while processing the request. For simplicity, we refer to this process as the attacker "querying the key-value store."

We make no assumption about the attacker's physical location with respect to the attacked system. We only assume that the attacker can observe microsecond-level timing differences in the response times of queries for different keys. Prior work has shown that this assumption is true for attacks over both local and wide area networks. For instance, Crosby et al. were able to measure a difference of 20 μs over the circa 2009 Internet (and 100 ns over a local area network) [17]. This ability

can be improved in specific cases. When attacking a system hosted in the public cloud, for example, the attacker can turn themselves into a local-area attacker by placing themselves in the datacenter hosting the target. Moreover, systems that process different requests concurrently (e.g., HTTP/2 servers) are vulnerable to concurrency-based timing attacks [36], which can observe timing differences of 100 ns over the Internet.

5 Prefix siphoning

Prefix siphoning is a general template for conducting timing attacks, extracting partial or full keys, on systems that use an LSM-tree based storage engine with a certain type of *vulnerable* range filter (for both point and range queries). The class of vulnerable range filters contains the filters SuRF [65] and RocksDB's prefix Bloom filter (PBF) [25].

Prefix siphoning exploits range filters that respond to point queries based on key prefix information, which exists to support range queries—i.e., filters where range query support affects the point query implementation. Accordingly, prefix siphoning is based *only on point queries* and does not perform range queries. Henceforth, therefore, the term "query" always refers to a *get()* point query. We leave exploring attacks against range queries to future work.

In the following, we describe the attack's high-level ideas (§ 5.1), characterize the class of vulnerable filters (§ 5.2), and present the attack template (§ 5.3). We describe instantiations of the attack against SuRF and PBF in §§ 6–7.

Notation We treat keys as sequences of symbols over an alphabet Σ (e.g., bytes). When x denotes a key or a set, then $|x|$ refers to the number of symbols or elements, respectively, that x contains.

5.1 High-level ideas

Prefix siphoning exploits an *inherent* trait of filter use in LSM-trees: that whether a key "passes" the filter determines if the LSM-tree searches the SSTable for the key to satisfy a query. This means that for SSTable files that do not reside in the OS page cache, the filter's output for a key significantly affects the LSM-tree's query response time. If the filter returns false for the key, the response is satisfied with only main memory access; otherwise, the LSM-tree needs to perform I/Os to read SSTables from secondary storage. Even for fast storage such as NVMe devices, the difference in query response times between these two cases is enough to affect the system's overall response time in an attacker-measurable way.

This basic filter trait suffices to mount an "approximate membership test" timing attack. The attack simply queries for the target key k and measures the response time. If the response time is fast (i.e., k is rejected by the filters), then k is definitely not stored in the LSM-tree. Otherwise (i.e., k passes some filter), then k is likely in the LSM-tree. The key k might also be a filter false positive and not exist in the LSM-tree, which occurs with a probability bounded by the filter's FPR.

Prefix siphoning starts by randomly generating keys until it finds a key that "passes" the membership test above. For random keys, passing the test overwhelmingly means that the key is a filter false positive. Crucially, it takes just hundreds of attempts to find a false-positive key, because filters are typically configured for FPRs of a few percents for space efficiency reasons [65].

Our main observation is that in vulnerable range filters, a false-positive key likely shares a prefix with some stored key k, whereas negative keys (rejected by the filter) do not (at least with high probability). The crux of a prefix siphoning attack is an algorithm exploiting this trait to *identify* the shared prefix k' through $O(|k|)$ further queries for modified keys iteratively derived from the initial false-positive key.

The revealed prefix of k can already contain sensitive information. But if the system's query responses distinguish between failures due to target key non-presence and lack of authorization, prefix siphoning can fully extract k by performing brute force search of the unknown suffix, thereby *extending* the revealed prefix to k.

Of course, a system whose responses distinguish between non-present and unauthorized keys is also vulnerable to "brute force" key guessing or enumeration attacks based using the above "membership test" primitive. But such attacks are infeasible for many key spaces (e.g., 64-bit or string keys). The point of prefix siphoning is to narrow down the search space by exploiting vulnerable range filters. Moreover, prefix siphoning extracts key prefixes even if the target system's responses do not reveal whether a key is non-present or unauthorized, whereas the "membership test" primitive cannot.

5.2 Vulnerable range filter characterization

We denote an instance of the filter by F and the set of keys it represents by D. A range filter is *vulnerable to prefix siphoning* if it has the following characteristics, denoted C1–C2. They say that a false-positive key κ likely shares a prefix with some key from D and that an attacker can efficiently identify this prefix by making queries for keys derived from κ.

C1 If κ is a false-positive key for F, then with high probability, κ shares a prefix with some $k \in D$.

C2 There exist the following probabilistic algorithms, which work by querying the system:

1. *FindFPK()*: Using an expected constant number of queries, outputs a random false-positive key κ.
2. *IdPrefix*(κ)*:* Given a false-positive κ, uses $O(|\kappa|)$ queries to identify the shared prefix k' that κ shares with some key $k \in D$, if such a prefix exists; otherwise, the output is unspecified.

The *FindFPK* and *IdPrefix* algorithms are specific to the

range filter design, and need to be developed by the attacker.[1] We refer to designing such algorithms for a range filter as *instantiating* the attack against that filter.

C2 implies existence of a timing attack, and is therefore formally sufficient to characterize the vulnerability. In practice, however, our attack instantiations rely on fundamental properties of filter use in LSM-trees. To highlight this aspect of the attacks, we explicitly capture these properties in C3.

C3 1. A $get(k)$ query's response time is measurably lower if k misses in every filter than if k hits in some filter.

2. The filter's FPR is small but non-negligible (e.g., 1% or 0.1%).

C3(1) implies that it is possible to distinguish negative from positive keys using query response times. It is trivially true because LSM-trees employ filters to speed up queries for which SSTable searching is superfluous, such as filter misses. Our attacks in this paper exploit microsecond-level time differences between queries satisfied completely from main memory and those that require I/O to secondary storage. (There remain time differences between queries that read an in-memory SSTable residing in the OS page cache and those that do not, due to a filter miss. We leave exploiting such smaller time differences to future work.)

C3(2) implies that generating keys uniformly at random will generate a false-positive key with hundreds to thousands of attempts, on average. It holds because in practice, filters are typically configured with small but non-negligible FPRs (e.g., 0.5%–5%), as negligibly small FPRs blow up the filter's memory consumption [65].[2]

5.3 Prefix siphoning template

Prefix siphoning consists of two phases. First, a preliminary phase learns to distinguish queries of negative and positive keys (§ 5.3.1). The second phase consists of multiple rounds, each of which extracts a key or key prefix (§ 5.3.2). Rounds are run concurrently (see § 9).

5.3.1 Learning to distinguish positive from negative keys

The attack starts with a preliminary phase that builds a distribution of query response times, which is used by the second phase to distinguish positive from negative keys.

The distribution is built by measuring response times of multiple *get()* requests for random keys. With large key spaces, such random keys are mostly negative keys, but a small (though non-negligible) fraction will be positive (due to C3). Such positive keys are overwhelmingly likely to be false positives, but that does not matter for this step, which is only concerned with distinguishing negative from positive keys, regardless of whether the positive output is correct.

The expected distribution observed is a bimodal distribution, with peaks corresponding to the average response time of negative and positive keys. From this distribution, the attacker can derive a cutoff value that likely distinguishes negative (fast) from a positive (slow) queries.

5.3.2 Extracting keys

This phase consists of multiple rounds, each of which extracts a key. Each round consists of three steps: ① finding a false-positive key κ, ② identifying the prefix that κ shares with some stored key k, and, when possible, ③ extending the prefix to extract k. Rounds are run concurrently (§ 9).

Step ① and ② simply invoke the attacker's *FindFPK* and *IdPrefix* algorithms, respectively. These steps are actually the "meat" of the attack, and we later describe their instantiations for SuRF (§ 6) and RocksDB's prefix Bloom filter (§ 7).

Whether step ③ is possible depends on the properties of the attacked system (and this is why it is not part of the vulnerable range filter characterization). If the system's query responses distinguish between failures due to target key absence and lack of authorization, then the attacker can extend the revealed prefix k' with some symbol sequence α and query for the key $k'\alpha$. The response will indicate lack of authorization if and only if $k'\alpha$ is a valid key. The attacker can thus iterate over all possible suffixes until k is found. Because k is not known to the attacker, they must first try all possible single symbol extensions, then all two symbol extensions, and so on. This process requires $O(|\Sigma|^{|k|-|k'|})$ queries, which can be several orders of magnitude less than a full-key brute force search. Crucially, step ③ only attempts to extend "long" prefixes, for which extension is feasible. Other prefixes are discarded.

Rationale for step decomposition For fixed-length keys, it might seem that the *IdPrefix* algorithm (step ②) for identifying the prefix is superfluous. After all, given that κ shares a prefix with some stored key k, the attacker can enumerate all possible suffixes from the end to the beginning, until identifying k. For example, suppose keys are 14-character strings and the attacker has found a false-positive key `manchestercars` because it shares the prefix `manchesterc` with the stored key `manchestercity`. Without knowing (or caring about) the shared prefix, the attacker can start querying for `manchestercara`, `manchestercarb`, ..., `manchestercaaa`, `manchestercaab`, and so on—all of which fail due to key absence—until reaching `manchestercity`, which will fail due to lack of authorization. As before, this process requires $O(|\Sigma|^{|k|-|k'|})$ queries and so it theoretically achieves the same results directly, without requiring an *IdPrefix* algorithm.

Why, then, is existence of an *IdPrefix* algorithm defined as one of the characteristics of a vulnerable filter? The answer is that without knowledge of the prefix, the attacker cannot efficiently schedule their work in step ③. They cannot distinguish a small suffix space (as in the example above) from

[1] Existence of *FindFPK* and *IdPrefix* is required in addition to C1 because a filter satisfying only C1 may not allow an attacker to extract the prefixes.

[2] Prefix siphoning can still be performed for exponentially low false positive rates, but its cost (in terms of number of queries needed) increases proportionally to the decrease in the false positive rate.

a huge space—e.g., if the false-positive key only shared the prefix `m` with `manchestercity`.

The *IdPrefix* algorithm protects us from the above pitfall. By identifying the shared prefix, it enables the attacker to decide whether to try and extend the prefix to a full key. Moreover, when multiple rounds execute concurrently, the attacker can collect many prefixes and then prioritize extending the longest ones.

6 SuRF prefix siphoning

Here, we instantiate a prefix siphoning attack against LSM-trees employing the SuRF [65] range filter. § 6.1 summarizes SuRF and § 6.2 shows that it is vulnerable to prefix siphoning.

6.1 SuRF primer

The succinct range filter (SuRF) [65] is the first proposed general range filter. Like the LSM-tree SSTables it approximates, SuRF is an immutable structure. SuRF can speed up LSM-tree range queries by 5×, but it imposes a modest cost on point queries due to having higher FPRs than a Bloom filter [65].

At a high level, SuRF is a pruned trie. A *trie* is a tree data structure that stores keys sorted according to the lexicographic order of Σ. Each edge is labeled with a symbol and each node corresponds to the concatenation of all edge labels on the path to that node. Each leaf thus corresponds to a key and each internal node to a key prefix (Figure 1(a)). An internal node can also correspond to a key (if the key set is not prefix-free), which is indicated by one of its fields. For space-efficiency, SuRF uses a succinct trie representation.

SuRF further saves space by *pruning* the trie. The basic SuRF variant (SuRF-Base) stores the minimum length key prefixes that uniquely identify each key, i.e., shared key prefixes plus the symbol following the shared prefix of each key (Figure 1(b)). SuRF's pruning results in a space-efficient but only approximate representation of the key set.

Both point and range queries are satisfied from the pruned trie structure. A *get*(k) returns true (possibly erroneously) if and only if the path induced by k terminates at a node associated with a key. For example, in Figure 1(b), `BLOOD` is a false positive. Range queries rely on the trie's ordered structure. For example, to check if the SuRF contains a key $k \in [a,b]$, the query finds the node corresponding to the smallest key $\geq a$. If it corresponds to a key $> b$, the query returns false; otherwise, it returns (possibly erroneously) true.

SuRF variants to reduce FPR SuRF-Base's FPR is data-dependent, i.e., depends on the key set. Compare, for example, two sets of 26 keys: $A = \{x\alpha | x \in A,\ldots,Z\}$ and $B = \{\alpha x | x \in A,\ldots,Z\}$, where α is some long string. For A, SuRF's FPR is nearly 100%, as any key except $A,\ldots,Z$ is a false positive. But for B, the FPR is extremely small, as only keys that begin with α pass the filter.

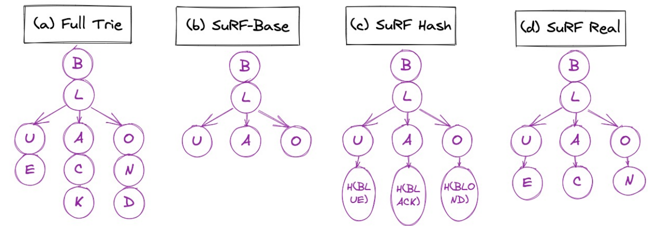


Figure 1: Trie and SuRF variants over the key set `BLUE`, `BLACK`, and `BLOND`. (Figure adapted from [65].)

To improve the FPR, SuRF offers variants that augment SuRF-Base's pruned structure with a few bits per leaf of information about the leaf's suffix. These bits reduce the FPR by allowing queries to reject keys that share a prefix with the stored key but have a different suffix, in exchange for increasing per-key memory consumption.

SuRF-Hash (Figure 1(c)) hashes the leaf's key and stores n bits from the hash value, where n is configurable. *SuRF-Real* (Figure 1(d)) stores the first m bits of the key's suffix, where m is configurable.

6.2 Vulnerability of SuRF

Every SuRF variant has the characteristics defined in § 5.2. C3(1) holds trivially. C3(2) holds empirically: SuRF-Base has an FPR of 4% for random 64-bit keys and SuRF-Hash reduce this FPRs to $\approx 0.1\%$ [65]. C1 holds because in every SuRF variant, *every* false-positive key κ shares a prefix with some stored k—C1 holds with probability 1.

To show that C2 holds, we describe how to efficiently find a false-positive key (§ 6.2.1) and how to identify the prefix that it shares with a stored key (§ 6.2.2). We assume the ability to check if a key is a filter positive or negative key based on measuring query response times. The implementation of this check is described in § 9.

6.2.1 Finding a false-positive key (FindFPK)

For SuRF, our *FindFPK* algorithm simply generates queries for uniformly random keys until it detects a positive response, based on the cutoff determined in the attack's preliminary learning phase (§ 5.3.1). Due to C3, this step is expected to terminate with a few hundreds to thousands of attempts.

We refer to the random positive key found as a false-positive key, because that is the overwhelmingly likely event. However, the attack still works if, unbeknownst to the attacker, the found key is actually a true positive key.

6.2.2 Identifying a shared prefix (IdPrefix)

For a false-positive key κ, let $k = k(\kappa)$ be the stored key whose shared prefix k' with κ is the longest among all stored keys. We write $\kappa = k'\alpha$ and $k = k'\beta$. Our algorithm will output k'.

SuRF-Base/Real To find k', we exploit SuRF's structure, namely that any key starting with a proper prefix of k' is a

negative key. Let $\kappa = \kappa_1 \dots \kappa_n$. We repeatedly remove the last symbol from the key, iteratively checking if the keys $\kappa_1 \dots \kappa_{n-1}, \kappa_1 \dots \kappa_{n-2}, \dots$ are negative or positive keys. These keys will be positive until we remove a symbol from k'. Thus, the key checked before a negative key is found is k'.

If the attacked system does not support variable-length keys, removing symbols is not possible. In this case, instead of removing symbols, we change them. We iteratively check if the keys $\kappa_1 \dots \kappa'_n, \kappa_1 \dots \kappa'_{n-1}\kappa_n \dots$ are negative or positive keys, where $\kappa'_i \neq \kappa_i$. Similarly to before, if the first negative key found is $\kappa_1 \dots \kappa'_j \dots \kappa'_n$ then $k' = \kappa_1 \dots \kappa_j$.

Overall, the number of requests made is $O(|\kappa|)$.

SuRF-Hash SuRF-Hash complicates the attack, because modifying κ's suffix can change its hash value, leading to a key that is rejected by SuRF despite sharing the prefix k'. To address this problem, we assume SuRF's hash function *hash* is public knowledge. (This is a reasonable assumption, because the hash function's purpose is to reduce the FPR and not for security.) We perform essentially the same algorithm(s) as for SuRF-Base/Real, but we only query each modified key κ' if $hash(\kappa') = hash(\kappa)$. We are still essentially assured to find keys to query, because SuRF-Hash stores only a small subset of the hash bits, for space-efficiency reasons. For example, with the recommended 4 hash bits [65] and using 8-bit symbols, on average 1 in 16 symbols tried will yield a hash collision and thus a key usable by the *IdPrefix* algorithm.

Similarly, when trying to extend an identified prefix to a full key (step ③ in § 5.3.2), we can skip querying any candidate key whose hash does not match the false-positive key's hash.

7 Prefix Bloom filter prefix siphoning

This section instantiates prefix siphoning against LSM-trees using the prefix Bloom filter (PBF) [25]. We describe the PBF in § 7.1 and show its vulnerability in § 7.2.

7.1 Prefix Bloom filter primer

The PBF is a Bloom filter-based range filter that supports range queries for ranges expressible as fixed-prefix queries. While PBFs do not provide general range queries, they are currently deployed in real-world key-value stores such as RocksDB [30] and LittleTable [52].

A PBF consists of a Bloom filter and a predetermined prefix length, l. When a key k is inserted into the PBF, both k and its l-bit prefix are inserted into the Bloom filter.

PBF range queries must be for ranges of the form "all keys starting with α," where α is an l-bit string. They are answered by querying the Bloom filter for α. If this query responds false, the dataset does not contain keys within the target range.

The PBF answers point queries by querying the Bloom filter for the queried key. We remark that if the high-level system does not prioritize point query efficiency, the PBF can be configured to only store key prefixes. In this case, the PBF implements a point query for key k by querying its Bloom filter for k's l-bit prefix. This option reduces the PBF's memory consumption but increases the FPR of point queries. This PBF configuration does not affect the success of our attack, so we do not discuss it further.

7.2 Vulnerability of the PBF

The PBF has the characteristics defined in § 5.2. As with SuRF, C3(1) holds trivially. C3(2) holds because the PBF's FPR is based on its Bloom filter's FPR.

The PBF has an important property: it not only has the usual Bloom filter false positives caused by hash collisions but also has what we call *prefix false positives*. These occur when a PBF point query falsely returns positive for an input κ that is an l-bit prefix of a dataset key, simply because the Bloom filter stores both dataset keys and their l-bit prefixes. This property implies that C1 holds: with probability $1 - FPR$, an l-bit false-positive is actually the prefix of some stored key.

To show that C2 holds, we need only describe how to find prefix false positives (§ 7.2.1). Finding them makes the *IdPrefix* algorithm of C2 trivial: given an l-bit false positive κ, it outputs κ.

7.2.1 Finding l-bit false-positive keys (FindFPK)

The *FindFPK* algorithm first determines the length of key prefixes stored in the PBF, l, and then proceeds to guess prefix false positives. Crucially, finding l needs to be performed only once per attack. That is, when running the attack's rounds concurrently (§ 9), we run this step only once.

Once l is known, generating queries for uniformly random l-bit strings will find false-positive keys, similarly to the SuRF attack's *FindFPK* (§ 6.2.1). Given a set of false positive l-bit keys thus found, an expected fraction of $p/2^l$ will be prefix false positives, where p is the number of distinct l-bit prefixes of dataset keys. The remaining false positives will be hash-collision Bloom filter false positives. Because we cannot distinguish between the two types of false positives, the attack's later steps must try to extend all of them to full keys.

The crux of the *FindFPK* algorithm is to identify l. To this end, we rely on the PBF property that made it vulnerable in the first place. For any prefix length $l' \neq l$, the probability of an l'-bit key being a false positive is exactly the filter's FPR. Only for l-bit keys will we observe a "bump" in the probability of a random l-bit key being a false positive, due to the presence of prefix false positives.

Accordingly, the *FindFPK* algorithm first generates j queries for uniformly random keys of length l', for every non-trivial prefix length l' (e.g., $l' \geq 3$). It observes the fraction of false positives found and deduces that l is the length l' for which the fraction of false positives found is maximal.

8 Complexity analysis

The key factor determining prefix siphoning's effectiveness is the probability of *FindFPK* (step ① in § 5.3.2) guessing an *exploitable* key k, which is a false positive whose longest

shared prefix with stored keys is of length l, where l is a predetermined constant for which extending k into a full key is feasible (step ③).

The full version of this paper [39] includes a theoretical analysis of the SuRF and PBF attacks, which is omitted here due to space constraints. We analyze the case of uniformly random keys, which is the worst case for our attack. (If the key distribution is skewed, then (1) the guessing and full-key extraction steps can incorporate this knowledge; and (2) the prefixes SuRF stores are longer, so our attack will identify longer prefixes and thus extend them to full keys faster.)

The analysis derives the probability of *FindFPK* guessing an exploitable key. This determines the expected number of queries to guess an exploitable key or, equivalently, the number of keys we ultimately expect to extract after investing G guesses in *FindFPK*. These values also allow comparing the cost (in queries) of prefix siphoning to brute force search.

Under the realistic constraint that $|D| \ll 2^l$, where D is the dataset (e.g., $|D| = 500$ M and $l = 40$), we find that (1) prefix siphoning becomes more effective with growth in dataset size and better FPR—i.e., as the LSM-tree becomes more effective, so does prefix siphoning; and (2) prefix siphoning takes several orders of magnitude fewer queries to extract a key than an exhaustive brute force search.

9 Implementation issues

In previous sections, we assume the attacker can check if a key is a filter positive or negative key, based on measuring query response times. Here, we describe our implementation of this check.

The basic idea is simple. Prefix siphoning's preliminary phase (§ 5.3.1) derives a response time cutoff. Keys whose query response time is below this cutoff are considered negative; otherwise, they are considered positive. However, this cutoff only distinguishes between queries satisfied from memory and those involving I/Os. Once a query for a false-positive key completes, the I/O it performs reads the relevant SSTable into the in-memory page cache. Future queries for false-positive keys covered by this SSTable will thus get satisfied from memory.

To overcome this problem, we exploit the fact that the attack targets some production system, which is assumed to sustain heavy I/O load due its legitimate operation. This property implies that if the attacker *waits* after performing a false positive query, the SSTable brought in will be evicted from the page cache due to legitimate I/O traffic.

Unfortunately, waiting for even a few seconds after every query would make the attack impractical. We solve this challenge by performing attack rounds in a *concurrent, breadth-first* manner, as described below, instead of working depth-first (finding a false-positive key and proceeding to identify its prefix and then to extract the full key).

Step ① of § 5.3.2 (*FindFPK* execution) generates N random keys (false positive candidates) and measures a four-query average response time for each key to identify false-positive keys. The averages are computed in a breadth-first manner: there are four iterations, each of which performs one query for each key. Waiting for page cache evictions is done only between each iteration.

Step ② (*IdPrefix*) similarly executes iteratively, interleaving the next step of *IdPrefix* for each false-positive key in each iteration, until all invocations output a prefix. Again, waiting for page cache evictions is only done between iterations.

Step ③ (key extraction) likewise interleaves the searches extending each prefix. We optimize step ③'s general-case brute force suffix search by leveraging the fact that step ② outputs a set of prefixes. This enables us to discard short prefixes, so that step ③ only attempts to extend prefixes where the suffix search is feasible.

The interleaved execution of each step can be sped up using multi-core parallelization by assigning each core a subset of the N random keys, false-positive keys, or prefixes when executing step ①, ②, and ③, respectively, in the above described manner. This results in linear speedup (in the number of cores) of step execution time. Our implementation parallelizes step ③, whose execution time dominates the attack (§ 10.2.2), over 16 cores and leaves the other steps single-threaded.

10 Evaluation

In this section, we evaluate prefix siphoning attacks on SuRF and PBF in RocksDB. We demonstrate the attack's feasibility, successfully mounting it against a full-fledged RocksDB key-value store employing SuRF (§ 10.2).[3] We empirically analyze the SuRF attack's efficiency and sensitivity to data store size and filter FPR (§ 10.3). Consistent with our theoretical analysis, we find that the attack becomes more effective with growth in dataset size and better FPR—i.e., as the LSM-tree becomes more effective, so does prefix siphoning. Finally, we demonstrate the attack against the PBF (§ 10.4).

10.1 Experimental setup

Both clients and the attacked key-value store run on the same server. However, the time differences we exploit can be measured over the network using prior techniques (see § 4).

We use a server with two Intel Xeon Gold 6132 v6 (Skylake) processors, each of which has 14 2.6 GHz cores with two hyperthreads per core. The server is equipped with 192 GB DDR4 DRAM and two 0.5 TB NVMe SSDs. The server runs Ubuntu 18.04 and code is compiled with GCC 4.8.

RocksDB setup We use a version of RocksDB modified by the SuRF authors to employ SuRF [65]. The target RocksDB instance uses the NVMe devices as secondary storage. We use Linux `cgroup` to limit RocksDB's available DRAM to

[3] We use the SuRF's authors' implementation, https://github.com/efficient/SuRF.

2 GB. This configuration emulates an industrial-scale, I/O heavy key-value store setup, in which storage capacity far exceeds DRAM capacity.

The RocksDB engine stores 64-bit keys and 1000-byte values and the SuRF-Real variant. Unless noted otherwise, we use a datastore of 50 M uniformly random keys (generated using SHA1). We invoke RocksDB LSM-tree compaction after populating the datastore. We do this to emulate the compaction that naturally occurs in a real workload due to insertions, because our experiments perform only *get()*s.

Background load In all experiments, we emulate a realistic, loaded system by running 32 threads that constantly perform *get()* queries for random keys, with 50% of the queries targeting stored keys and 50% targeting non-present keys.

10.2 RocksDB+SuRF-Real key extraction

We implement the attack as described in § 9. § 10.2.1 evaluates the attack's first phase (§ 5.3.1), demonstrating that query response times can be used to distinguish negative from positive keys in practice, even in the presence of heavy background load. § 10.2.2 evaluates the attack's second phase, which extracts full keys, and compares it to a brute force search.

10.2.1 Negative/positive query time differences

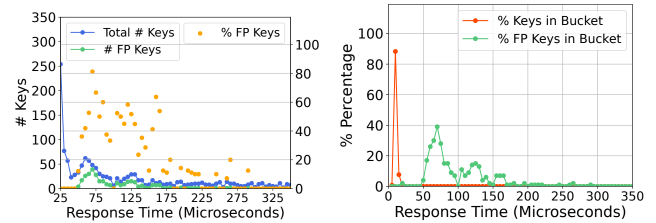
In this phase, the attacker performs 10 M *get()* queries for randomly generated keys to build the response time distribution. Table 1 shows the distribution of response times in terms of 5 microsecond buckets. The distribution is extremely skewed toward values $< 25\,\mu$s, which our attack therefore assumes are associated with negative keys.

To validate this assumption, Figure 2 visualizes the distribution while breaking the response times by queried key type (negative or false-positive). This breakdown is presented for analysis purposes; it is not available to the attacker. For readability, we present the breakdown in two ways. Figure 2(a) shows only the buckets $\geq 25\,\mu$s, which are otherwise dwarfed by the lower end of the distribution. We show both the number of keys (blue) and false-positive (green) keys in each bucket, and the percent of false-positive keys in each bucket (orange). Figure 2(b) shows the entire distribution, but bucket sizes (Y axis) are percentages instead of absolutes. For each bucket, we report the number of keys in the bucket as well as the percentage of false positives (out of all positives).

Figure 2(a) shows that the vast majority of false positive queries have a response time of 25–35 μs. Conversely, Figure 2(b) shows that this response time range contains over 50% of the false-positive keys. Overall, these results show that picking a cutoff point of 25 μs for distinguishing a negative from positive key—which is done based only on the distribution's shape, without knowledge of key types—yields a good distinguisher.

Bucket range (microseconds)	**% of responses**
< 5	0.77%
5 - 10	**88.3%**
10 – 15	7.65%
15 – 20	0.53%
20 – 25	0.05%
≥ 25	2.7%

Table 1: Distribution of query response times.



(a) Buckets $\geq 25\,\mu$s: Absolute number of queried keys

(b) All buckets: Percentage of queried keys.

Figure 2: Breakdown of query response time distribution.

10.2.2 Key extraction

The attack executes as described in § 9; specifically, wait is set to 20 seconds and each step is executed in a parallel, breadth-first manner, to minimize the amount of time spent waiting for page cache evictions. The attacker generates a set of 10 M random keys to find false-positive keys (step ① of § 5.3.2). The attacker next identifies the prefix each false-positive key shares with a stored key (step ②). Finally, the attacker discards every prefix of length < 40 bits and attempts to extend the remaining prefixes into full keys (step ③).

Figure 3 shows the number of keys extracted as a function of the number of total number of *get()* requests issued by the attack (aggregated over steps ①–③). The figure also compares the attack to an *idealized* attack, which uses internal RocksDB debugging counters to accurately determine the filters' responses for each queried key, instead of relying on query response times.

Because the idealized attack never incorrectly classifies a key, it identifies more false positives than the actual attack in step ①. It thus requires more queries in step ② to identify the shared prefixes of the keys provided to step ②, as there are more of them. Consequently, the idealized attack begins step ③ later (in terms of queries) than the actual attack, which is why its line is "shifted" compared to actual attack. For this reason, the idealized attack also requires more queries overall. Ultimately, however, the actual attack extracts only 74 fewer keys than the idealized version.

The idealized attack is also faster (in real time) than the actual attack, because it does not require waiting for page cache evictions. The actual attack's key extraction rate is ≈ 10 minutes/key, while the idealized attack achieves 0.2 minutes/key.

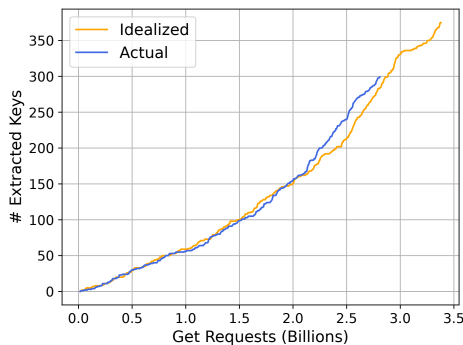


Figure 3: Actual vs. idealized prefix siphoning against SuRF-Real: Number of keys extracted as attack progresses.

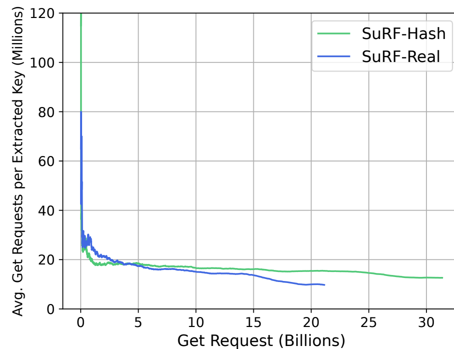


Figure 4: SuRF-Hash vs. SuRF-Real: Moving average of queries per extracted key as a function of attack progress (measured in queries).

attack step	# queries (millions)	queries/total (%)
① Find false positives	10M	0.35%
② Identify prefixes	0.025M	0.0009%
③ Extract keys	2581M	91.68%
Wasted	224M	7.9%

Table 2: Attack queries per stage. Wasted queries futilely attempt to extend an incorrectly identified prefix into a full key.

Table 2 shows a breakdown of the (actual) attack's queries across all three steps. The bulk of the attack is spent on step ③, extending prefixes into full keys. Our later analysis (§ 10.3.2) explains this number. The table also reports wasted queries, which are issued when the attack futilely tries to extract a key from an incorrect prefix, which was misidentified due to incorrectly classifying a key as a false-positive (based on its query response time). Additional wasted queries (not shown) are spent identifying prefixes of length < 40 bits in steps ①–②, which are then discarded. While over 90% of prefixes identified by steps ①–② are discarded, this waste is negligible, as they are discarded before the most expensive step.

Comparison to brute force We further evaluate a brute force attack, that randomly guesses keys until a stored key is found. We allow this attack to run for 10× more time than the prefix siphoning experiment—but it fails to guess a key. Unsurprisingly, brute force search for a large key space is infeasible.

SuRF-Hash vs. SuRF-Real SuRF-Hash complicates the attack. Compared to SuRF-Real with the same per-key space budget, SuRF-Hash replaces key bits (SuRF-Real's suffix bits) with hash value bits. This means that possible prefixes to identify are shorter and that the filter's FPR is lower, making the number of false positives identified in step ② lower. On the other hand, as discussed in § 6.2.2, when identifying the prefixes and performing key extraction, the attacker can use the false-positive key's hash value to ignore definitely incorrect guess—potentially improving the attack's efficiency.

To evaluate this trade-off, we compare idealized attacks against the same dataset, with RocksDB using either SuRF-Real with 8-bit suffixes or SuRF-Hash with 8-bit hashes. Thus, in SuRF-Hash, the suffix search space when extracting a key 256× larger than in SuRF-Real, but the attacker will ignore 255/256 of its guesses on average. To compensate for SuRF-Hash's lower FPR, the initial false-positive key search of the SuRF-Hash attack uses 3× the number of candidate keys used for SuRF-Real. Figure 4 therefore compares the attacks' amortized cost, in terms of a moving average of queries per extracted key as a function of attack progress.[4] The SuRF-Hash attack's extra initial queries (for finding false positives) manifest as the peak of the per-key cost, when all these extra queries are amortized across only a handful of keys. The extra cost is eventually amortized away, into a per-key cost of 12 M vs. 10 M queries for SuRF-Hash vs. SuRF-Real, respectively. For this similar cost, the SuRF-Hash attack extracts 2490 keys vs. 2171 keys for the SuRF-Real attack.

10.3 Attack analysis

This section analyzes the attack's efficiency (§ 10.3.1) and sensitivity to data store size (§ 10.3.2) and filter FPR (§ 10.3.3).

10.3.1 Efficiency

Figure 5 shows the attack's efficiency, measured as average *get()*s per extracted key as a function of attack progress. We compare across three 50 M random 64-bit key sets to show the results are not a function of the specific key set.

The average number of queries per extracted key converges to about $9\,\text{M} \approx 2^{23}$. This indicates that the attack extracts keys with roughly the work required to search a 23-bit space—40992× better than a brute force search of the full key space ($2^{64}/50\,\text{M} \approx 2^{38.4}$). The attack also extracts a substantial number of keys (375, 419, and 423 keys).

[4] I.e., the Y axis reports the number of *get()*s issued divided by the number of keys extracted up to the current X-axis point.

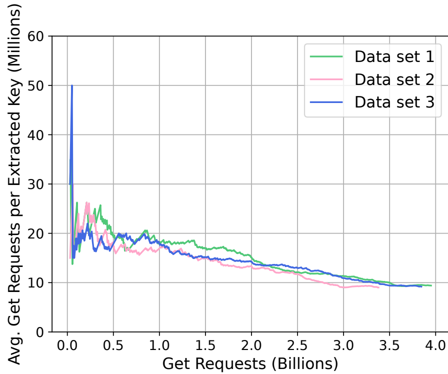


Figure 5: Attack efficiency: average number of *get()*s per extracted key as attack progresses.

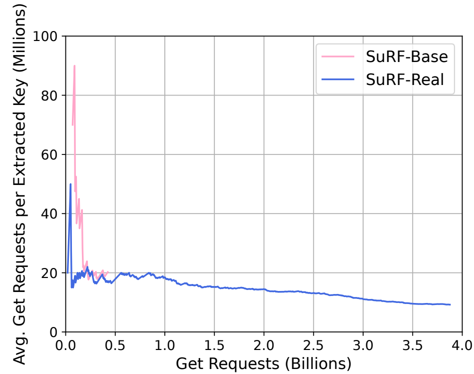


Figure 7: SuRF-Real vs. SuRF-Base: Moving average of queries per extracted key as a function of attack progress (measured in queries).

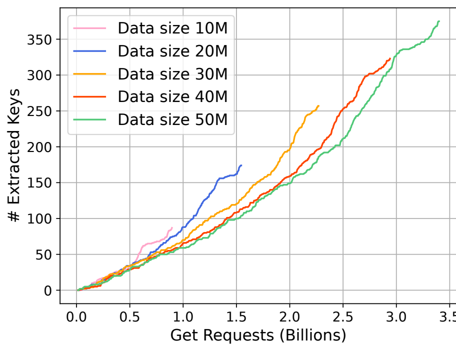


Figure 6: Idealized attack against SuRF-Real: Number of keys extracted for different dataset sizes.

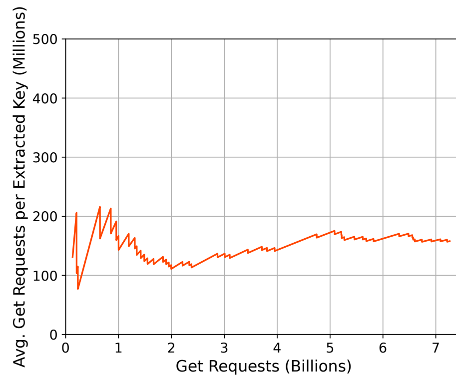


Figure 8: Idealized prefix siphoning against PBF ($l = 40$ bits).

10.3.2 Sensitivity to dataset size

To evaluate the attack's sensitivity to the dataset size, we progressively shrink our original 50 M key set into smaller subsets of size $c \cdot 10$ M keys for $c \in [1,5]$. We then perform an idealized attack against the system with each dataset, but using the same set of random keys for step ①, so any difference in attack behavior can related only to the datastore size and not the key distribution.

Figure 6 shows the number of keys extracted as the attack progresses. Prefix siphoning is more effective as the dataset size increases: it extracts ≈ 100 keys from the 10 M dataset, but almost 400 keys from the 50 M dataset.

10.3.3 Sensitivity to SuRF FPR

We show that prefix siphoning becomes more effective as SuRF's FPR improves, i.e., the attack becomes more harmful to the system as SuRF becomes more productive to the system. To demonstrate this effect, we compare idealized attacks against the same dataset, with RocksDB using either SuRF-Base or SuRF-Real. SuRF-Base stores shared key prefixes, padded to the next full byte (which adds 1–8 bits to the prefix). SuRF-Real does the same, plus stores a byte from the key's unique suffix, and thereby improves its FPR (see § 6.1).

We carry out the attacks against each SuRF variant using the same initial random key set, used to identify false-positive keys. Figure 7 reports the attack's amortized cost (queries per extracted key) as the attack progresses.

In both cases, the attack has similar efficiency of ≈ 10 M queries per extracted key, as evident from the similar slope of the two lines. However, the attack is more successful against SuRF-Real, where it extracts 420 keys, than against SuRF-Base, where is extracts 21 keys. The reason for the improved effectiveness is that SuRF-Real's extra key byte storage makes an initial false-positive key much more likely to have a prefix length of > 40 bits, resulting in more false positives making it to step ③.

The situation is similar with SuRF-Hash, which further improves the FPR over SuRF-Real (Figure 4). As mentioned in § 10.2.2, the idealized SuRF-Hash attack extracts 2490 keys vs. 2171 keys for the idealized SuRF-Real attack.

10.4 RocksDB+PBF key extraction

We evaluate an idealized prefix siphoning attack against RocksDB's PBF. We use a dataset of 50 M uniformly random 64-bit keys. We configure the PBF to store prefixes of length

$l = 40$ bits and to consume 18 bits/key (which is roughly the space usage of SuRF in our experiments).

Step ① (*FindFPK*) perform 1 M queries for uniformly random 40-bit keys, which result in 457 false-positive keys. The attack then attempts to extend these false positives into full keys. It eventually extracts 46 keys, which matches the expected number of prefix false positives observed in 1 M random guesses ($1\text{M} \cdot 50\text{M}/2^{40} = 45.4$). Figure 8 plots the attack's amortized cost (queries per extracted key) as the attack progresses. The PBF attack makes 160 M queries per extracted key, which is 20× more queries/key than the SuRF attack, but still three orders of magnitude better than a brute force search. The reason for this difference is that the PBF attack wastes effort trying to extend Bloom filter false positives that are not prefix false positives.

11 Mitigation

Here, we discuss approaches for mitigating prefix siphoning attacks. Unfortunately, every potential solution constitutes some trade-off, whether in query performance, memory efficiency, complexity, or other system aspects.

System-level approaches A system can block prefix siphoning attacks by only querying its key-value storage engine for keys the requesting user is allowed to access. This approach requires re-architecting the system so that a key's ACL is kept outside of the key-value store. In addition, a system can rate limit user requests, thereby slowing down prefix siphoning attacks. This approach is viable only if the system is not meant to handle a high rate of normal, benign requests.

Key-value store mitigation A key-value engine can block prefix siphoning by maintaining separate filters for point and range queries for each SSTable file. Unfortunately, this approach will double filter memory consumption. In addition, it will not block attacks that target range queries (which we believe are possible, and are currently exploring).

Filter-level mitigation A natural mitigation is for key-value stores to employ non-vulnerable range filters. Like the separate filter approach described above, this mitigation carries the risk of being vulnerable to future extensions of prefix siphoning to range queries.

In addition, the properties that make a range filter non-vulnerable to point query-based prefix siphoning may limit its utility in practice. For example, Rosetta (Robust Space-Time Optimized Range Filter) [45] is a range filter that does not conform to our vulnerable range filter characterization (§ 5.2), but it lacks support for variable-length keys, which are important in practice.

Rosetta uses Bloom filters for SuRF-like prefix-based filtering. Rosetta assumes a bound on the possible key length in bits, L. A Rosetta instance consists of L Bloom filters, $B_1, \dots, B_L$. When a key k is inserted into the filter, each i-bit prefix is inserted into the i-th Bloom filter B_i. A Rosetta point query thus simply queries B_L, making Rosetta non-vulnerable to prefix siphoning.

The Rosetta paper does not specify how variable-length keys are handled. Its design is clearly incompatible with such keys if there is no predetermined bound on their size. Even if such a bound exists (and can thus be used for L), Rosetta requires every key to be padded to L bits, so that point queries function correctly. This requirement significantly increases the filter's memory consumption.

Encrypted key-value stores Disclosed keys reveal no sensitive information if they are stored encrypted in the storage engine. However, encrypting key-value pairs requires re-architecting the entire system so it can query on encrypted data [63, 64]. Most if not all deployed key-value stores do not support such encryption.

12 Related Work

Key-value store timing attacks Existing key-value store timing attacks aim to disclose stored values. These attacks work by exploiting external mechanisms such as memory deduplication [55] or memory compression [56], which can be disabled for protection. In contrast, prefix siphoning exploits a mechanism of the key-value store itself, which cannot be disabled for protection without suffering significant throughput degradation and additional I/O traffic.

Storage engine timing attacks Timing attacks mostly target cryptographic software rather than storage engines. Futoransky et al. [35] extract private keys from a MySQL database with a timing attack, but the attack relies on insertions of attacker-chosen data. Wang et al. [59] show a practical timing attack on a multi-user search system, such as Elasticsearch.

13 Conclusion

This paper shows that certain range filters make LSM-trees vulnerable to novel *prefix siphoning* timing attacks, which exploit differences in query response times to reveal keys and prefixes of keys stored in the LSM-tree. Our results show that key-value store performance improvements may trade security in exchange, and encourage practitioners and researchers to evaluate the security impact of their work. We also hope that our characterization of vulnerable range filters will spur research on more secure filters.

Acknowledgments

We extend our deepest thanks to Yuvraj Patel, the paper's shepherd, and the anonymous reviewers for their dedication and assistance in improving this paper and their valuable feedback. We thank Guy Khazma for his work on an earlier stage of this project.

References

[1] CEPH. https://github.com/ceph/ceph.

[2] MongoDB. https://www.mongodb.com/.

[3] MySQL Server. https://github.com/mysql/mysql-server.

[4] Amazon. Amazon S3. https://aws.amazon.com/s3/, 2020.

[5] David G. Andersen, Jason Franklin, Michael Kaminsky, Amar Phanishayee, Lawrence Tan, and Vijay Vasudevan. FAWN: A Fast Array of Wimpy Nodes. In *SOSP*, 2009.

[6] Apache. Apache Flink — Stateful Computations over Data Streams. https://flink.apache.org, 2022.

[7] Mikhail Bautin. How We Built a High Performance Document Store on RocksDB? https://www.yugabyte.com/blog/how-we-built-a-high-performance-document-store-on-rocksdb/, 2019.

[8] Michael A. Bender, Martin Farach-Colton, Rob Johnson, Russell Kraner, Bradley C. Kuszmaul, Dzejla Medjedovic, Pablo Montes, Pradeep Shetty, Richard P. Spillane, and Erez Zadok. Don't Thrash: How to Cache Your Hash on Flash. In *VLDB*, 2012.

[9] Peter Benjamin. s3-fuzzer. https://github.com/pbnj/s3-fuzzer, 2017.

[10] Burton H. Bloom. Space / Time Trade-offs in Hash Coding with Allowable Errors. *CACM*, 13(7), 1970.

[11] David Brumley and Dan Boneh. Remote Timing Attacks are Practical. In *USENIX Security Symposium*, 2003.

[12] Zhichao Cao, Siying Dong, Sagar Vemuri, and David H.C. Du. Characterizing, Modeling, and Benchmarking RocksDB Key-Value Workloads at Facebook. In *FAST*, 2020.

[13] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. Bigtable: A Distributed Storage System for Structured Data. *ACM TOCS*, 26(2), 2008.

[14] Guoqiang Jerry Chen, Janet L. Wiener, Shridhar Iyer, Anshul Jaiswal, Ran Lei, Nikhil Simha, Wei Wang, Kevin Wilfong, Tim Williamson, and Serhat Yilmaz. Realtime Data Processing at Facebook. In *SIGMOD*, 2016.

[15] Alex Conway, Martín Farach-Colton, and Philip Shilane. Optimal Hashing in External Memory. In *ICALP*, 2018.

[16] Alexander Conway, Abhishek Gupta, Vijay Chidambaram, Martin Farach-Colton, Richard Spillane, Amy Tai, and Rob Johnson. SplinterDB: Closing the Bandwidth Gap for NVMe Key-Value Stores. In *USENIX ATC*, 2020.

[17] Scott A. Crosby, Dan S. Wallach, and Rudolf H. Riedi. Opportunities and Limits of Remote Timing Attacks. *ACM Transactions on Information and System Security*, 12(3), 2009.

[18] Niv Dayan, Manos Athanassoulis, and Stratos Idreos. Monkey: Optimal Navigable Key-Value Store. In *SIGMOD*, 2017.

[19] Niv Dayan, Manos Athanassoulis, and Stratos Idreos. Optimal Bloom Filters and Adaptive Merging for LSM-Trees. *ACM TODS*, 43(4), 2018.

[20] Niv Dayan and Stratos Idreos. Dostoevsky: Better Space-Time Trade-Offs for LSM-Tree Based Key-Value Stores via Adaptive Removal of Superfluous Merging. In *SIGMOD*, 2018.

[21] Niv Dayan and Stratos Idreos. The Log-Structured Merge-Bush & the Wacky Continuum. In *SIGMOD*, 2019.

[22] Niv Dayan and Moshe Twitto. Chucky: A Succinct Cuckoo Filter for LSM-Tree. In *SIGMOD*, 2021.

[23] Tom de Vries. Teh s3 bucketeers. https://github.com/tomdev/teh_s3_bucketeers/, 2021.

[24] Biplob Debnath, Sudipta Sengupta, and Jin Li. SkimpyStash: RAM Space Skimpy Key-Value Store on Flash-Based Storage. In *SIGMOD*, 2011.

[25] Siying Dong, Mark Callaghan, Leonidas Galanis, Dhruba Borthakur, Tony Savor, and Michael Stumm. Optimizing Space Amplification in RocksDB. In *CIDR*, 2017.

[26] Phil Eaton. What's the big deal about key-value databases like FoundationDB and RocksDB? https://notes.eatonphil.com/whats-the-big-deal-about-key-value-databases.html, 2022.

[27] Nathan Eddy. Cloud Misconfig Exposes 3TB of Sensitive Airport Data in Amazon S3 Bucket: 'Lives at Stake'. https://www.darkreading.com/application-security/cloud-misconfig-exposes-3tb-sensitive-airport-data-amazon-s3-bucket, 2022.

[28] David Eisenstat. Structured data encoding in CockroachDB SQL. https://github.com/cockroachdb/

cockroach/blob/master/docs/tech-notes/encoding.md, 2021.

[29] Robert Escriva, Bernard Wong, and Emin Gün Sirer. HyperDex: A Distributed, Searchable Key-Value Store. In *SIGCOMM*, 2012.

[30] Facebook. RocksDB. https://github.com/facebook/rocksdb.

[31] Facebook. MyRocks. http://myrocks.io/, 2015.

[32] Facebook. MyRocks record format. https://github.com/facebook/mysql-5.6/wiki/MyRocks-record-format, 2019.

[33] Bin Fan, David G. Andersen, Michael Kaminsky, and Michael Mitzenmacher. Cuckoo Filter: Practically Better Than Bloom. In *CoNEXT*, 2014.

[34] Roy Thomas Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine, 2000.

[35] Ariel Futoransky, Damián Saura, and Ariel Waissbein. The ND2DB Attack: Database Content Extraction Using Timing Attacks on the Indexing Algorithms. In *WOOT*, 2007.

[36] Tom Van Goethem, Christina Pöpper, Wouter Joosen, and Mathy Vanhoef. Timeless Timing Attacks: Exploiting Concurrency to Leak Secrets over Remote Connections. In *USENIX Security Symposium*, 2020.

[37] Guy Golan-Gueta, Edward Bortnikov, Eshcar Hillel, and Idit Keidar. Scaling Concurrent Log-Structured Data Stores. In *EuroSys*, 2015.

[38] Google. LevelDB. https://github.com/google/leveldb.

[39] Adi Kaufman, Moshik Hershcovitch, and Adam Morrison. Prefix Siphoning: Exploiting LSM-Tree Range Filters For Information Disclosure (Full Version). *arXiv e-prints*, abs/2306.04602, 2023.

[40] Redis Lab. Redis. https://github.com/redis/redis.

[41] Cockroach Labs. CockroachDB. https://www.cockroachlabs.com/, 2022.

[42] Avinash Lakshman and Prashant Malik. Cassandra: A Decentralized Structured Storage System. *SIGOPS Operating Systems Review*, 44(2), 2010.

[43] Hyeontaek Lim, Bin Fan, David G. Andersen, and Michael Kaminsky. SILT: A Memory-efficient, High-performance Key-value Store. In *SOSP*, 2011.

[44] LinkedIn. FollowFeed: LinkedIn's Feed Made Faster and Smarter. https://engineering.linkedin.com/blog/2016/03/followfeed--linkedin-s-feed-made-faster-and-smarter, 2016.

[45] Siqiang Luo, Subarna Chatterjee, Rafael Ketsetsidis, Niv Dayan, Wilson Qin, and Stratos Idreos. Rosetta: A Robust Space-Time Optimized Range Filter for Key-Value Stores. In *SIGMOD*, 2020.

[46] Netflix. Application Data Caching using SSDs. http://techblog.netflix.com/2016/05/application-data-caching-using-ssds.html, 2016.

[47] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, and Venkateshwaran Venkataramani. Scaling Memcache at Facebook. In *NSDI*, 2013.

[48] OWASP. Insecure Direct Object Reference Prevention Cheat Sheet. https://cheatsheetseries.owasp.org/cheatsheets/Insecure_Direct_Object_Reference_Prevention_Cheat_Sheet.html, 2021.

[49] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. The Log-Structured Merge-Tree (LSM-Tree). *Acta Informatica*, 33(4):351–385, 1996.

[50] Jordan Potti. Awsbucketdump. https://github.com/jordanpotti/AWSBucketDump, 2018.

[51] Pandian Raju, Rohan Kadekodi, Vijay Chidambaram, and Ittai Abraham. PebblesDB: Building Key-Value Stores Using Fragmented Log-Structured Merge Trees. In *SOSP*, 2017.

[52] Sean Rhea, Eric Wang, Edmund Wong, Ethan Atkins, and Nat Storer. LittleTable: A Time-Series Database and Its Uses. In *SIGMOD*, 2017.

[53] Dan Salmon. S3scanner. https://github.com/sa7mon/S3Scanner, 2022.

[54] Apache Samza. State Management. http://samza.apache.org/learn/documentation/0.8/container/state-management.html, 2017.

[55] Martin Schwarzl, Erik Kraft, Moritz Lipp, and Daniel Gruss. Remote Memory-Deduplication Attacks. In *NDSS*, 2022.

[56] Martin Schwarzl, Pietro Borrello Gururaj Saileshwar, Hanna Müller, Michael Schwarz, and Daniel Gruss. Practical Timing Side-Channel Attacks on Memory Compression. In *IEEE S&P*, 2023.

[57] Russell Sears and Raghu Ramakrishnan. BLSM: A General Purpose Log Structured Merge Tree. In *SIGMOD*, 2012.

[58] Mark Sutherland, Babak Falsafi, and Alexandros Daglis. Cooperative Concurrency Control for Write-Intensive Key-Value Workloads. In *ASPLOS*, 2023.

[59] Liang Wang, Paul Grubbs, Jiahui Lu, Vincent Bindschaedler, David Cash, and Thomas Ristenpart. Side-Channel Attacks on Shared Search Indexes. In *IEEE S&P*, 2017.

[60] Peng Wang, Guangyu Sun, Song Jiang, Jian Ouyang, Shiding Lin, Chen Zhang, and Jason Cong. An Efficient Design and Implementation of LSM-Tree Based Key-Value Store on Open-Channel SSD. In *EuroSys*, 2014.

[61] Brian Warehime. insp3ctor. https://github.com/brianwarehime/inSp3ctor, 2018.

[62] Ian Williams. Bucket finder. https://github.com/FishermansEnemy/bucket_finder, 2013.

[63] Xingliang Yuan, Yu Guo, Xinyu Wang, Cong Wang, Baochun Li, and Xiaohua Jia. EncKV: An Encrypted Key-Value Store with Rich Queries. In *ASIA CCS*, 2017.

[64] Xingliang Yuan, Xinyu Wang, Cong Wang, Chen Qian, and Jianxiong Lin. Building an Encrypted, Distributed, and Searchable Key-Value Store. In *ASIA CCS*, 2016.

[65] Huanchen Zhang, Hyeontaek Lim, Viktor Leis, David G. Andersen, Michael Kaminsky, Kimberly Keeton, and Andrew Pavlo. SuRF: Practical Range Query Filtering with Fast Succinct Tries. In *SIGMOD*, 2018.