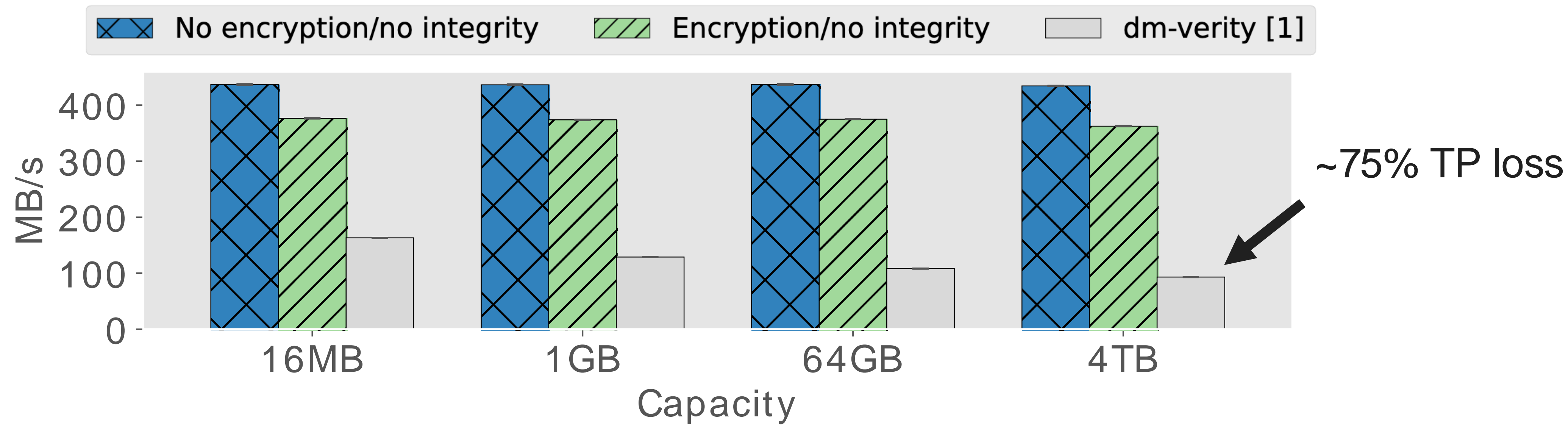
On Scalable Integrity Checking for Secure Cloud Disks

Quinn Burke, Ryan Sheatsley, Rachel King, Owen Hines,
Michael Swift, Patrick McDaniel

Integrity is costly in practice



Experiment params:
Cache=10%, Read ratio=1%, I/O size=32KB, Workload=Zipf(2.5), Threads=1, I/O depth=32

Our work

We developed optimized integrity data structures that exploit patterns in workloads to reduce integrity costs.

Research questions:

1. What is the root cause of integrity overheads?
2. Can we model an *optimal* integrity data structure?
3. Can we realize (or approximate) the optimal in a real system?

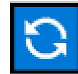
Our work

We developed optimized integrity data structures that exploit patterns in workloads to reduce integrity costs.

Research questions:

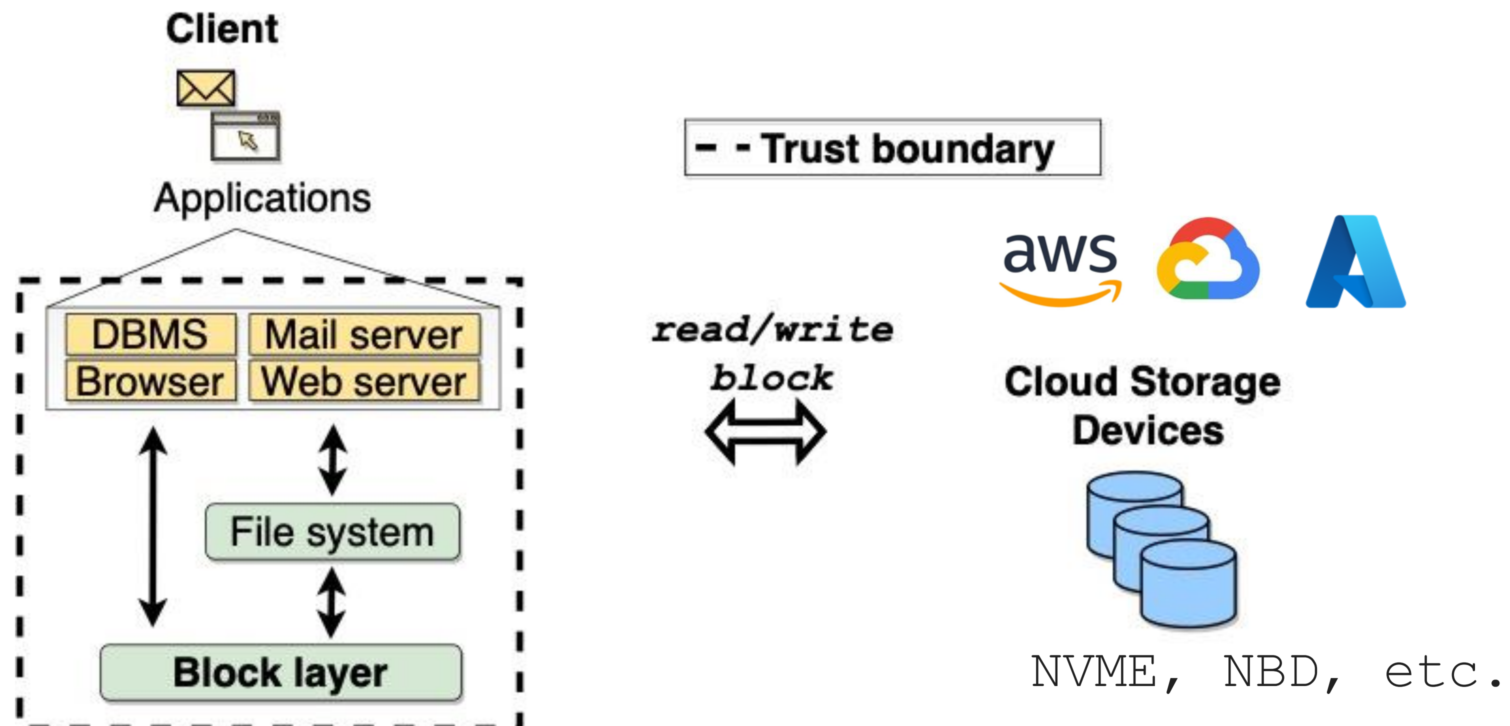
1. What is the root cause of integrity overheads?
   - A: Hashing (CPU costs)

2. Can we model an *optimal* integrity data structure?
   - A: Yes, with a priori knowledge



3. Can we realize (or approximate) the optimal in a real system?
   - A: Yes, by learning workload patterns

Background: Integrity checking in practice



Background: Integrity checking in practice

Application isolation enforced at hardware-level (Intel TDX, ARM CCA, etc.)

Client

Applications

DBMS | Mail server
Browser | Web server

File system

Block layer

Security layer

Confidential VM

- - Trust boundary

read/write block

read/write security metadata

Cloud Storage Devices

NVME, NBD, etc.

Security layer interface:

`update()` security metadata when block is written

`verify()` security metadata when block is read

Background: Security metadata

Integrity guarantees:

MACs ⟶ Authenticity

Merkle tree ⟶ Freshness

`write():`

- Compute new block MAC, update tree with MAC, return to caller

`read():`

- Check MAC against block, verify MAC in tree, return to caller

Background: Security metadata

Integrity guarantees:

- MACs ⟶ Authenticity
- Merkle tree ⟶ Freshness

`write()`:

- Compute new block MAC, update tree with MAC, return to caller

`read()`:

- Check MAC against block, verify MAC in tree, return to caller

Accessed block/leaf Proof Update/Verify path

Root hash (sealed in TPM, on-chip, etc.)

Internal nodes

SHA256(H0 || H1)

H0 H1 H2 H3 H4 H5 H6 H7

Leaf nodes

H0	H1	H2	H3
H4	H5	H6	H7

Security metadata

B0	B1	B2	B3
B4	B5	B6	B7

Data blocks

aws

CONFIDENTIAL CONTAINERS

Root cause: Merkle tree traversals

Integrity guarantees:

- MACs → Authenticity
- Merkle tree → Freshness

`write()`:

- Compute new block MAC, update tree with MAC, return to caller

`read()`:

- Check MAC against block, verify MAC in tree, return to caller

Accessed block/leaf   Proof   Update/Verify path

Root hash (sealed in TPM, on-chip, etc.)

Bottleneck

SHA256(H0 ||

Internal nodes

H0 H1 H2 H3 H4 H5 H6 H7

Leaf nodes

H0	H1	H2	H3
H4	H5	H6	H7

Security metadata

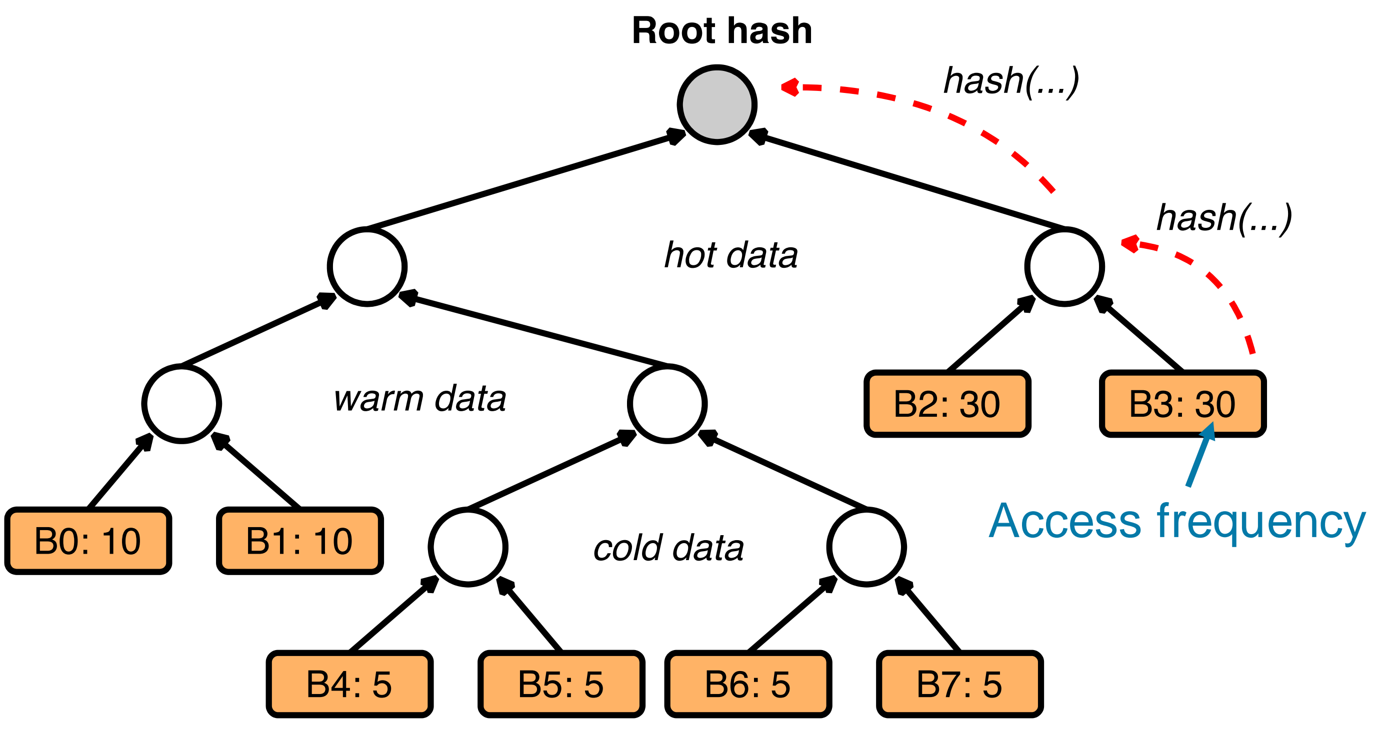
B0	B1	B2	B3
B4	B5	B6	B7

Data blocks

aws

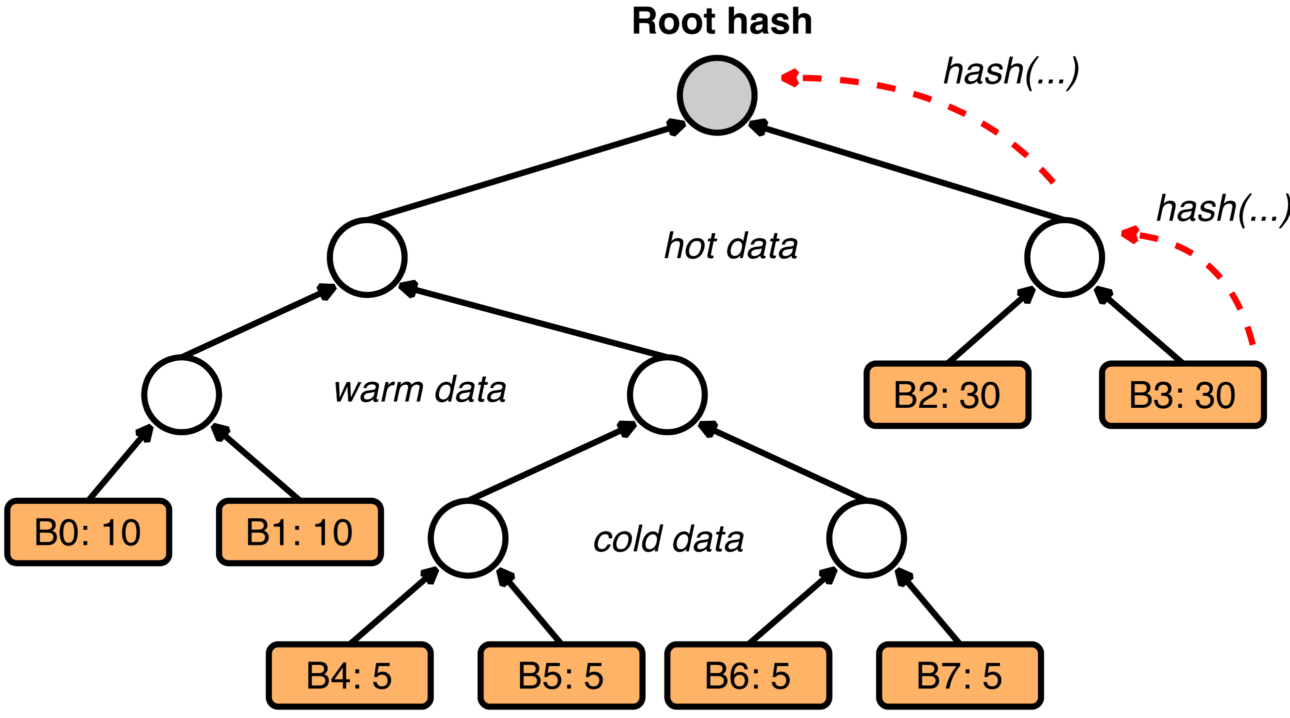
CONFIDENTIAL CONTAINERS

Opportunity: Exploit reference locality



- Allow the tree to become **unbalanced**
- Keep **hot leaves** (i.e., block MACs) closer to root
- Leads to lower hashing costs for hotter data (and overall, by weighted average)

Opportunity: Exploit reference locality



Prior works:

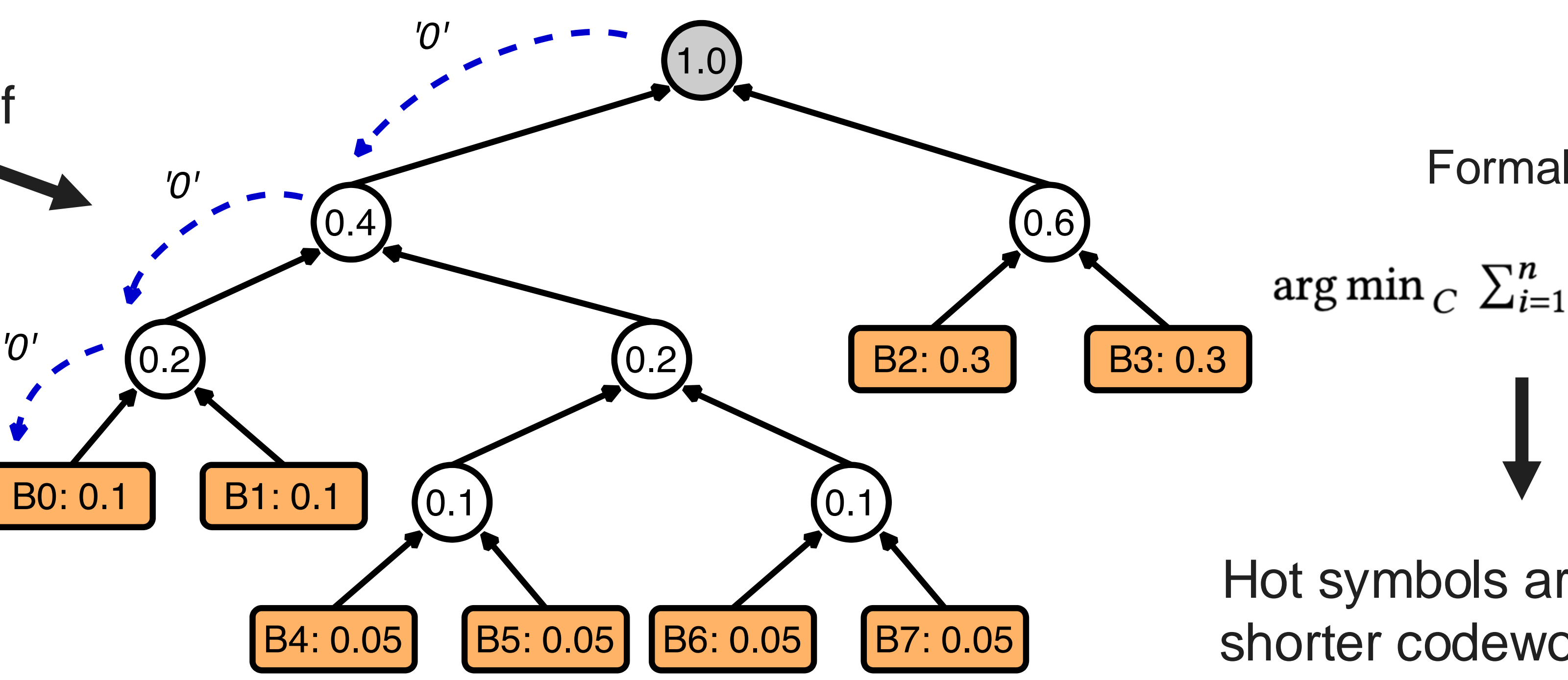
1. Focused on offline integrity checks, performance not a priority, balanced trees suffice
2. Focused on slow media or small capacity

Our work:

1. Optimal tree definition
2. Optimal tree approximation (heuristic)

Modelling optimal performance

Encoding: traverse a path to the symbol leaf and append a '0' or '1'



Formally:

$$\arg\min_C \sum_{i=1}^{n} w_i |c_i|, c_i \in C$$

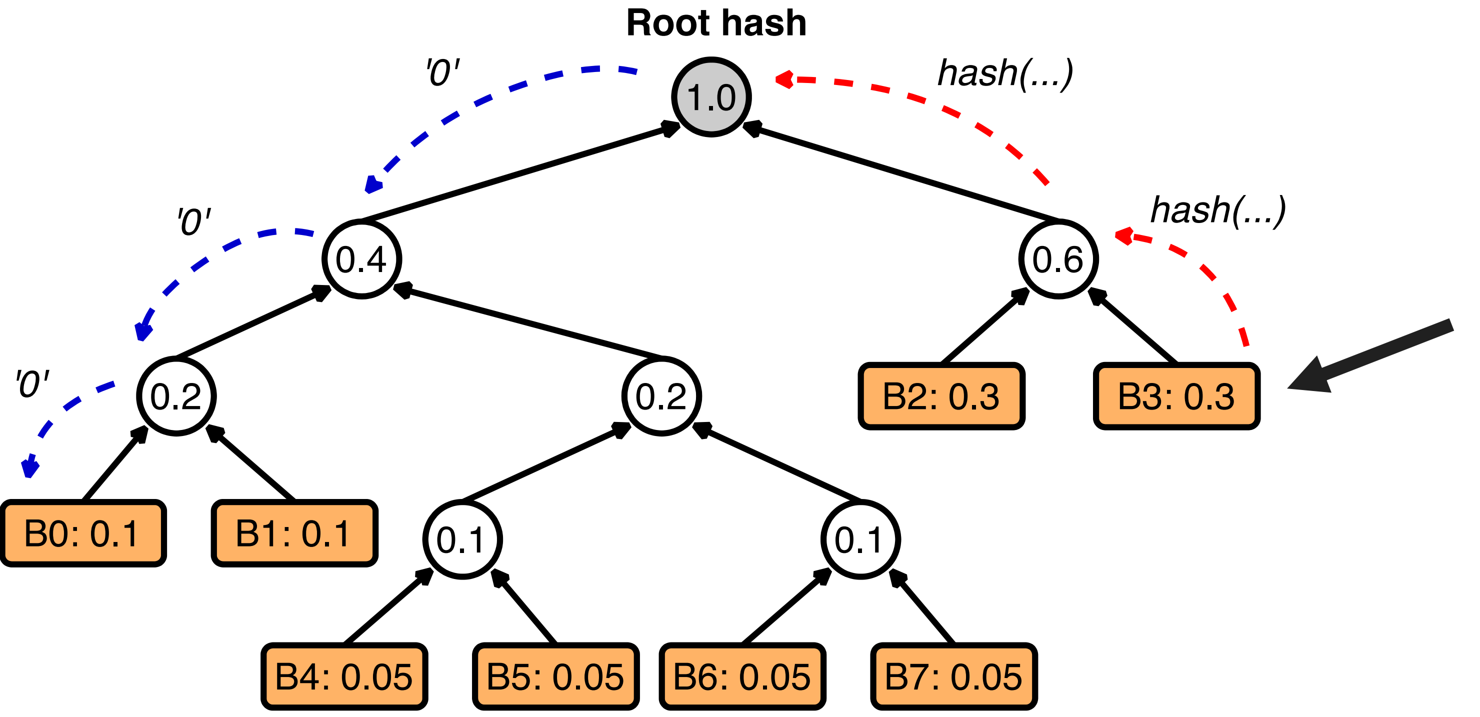
Hot symbols are assigned shorter codewords (paths)

Access probabilities

Symbol/Block	B0	B1	B2	B3	B4	B5	B6	B7
Probability	0.1	0.1	0.3	0.3	0.05	0.05	0.05	0.05

Modelling a hash tree as a prefix tree

hashes computed during update/verification
bits in codeword (c_{B0} = '000')

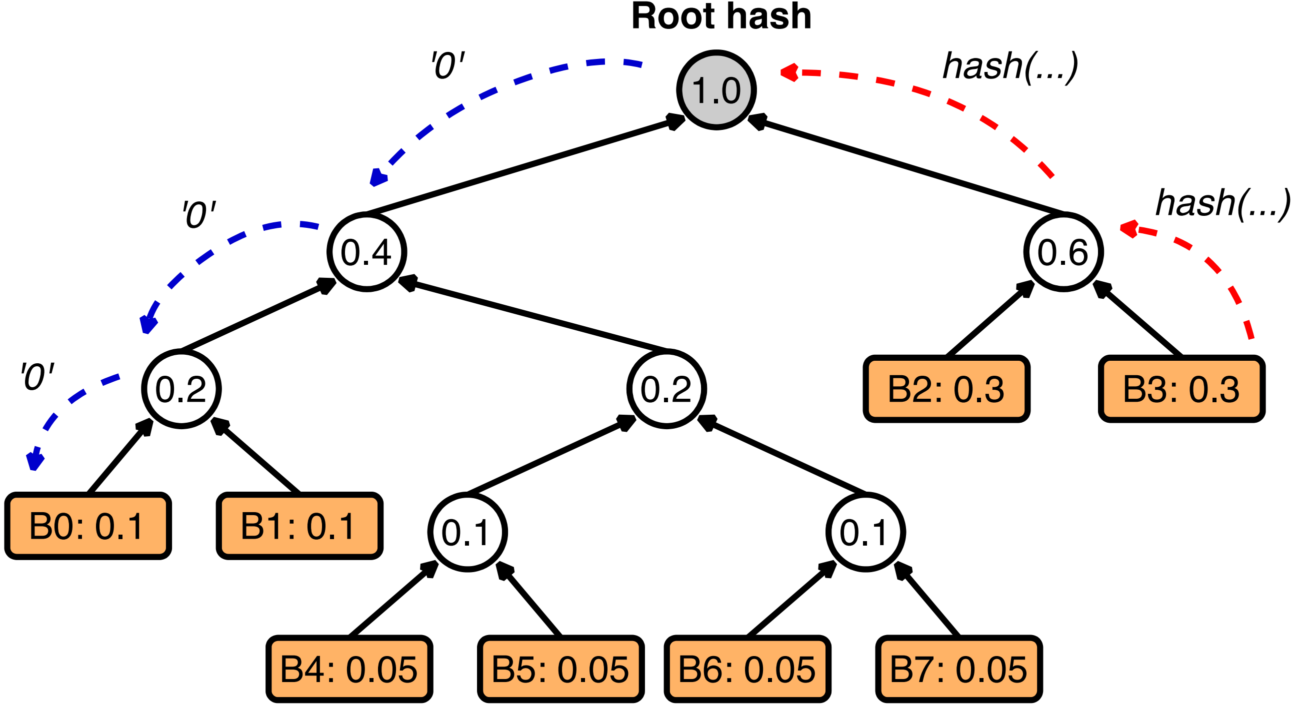


Access probabilities

Symbol/Block	*B0*	*B1*	*B2*	*B3*	*B4*	*B5*	*B6*	*B7*
Probability	*0.1*	*0.1*	*0.3*	*0.3*	*0.05*	*0.05*	*0.05*	*0.05*

- Frequently accessed MACs (i.e., blocks) have shorter paths
- Shorter paths = lower hashing costs (for hot data and overall)

Optimal trees require a priori knowledge



Access probabilities

Symbol/Block	*B0*	*B1*	*B2*	*B3*	*B4*	*B5*	*B6*	*B7*
Probability	*0.1*	*0.1*	*0.3*	*0.3*	*0.05*	*0.05*	*0.05*	*0.05*

Condition:

Optimal for a **known**, fixed access probability distribution

Utility:

Establishes an upper bound on performance (i.e., an optimal tree oracle)

Learning workload patterns: *Dynamic Merkle Trees (DMTs)*

zig

1 *rotate right*

p x B B0 A → x B0 p A B

B0: +1 level

Update from: p

zig-zig

1 *rotate right*

g p C x B B0 A → 2 *rotate right* p x g B0 A B C → x B0 p A g B C

Update from: g

B0: +2 levels

Update from: p

zig-zag

1 *rotate left*

g p C A x B B0 → 2 *rotate right* g x C p B0 A B → x p g A B B0 C

Update from: p

B0: +1 level

Update from: g

Mechanism:

- When a MAC (i.e., block) is accessed, rotate it higher in the tree

Splay trees:

- Unbalanced tree structure
- No a priori knowledge
- Start from any initial state
- Quickly adapts to new regions of interest

DMT operation

zig

1 *rotate right*

p x B B0 A → x B0 p A B

B0: +1 level

Update from: p

zig-zig

1 *rotate right*

g p C x B B0 A → 2 *rotate right*

p x g B0 A B C

Update from: g

→ B0: +2 levels

x B0 p A g B C

Update from: p

zig-zag

1 *rotate left*

g p C A x B B0 → 2 *rotate right*

g x C p B0 A B

Update from: p

→ B0: +1 level

x p g A B B0 C

Update from: g

Heuristic parameters: splay probability *p* and splay distance *d*

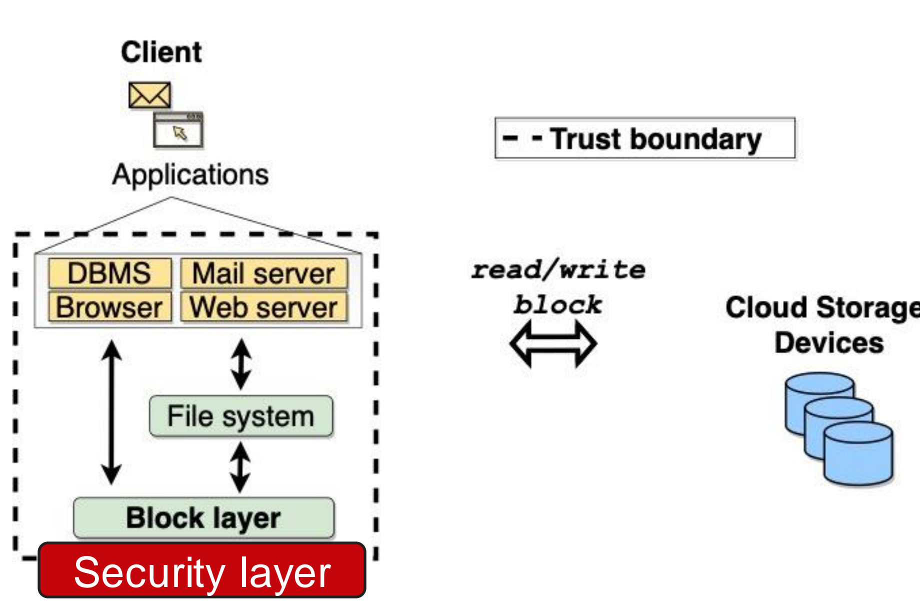
Technical approach:

1. Track node hotness
2. Rotate nodes up the tree based on their hotness
3. Maintain tree invariants

Implementation & Experiment setup

- **Implementation:** As a block device driver, 5K LoC (C++) using the BDUS framework
- **Platform:** Runs on standard Linux systems
- **Setup:** AWS EC2 i4i.8xlarge instances with local NVMe disks
- **Workloads:** Fio-generated Zipfian workloads, an Alibaba trace dataset, and Filebench

- **Evaluation questions (more in the paper):**
  1. Can DMTs scale better with capacity?
  2. Can DMTs adapt to workload changes?

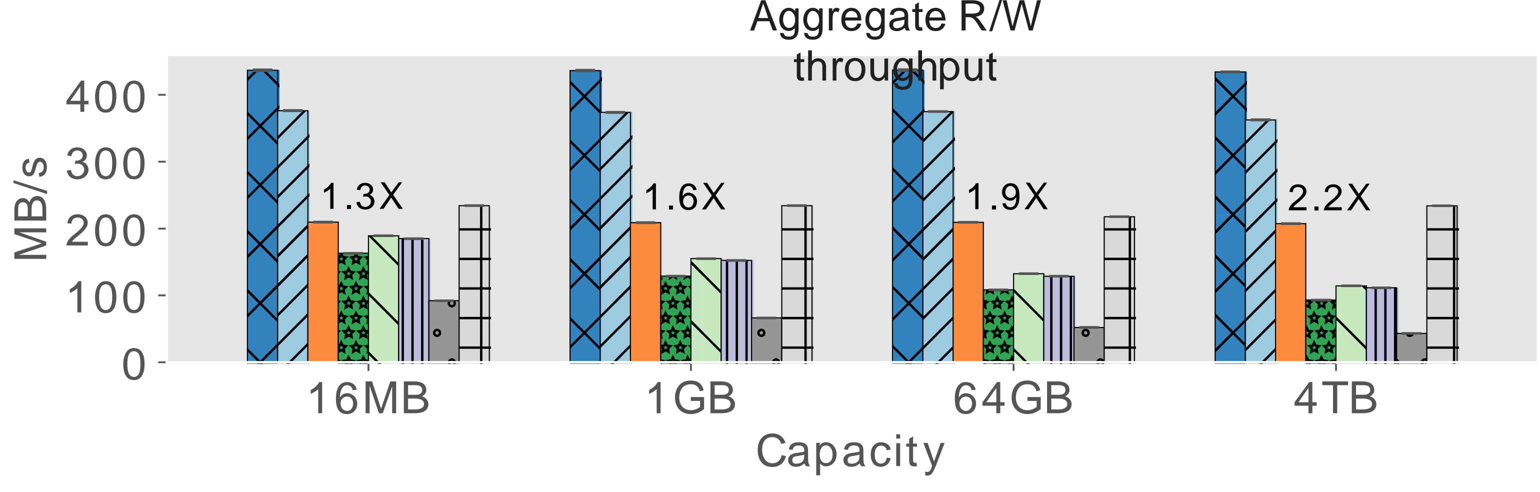


Key result: DMTs scale better with capacity

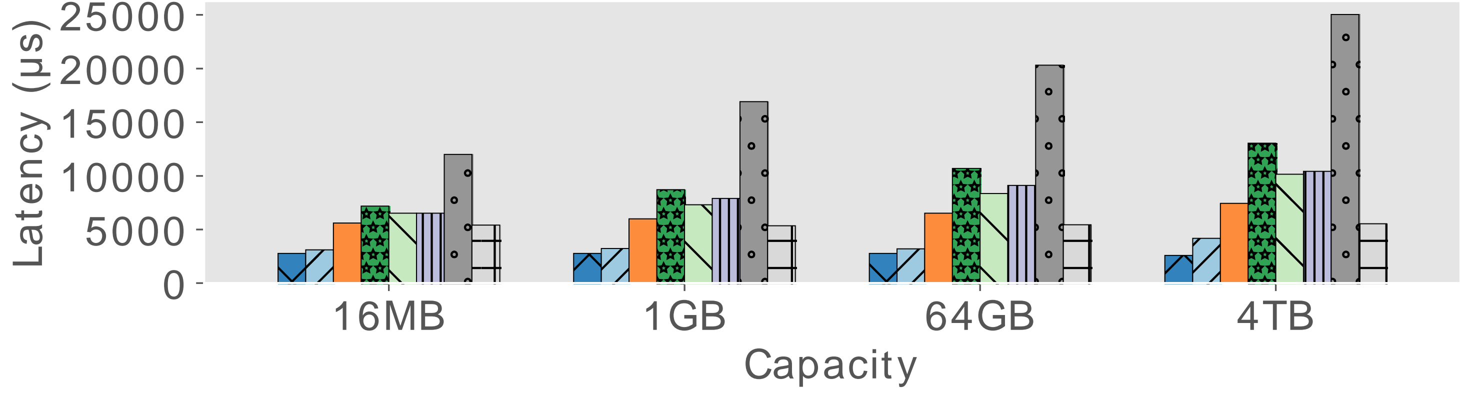
No encryption/no integrity
Encryption/no integrity
DMT
dm-verity [12]
4-ary
8-ary
64-ary
H-OPT

Max achievable throughput

Aggregate R/W throughput



DMTs deliver up to 2.2X speedup and consistently provide >85% of optimal (H-OPT) throughput



DMTs provide up to 40% lower p99.9 latency – through our design, splay costs are amortized over time

Experiment params:
Cache=10%, Read ratio=1%, I/O size=32KB, Workload=Zipf(2.5), Threads=1, I/O depth=32

Key result: DMTs adapt quickly to workload changes

dm-verity [1]　DMT

Uniform　Zipf

Aggregate TP (MB/s)

0 50 100 150 200

0 20 40 60 80 100 120 140

Time (s)

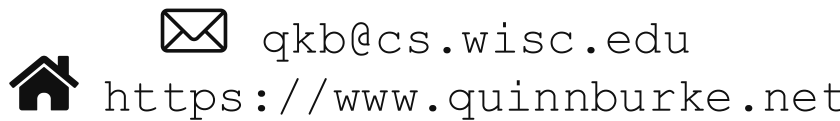
DMTs adapt quickly at the start of Zipf phases

DMTs perform comparable to dm-verity during uniform phases

Experiment params:
Capacity=1TB, Cache=10%, Read ratio=1%, I/O size=32KB,
Workload=alternating Zipf / Uniform, Threads=1, I/O depth=32

Questions?

- Storage integrity is essential, but costly in practice
- The root cause is Merkle tree traversal (hashing) costs
- Costs can be systematically reduced by formalizing optimization objectives [1] and leveraging adaptive data structures
- DMTs can be extended to other use cases (other data structures, distributed storage deployments, etc.)
- Code: https://github.com/MadSP-McDaniel/dmt



[1] Keeton, Kimberly, et al. "Don't Settle for Less Than the Best: Use Optimization to Make Decisions." *HotOS*. 2007.