# Prefix Siphoning:

## Exploiting LSM-Tree Range Filters For Information Disclosure

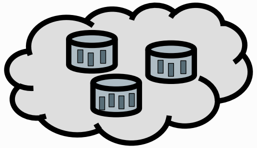
Adi Kaufman,[1] **Moshik Hershcovitch**,[1,2] Adam Morrison[1]

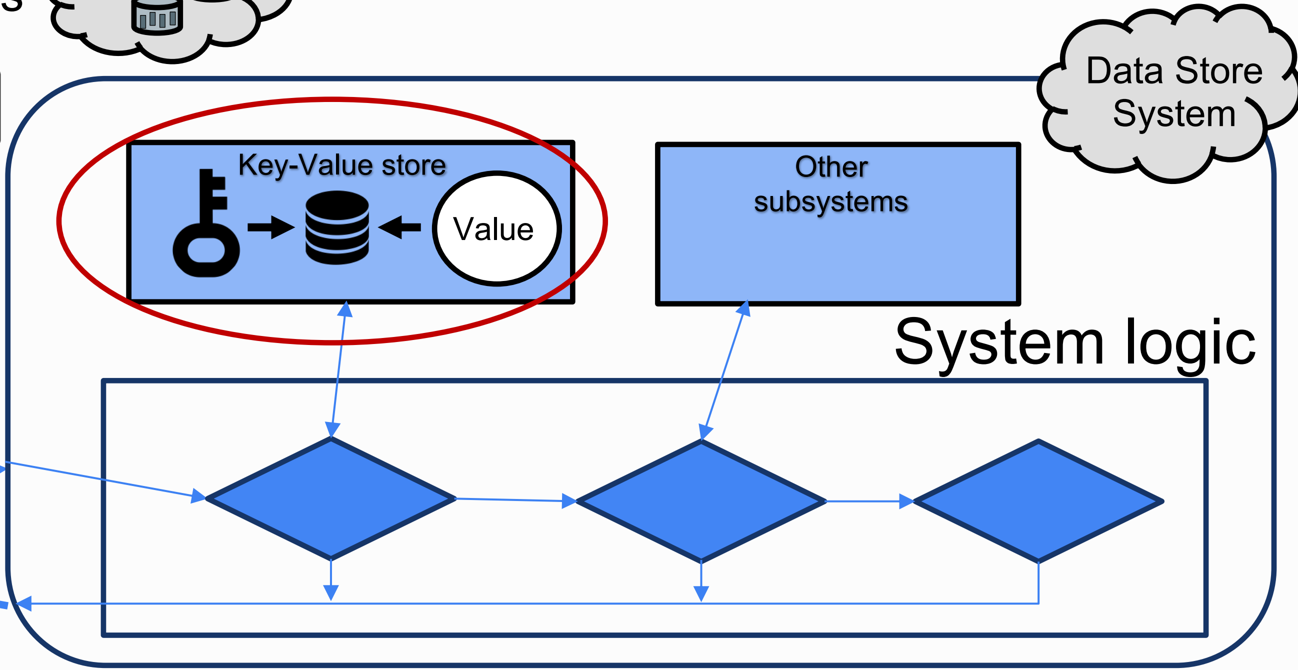
[1] TEL AVIV UNIVERSITY
Pursuing the Unknown

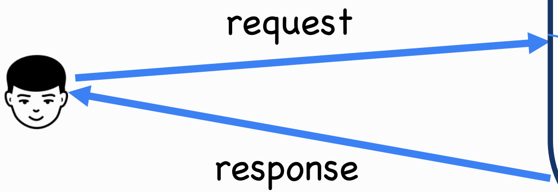
[2] IBM **Research**

# Key value storage engines

Key-value stores serve as storage engines for many data store systems, like:

- Object storage systems 
- Database systems 
- Storage Systems 



# Key-value store abstraction

A key-value store exposes a dictionary-like abstraction:

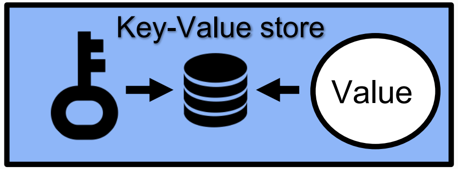
*Put(key,value)*

<“ATC_2023”: “BOSTON”>

*Get(key)*
*(point query)*

<“ATC_2023”>

<“ATC_2023”: “BOSTON”>




*RangeQ(key1,key2)*

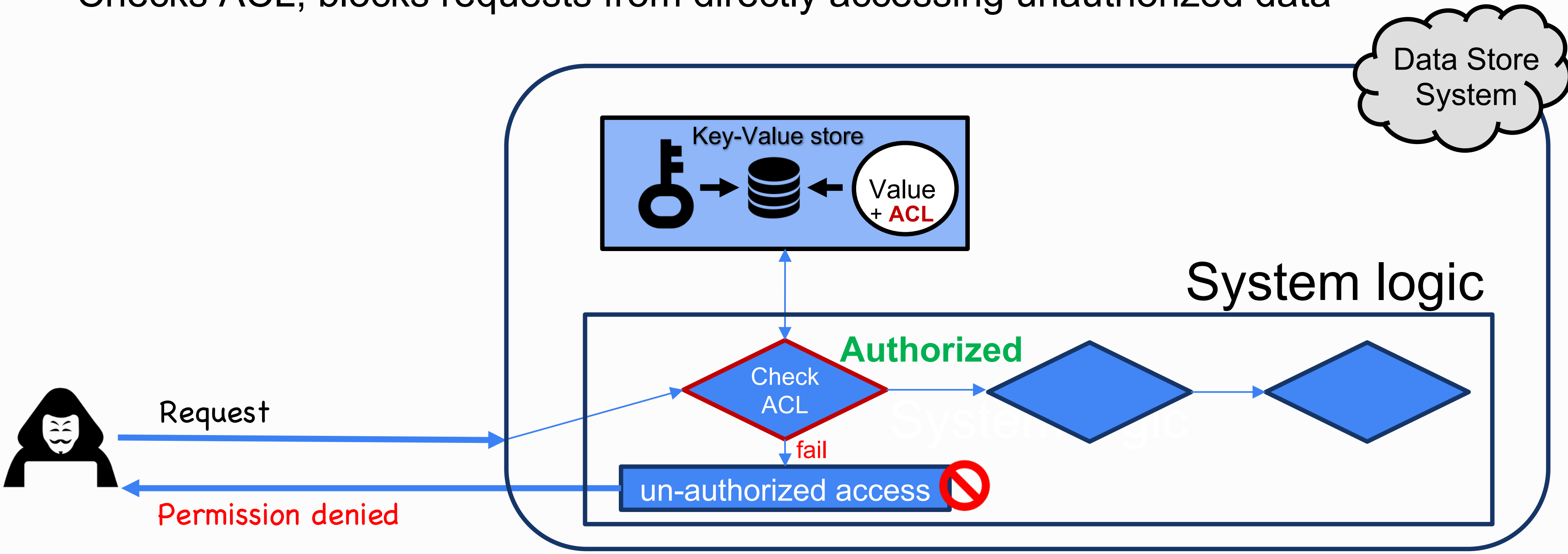
<“ATC_2021” – “ATC_2023”>

<“ATC_2023”: “BOSTON”>
<“ATC_2022”: “CARLSBAD”>
<“ATC_2021”: “VIRTUAL“>

# Key-value store as security components

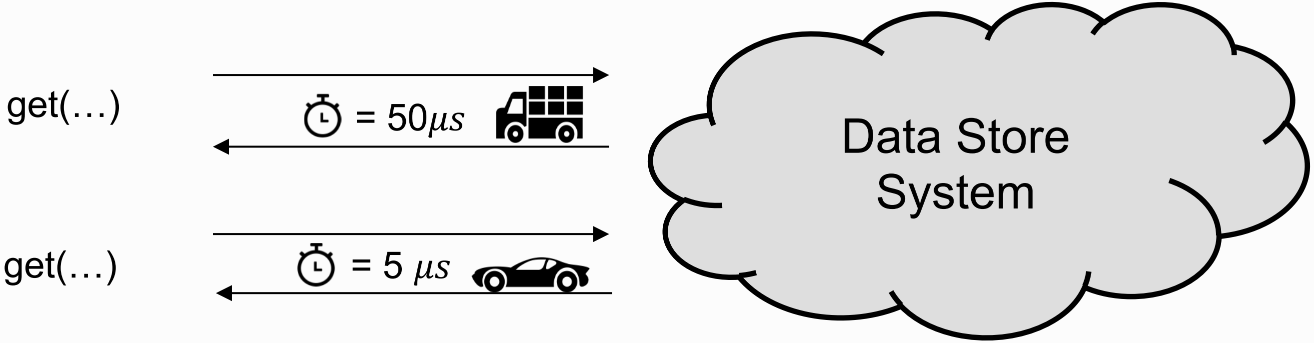
System stores **ACLs** (access control lists) as value metadata in key-value stores

Checks ACL, blocks requests from directly accessing unauthorized data



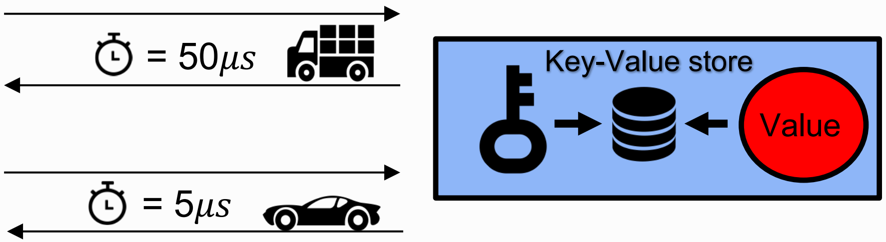
# Timing attacks on key-value stores

A **timing attack** exploits differences in query response times to **glean information about stored data**



# Current timing attacks target values

Existing timing attacks on key-value stores target **stored values**



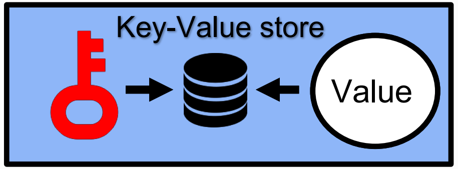

and exploit **external mechanisms**:

- Memory deduplication [Schwarzl et al., NDSS 2022]
- Memory Compression [Schwarzl et al., IEEE S&P 2023]

# Motivation: Key disclosure is also a threat

Keys may be **explicitly** secret

Keys may be **implicitly** considered secret



# Explicitly secret keys

Example: **DB systems with key-value storage engine**
(E.g., CockroachDB, YugabyteDB, MyRocks, …)

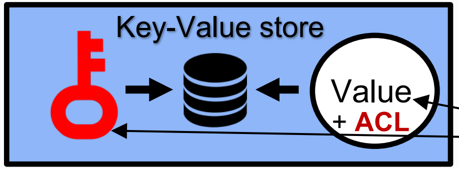
**Value:** Table row
**Key:** Subset of cells (e.g., primary DB key)

⇒ **Key disclosure = DB data disclosure**

Primary key

Table: Patient_Records

| Name | ID | Age | Hospital |
|---|---|---|---|
| Bar | 11467123 | 37 | Mt. Sinai |
| Miguel | 34562788 | 23 | Mt. Sinai |
| Lin | 25262934 | 18 | Mt. Sinai |
| Dan | 45876521 | 29 | Mt. Sinai |
| ... | ... | ... | ... |



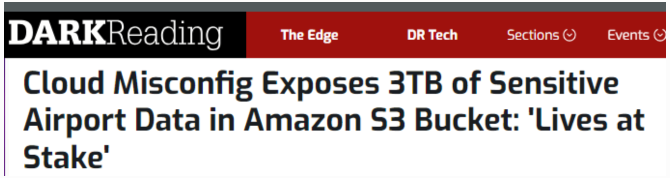
# Keys implicitly considered secret (Hard to guess)

**Use case: object identifiers**

Key-value store maps object IDs to object locations

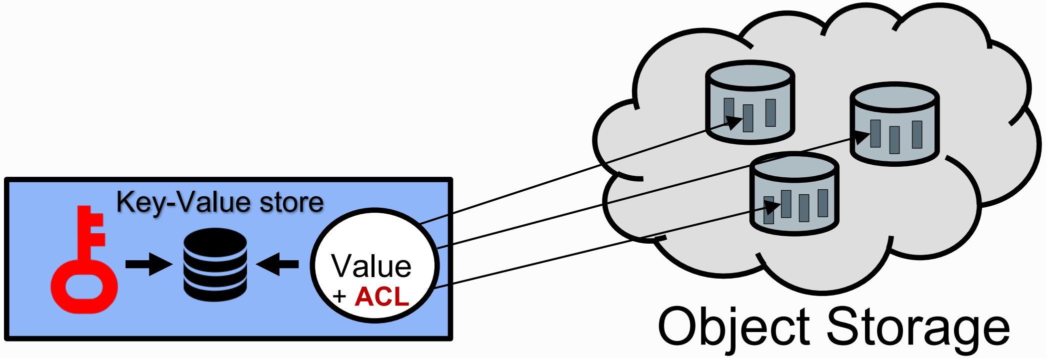
- Finding object IDs can be exploited for further attacks
- Not hypothetical: e.g., scanners for unprotected Amazon S3 objects[1]

[1]



DARKReading | The Edge | DR Tech | Sections | Events

**Cloud Misconfig Exposes 3TB of Sensitive Airport Data in Amazon S3 Bucket: 'Lives at Stake'**



# Contribution: Prefix siphoning

**Key disclosure** timing attack

Exploits **range filters**: an **internal** key-value store **mechanism**

### Attack Template

A general template for key disclosure attacks and the characterization of  vulnerable filters properties

### Proof of concepts (PoCs)

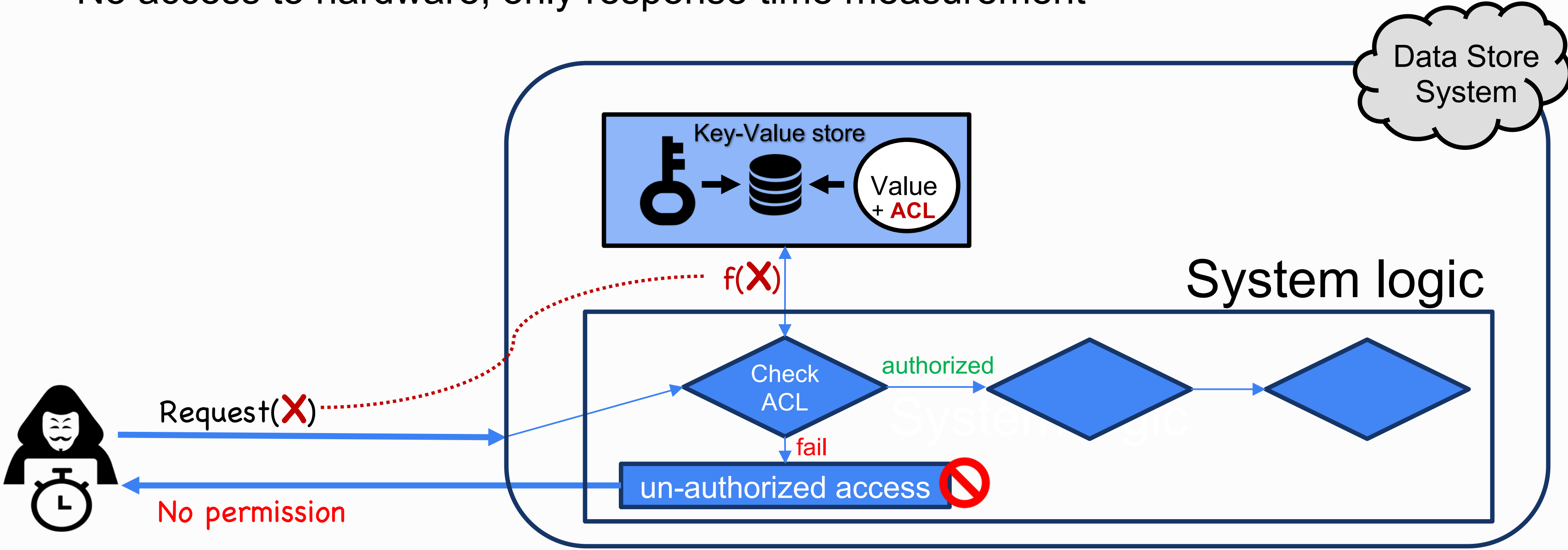
Demonstrate the attack against RocksDB with 2 different range filter types

# Threat model

Attacker can cause system to **point query** its key-value storage engine

⇒ Henceforth, just "query the key-value store"

No access to hardware, only response time measurement



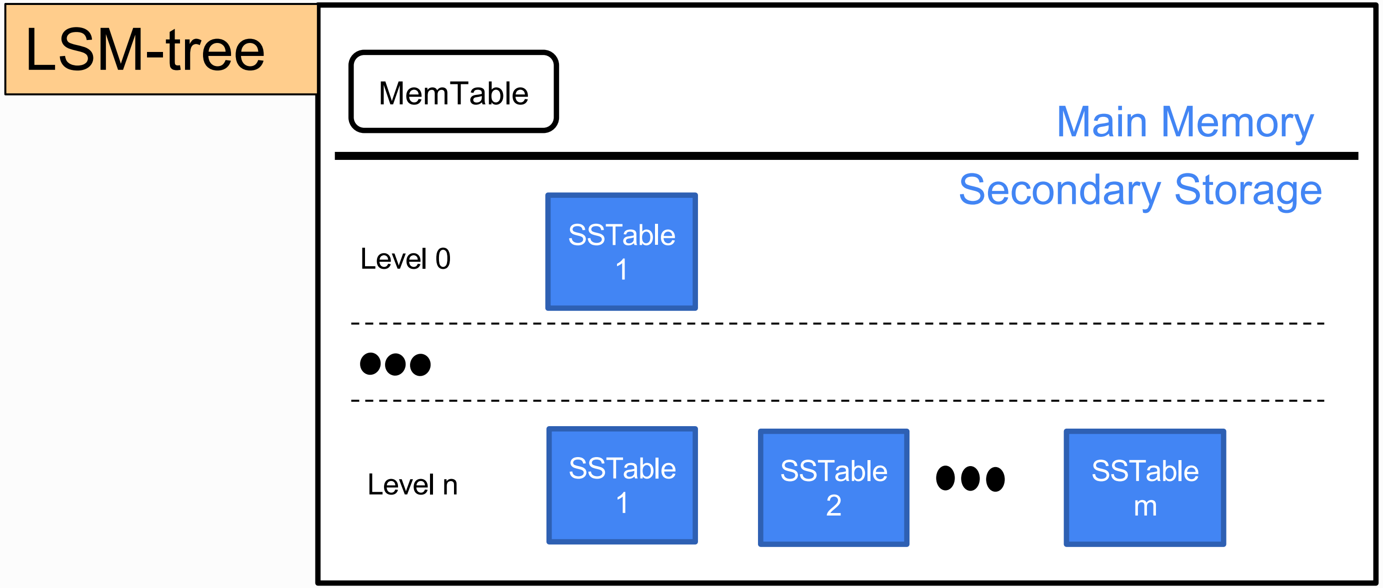
# Focus: LSM-trees

Focus on systems using LSM-tree based key value store

- E.g., RocksDB, LevelDB

LSM-tree uses a mix of in-memory and on-disk storage for (write) performance

Data stored in a series of Sorted Structures/Static Tables (SSTable)



# Filters in LSM-trees

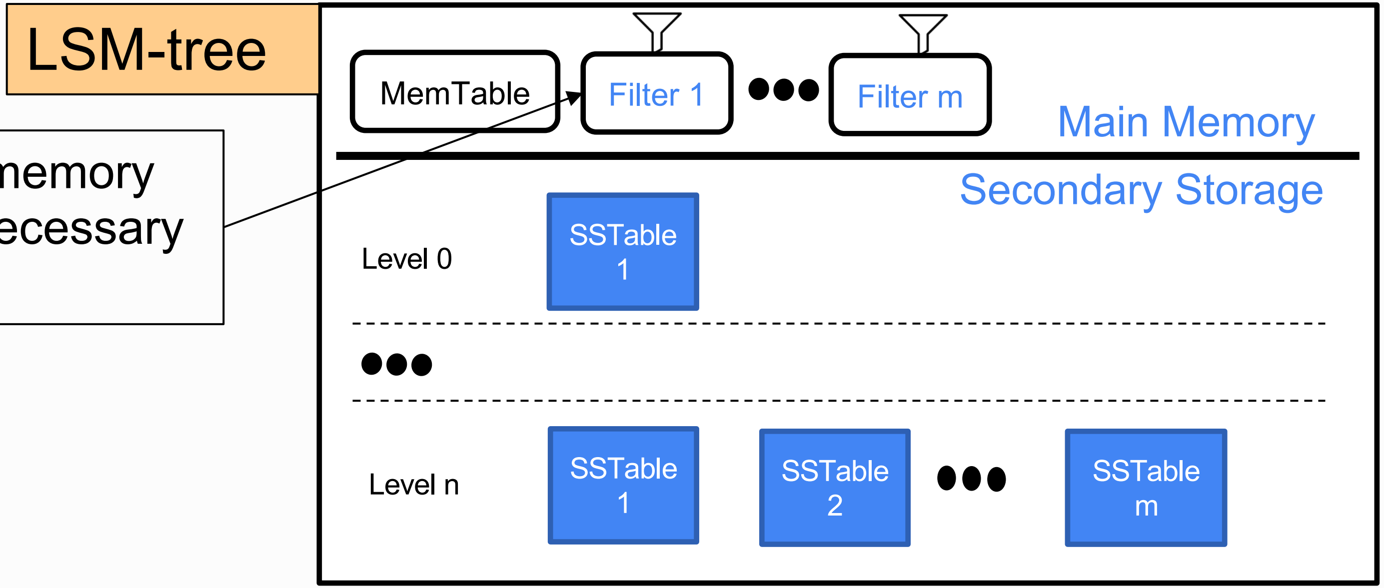
Focus on systems using LSM-tree based key value store

- E.g., RocksDB, LevelDB

LSM-tree uses a mix of in-memory and on-disk storage for (write) performance

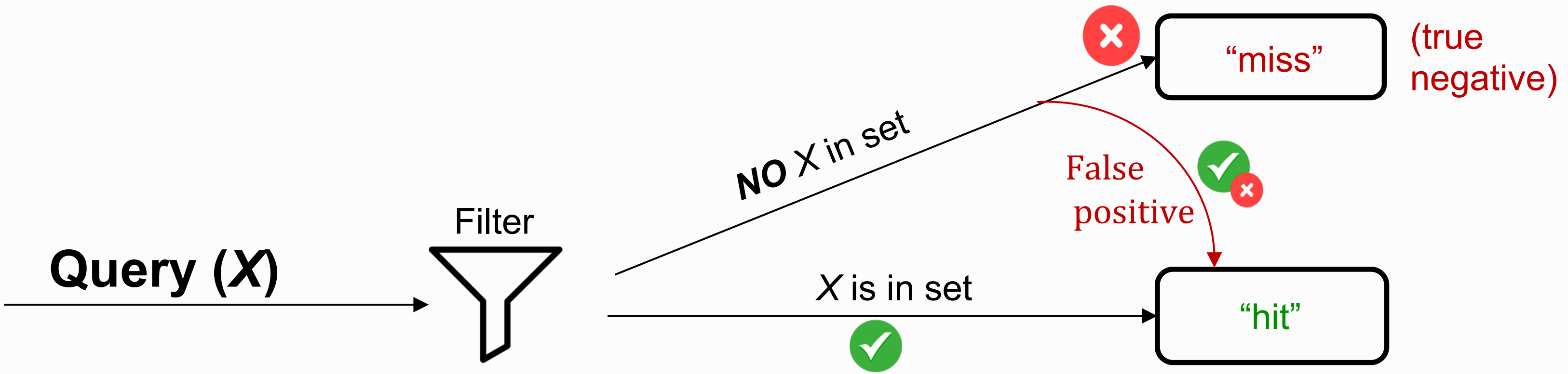
Data stored in a series of Sorted Structures/Static Tables (SSTable)

LSM-Trees use in-memory **filters** to avoid unnecessary I/Os



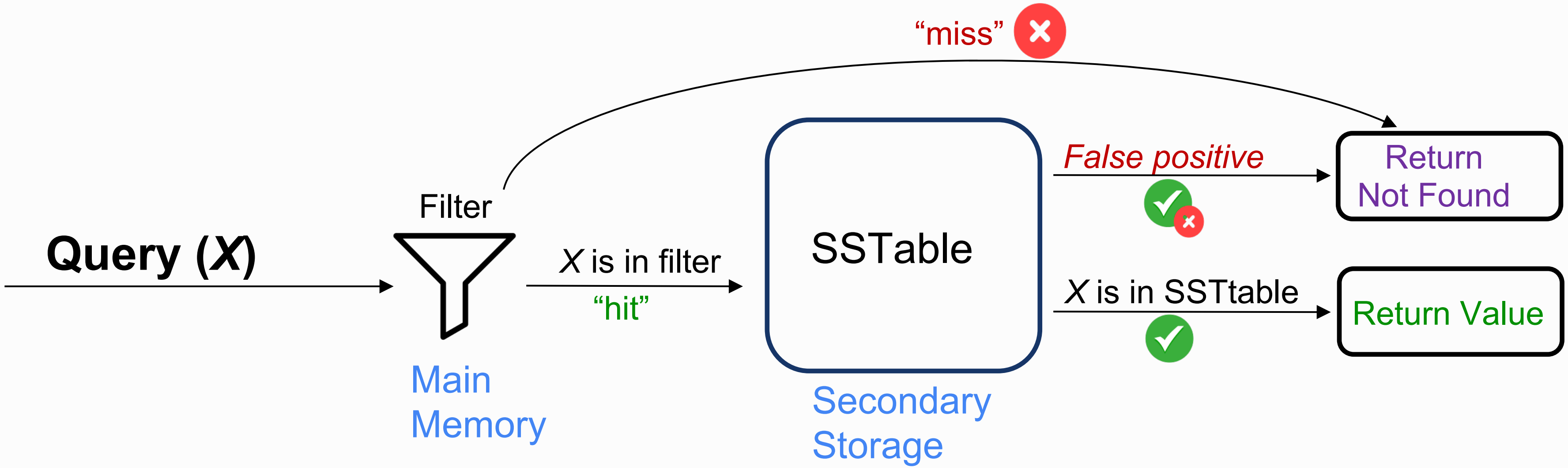
# What is a filter?

**Filter:** Data structure for **approximately** maintaining a set of keys




$$False\ positive\ rate\ (\text{FPR}) = \frac{FP}{FP+TN}$$

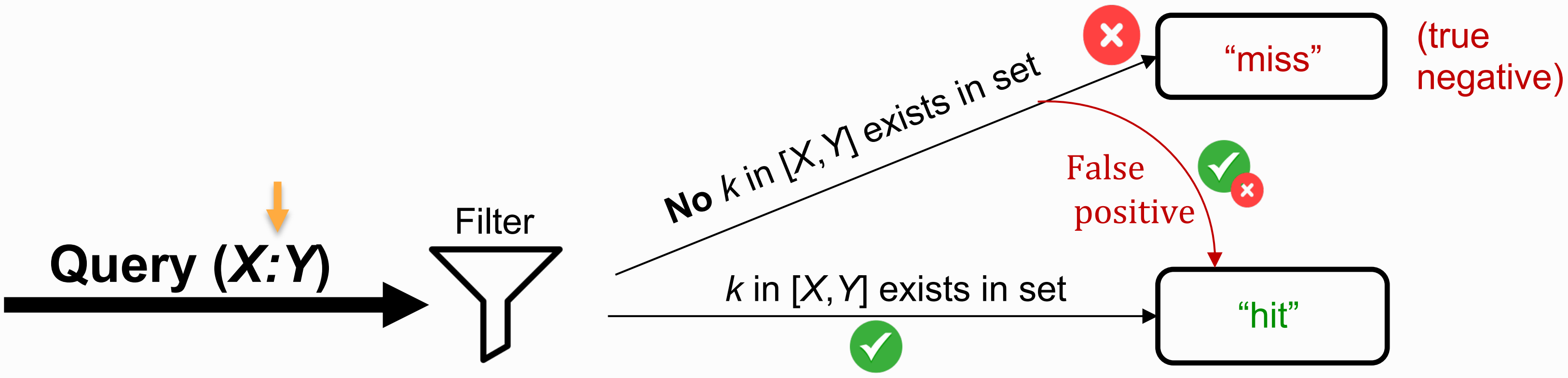
# Filters in LSM-trees



# Range filters

**Range filter**: supports **both range** and **point** queries with false positives

Range query: Is there a key in the dataset between X to Y?




**Insight:** Range query support can affect point query implementation

⇒ Vulnerability to timing attack

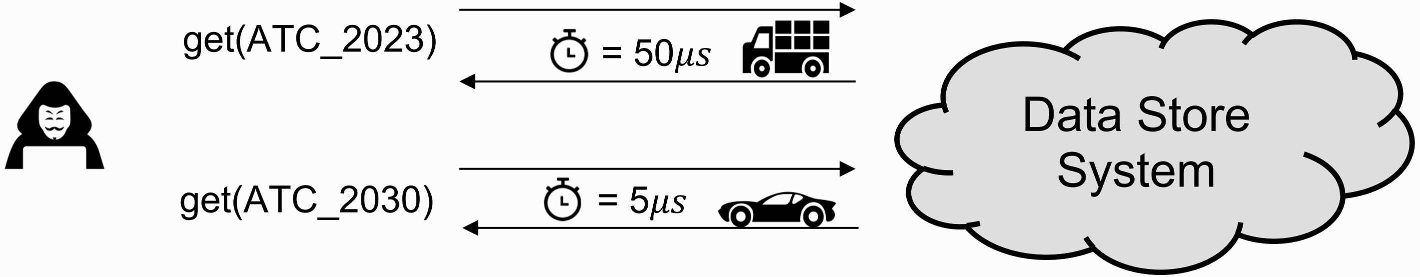
# Prefix siphoning

- General template for timing attacks on systems using LSM-trees with **vulnerable range filters**
- Can reveal **key prefixes** or **full keys**
- Uses **point queries**
- Leverages **key prefix information** stored by the range filter

# Vulnerable range filter characterization

## Common characterization for timing attack on filters

- Measurable response time difference
- Non-negligible FPR (false positive rate)



# Vulnerable range filter characterization

**Common characterization for timing attack on filters**

- Measurable response time difference
- Non-negligible FPR (false positive rate)

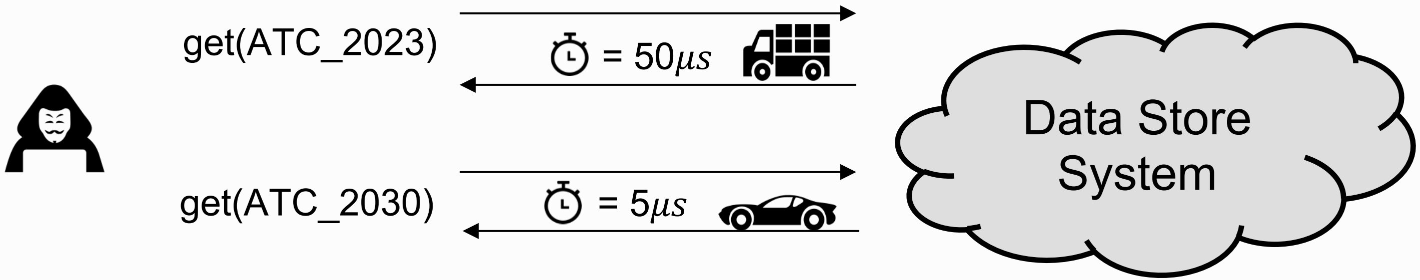
⇒ **Brute force attack**

**Infeasible!**

Query with random keys

Fast response -> no storage access-> “miss”

Slow response -> storage access -> “hit” -> **key exists!**



# Vulnerable range filter characterization

## Common characterization for timing attack on filters

- Measurable response time difference
- Non-negligible FPR (false positive rate)

## Characterization specific to range filters

- A false positive key shares a prefix with a key in the dataset (w.h.p.)
- **FindFPK():** Finds a random false positive key in O(1) queries
- **IdPrefix(FP):** Returns the shared prefix in O(|key|) queries

# Prefix siphoning template

Step 0

**Preliminary:**
Learn to distinguish positive from negative keys

Query with random keys

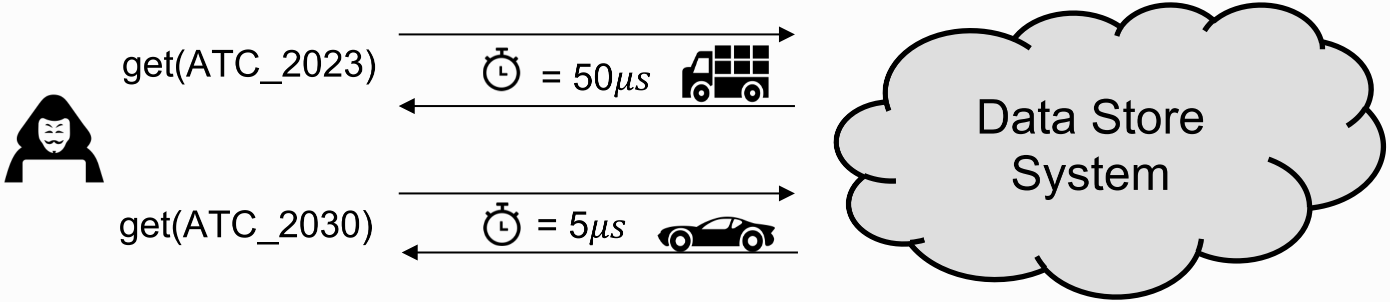
- FPR ensures some “hit” response

Build a distribution of response times

⇒ Bimodal with peaks corresponding to average “fast” and “slow” response times

Slow -> storage access -> “hit”

Fast -> no storage access -> “miss”



# Prefix siphoning template

Step 0

**Preliminary:**
Learn to distinguish positive from negative keys

Step 1

Find initial FP keys

- Call *FindFPK()* multiple times

Step 2

Find the prefixes of the FP key

- Call *IdPrefix(FP)* per FP key

**Extract key prefixes**

# Full key extraction

Step 1

Find initial FP keys

- Call *FindFPK*() multiple time

Step 2

Find the prefixes of the FP key

- Call *IdPrefix(FP)* per FP key

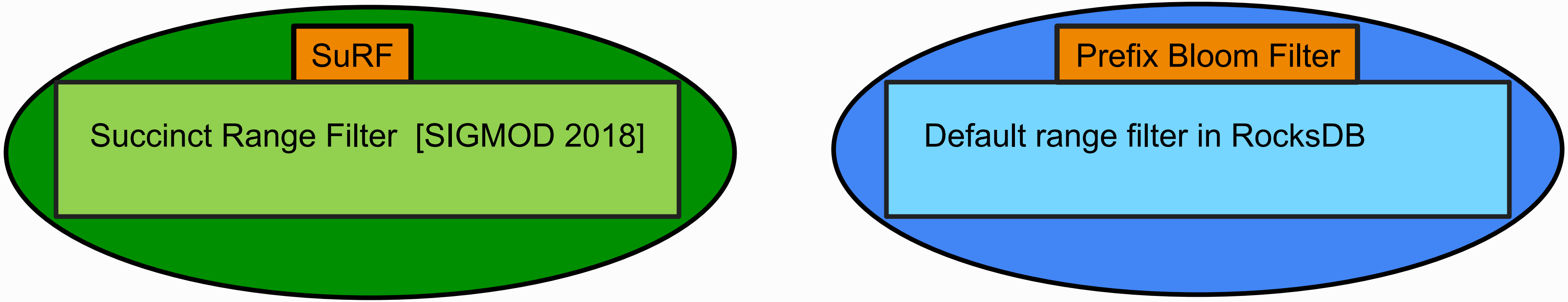
If system has different responses:
"miss": Not Found
"unauthorized hit": Permission Denied

Step 3

Extract full key

- Scan all suffixes per step-2 prefix

# Prefix siphoning instantiations



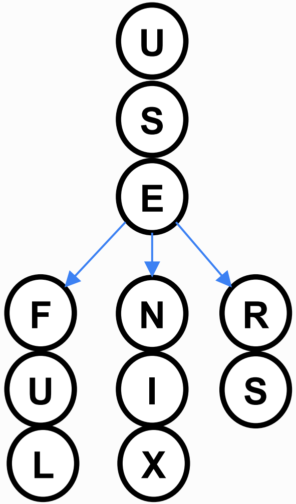
# SuRF – Succinct Range Filter [SIGMOD 2018]

Pruned trie

Stores only the shared prefixes + one extra byte

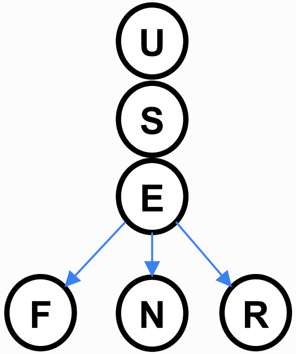
| | Data Store Keys |
|---|---|
| 1 | USEFUL |
| 2 | USENIX |
| 3 | USERS |

**Full trie**




Pruned trie

**SuRF Base**



# SuRF false positives -> vulnerability

Does **USENET** exist in dataset?

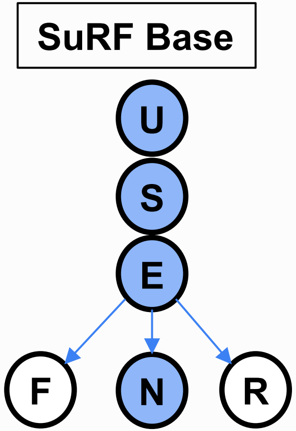
Appears as "hit" in the pruned trie

"hit" -> False positive

| | Point Query |
|---|---|
| | **USENET** |

| | Data Store Keys |
|---|---|
| 1 | USEFUL |
| 2 | **USENIX** |
| 3 | USERS |

Pruned trie



# SuRF-Base vulnerability

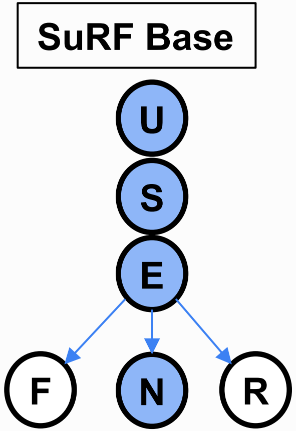
**Insight:** A "hit" indicates the queried key *K* shares a common prefix with a key *K** in the dataset

This is the **longest common prefix** (LCP) *K* shares with any key in the dataset

| | Point Query |
|---|---|
| *K* | **USENET** |

| | Data Store Keys |
|---|---|
| 1 | USEFUL |
| 2 (*K**) | **USENIX** |
| 3 | USERS |

Pruned trie



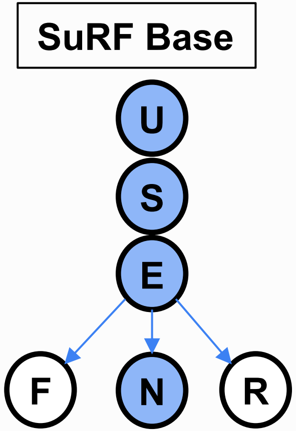
# SuRF-Base attack: (1) Find FP keys

Query random keys until finding a **false positive**

| | Point Query |
|---|---|
| | USENET |

| | Data Store Keys |
|---|---|
| 1 | USEFUL |
| 2 | USENIX |
| 3 | USERS |

Pruned trie



# SuRF-Base attack: (2) Identify prefixes

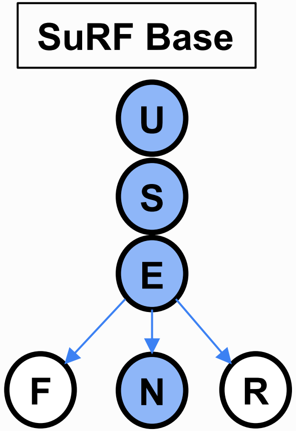
Extract the FP key prefix:

- Truncate FP key, one byte at a time
- Query each such key until a filter "miss" is identified (based on timing)
- Prefix is the last "hit"

| | Point Queries | Filter Result |
|---|---|---|
| 1 | USENET | HIT |
| 2 | USENE | HIT |
| 3 | USEN | HIT |
| 4 | USE | MISS |

| | Data Store Keys |
|---|---|
| 1 | USEFUL |
| 2 | **USENIX** |
| 3 | USERS |

Pruned trie




**Prefix extracted: USEN**

# SuRF-Base attack: (3) Extract full keys

Assuming system responds differently for “not found” and “unauthorized” keys

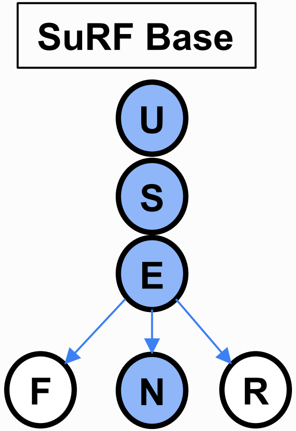
Query the data store with different suffixes

Can be done concurrently with brute force scanning

| | Point Queries | Datastore Result |
|---|---|---|
| 1 | USENAA | Not found |
| 2 | ... | Not found |
| 3 | USENIX | Unauthorized |

| | Data Store Keys |
|---|---|
| 1 | USEFUL |
| 2 | **USENIX** |
| 3 | USERS |

Pruned trie

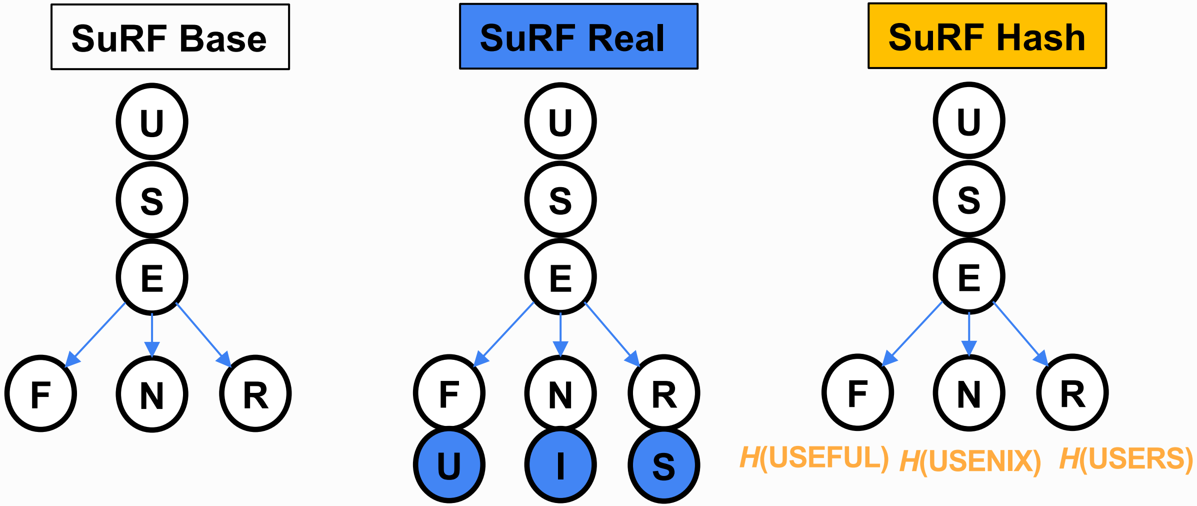



Extracted Key: **USENIX**

# SuRF variants

To improve FPR, store different information after prefix:

- **SuRF-Real**: *n* key bits
- **SuRF-Hash**: *n* bits of a Hash(key)



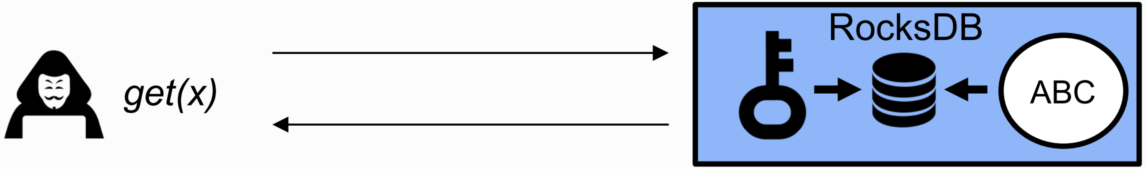
# SuRF-Real attack demonstration

**Target:** SuRF-Real

**System:** RocksDB with SuRF authors' code

**Dataset:** 50M random 64-bit keys

**Background load:** 32 threads constantly performing get()s

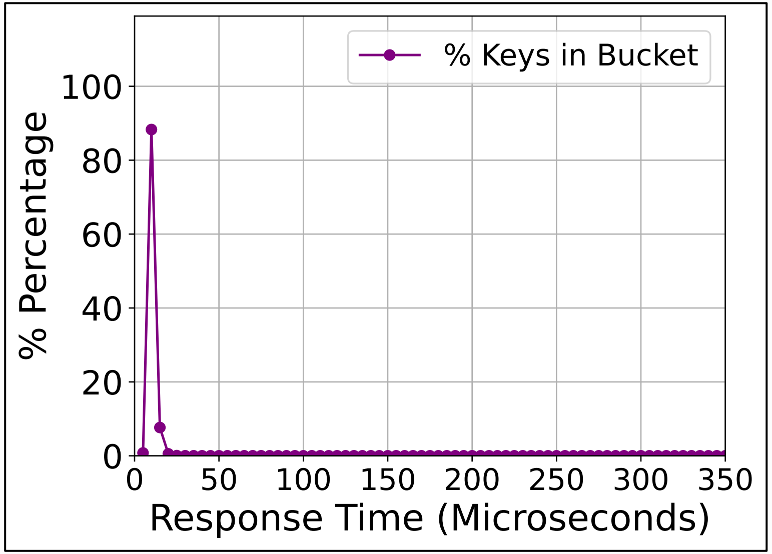


# SuRF-Real attack demonstration

- **Step 0** – Distinguish between a filter “hit” and “miss” -> 10M queries
- **Step 1** – Find initial FP keys -> 10M queries
- **Step 2** – Identify shared prefixes
- **Step 3** – Suffix scan for prefixes over 40 bits [not measuring time].
  (Parallelized using 16 cores)

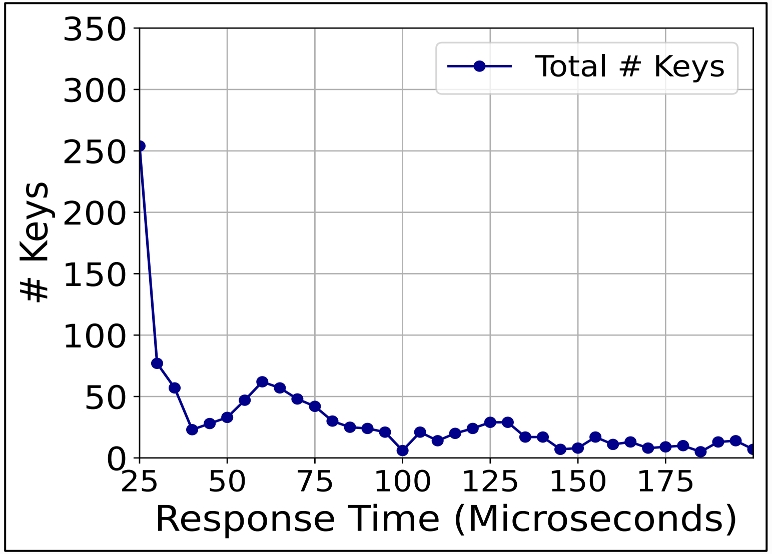
# Step 0 – Identify false positive response time

- 97% of the queries took $< 25\mu s$



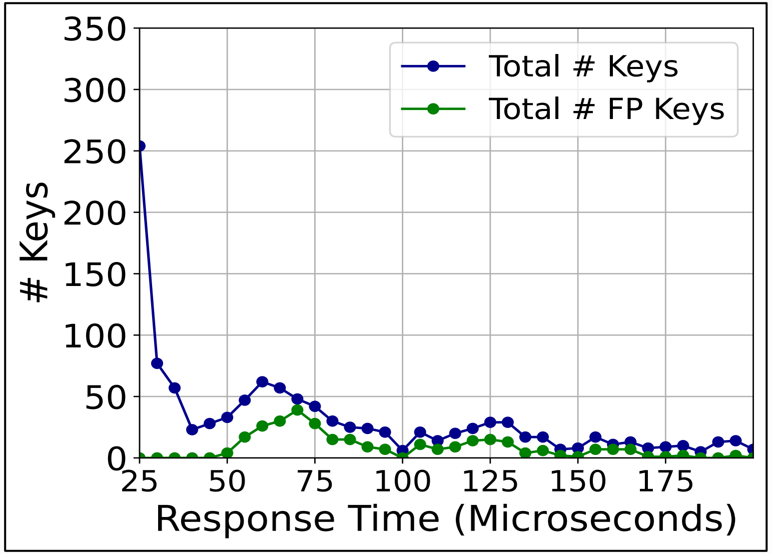
# Step 0 – Identify false positive response time

- 97% of the queries took $< 25\mu s$
- Zoom in: $> 25\mu s$



# Step 0 – Distinguish false and true positive keys

- 97% of the queries took $< 25\mu s$
- Zoom in: $> 25\mu s$
- 98% of the FP keys took $> 50\mu s$
- Major portion of queries $> 50\mu s$ are FP

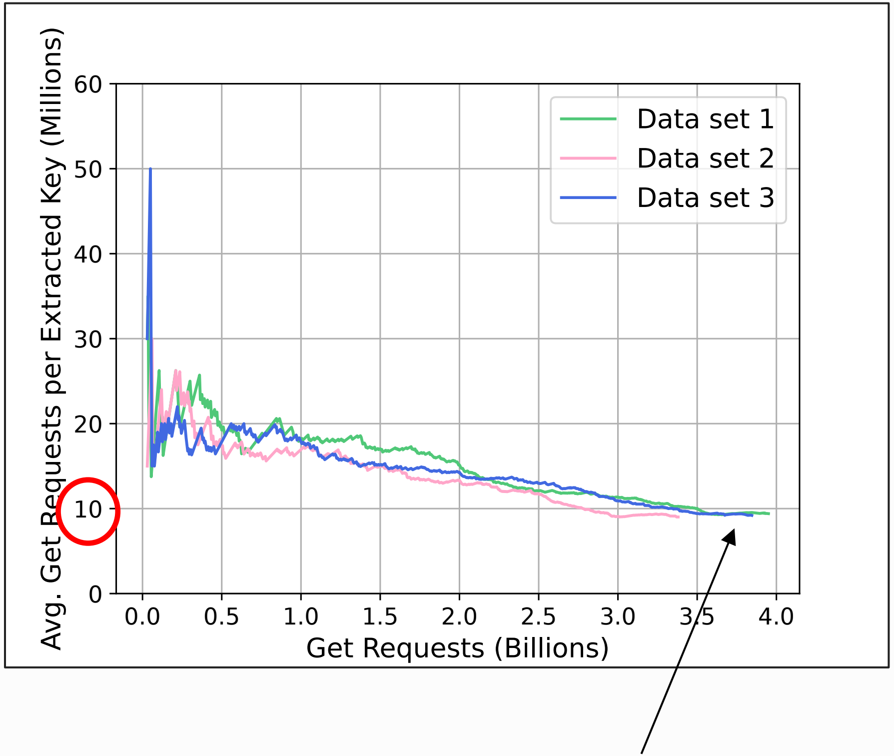


# Attack efficiency

**<10M** get() requests **per extracted key**

**40,000x better** than brute force

Repeated over different datasets



# Mitigations

Every mitigation is a **trade-off**

**System-level:**

- Query key-value store only for authorized keys — ☹ Re-architecting
- Rate limiting — ☹ Throughput

**Key-value store:**

- Separate filters for point/range queries — ☹ Memory waste

**Resilient range filter:**

- E.g., Rosetta [SIGMOD 2020] — ☹ Variable-length keys

# Mitigations

Every mitigation is a **trade-off**

**System-level:**

- Query key-value store only for authorized keys ☹ Re-architecting
- Rate limiting ☹ Throughput

**Key-value store:**

- Separate filters for point/range queries ☹ Memory waste

**Resilient range filter:**

- E.g., Rosetta [SIGMOD 2020] ☹ Variable-length keys

# Conclusion

**Prefix siphoning:**
General template for **key disclosure** timing attacks
against LSM-tree key-value stores with vulnerable range filters

Security vs. performance trade-off
$\Rightarrow$ More security analysis of optimizations!

Research on secure range filters