dm-x: Protecting Volume-level Integrity for Cloud Volumes and Local Block Devices

Anrin Chakraborti
Stony Brook University

Bhushan Jain
Stony Brook University &
UNC, Chapel Hill

Jan Kasiak
Stony Brook University

Tao Zhang
Stony Brook University &
UNC, Chapel Hill

Donald Porter
Stony Brook University &
UNC, Chapel Hill

Radu Sion
Stony Brook University

ABSTRACT

The verified boot feature in recent Android devices, which deploys dm-verity, has been overwhelmingly successful in eliminating the extremely popular Android smart phone rooting movement [25]. Unfortunately, dm-verity integrity guarantees are read-only and do not extend to writable volumes.

This paper introduces a new device mapper, **dm-x**, that efficiently (fast) and reliably (metadata journaling) assures volume-level integrity for entire, writable volumes. In a direct disk setup, dm-x overheads are around 6-35% over ext4 on the raw volume while offering additional integrity guarantees. For cloud storage (Amazon EBS), dm-x overheads are negligible.

1 INTRODUCTION

Dramatic advances have been made in recent years towards building tamper-evident systems with a trusted computing base (TCB). Recent TPM-based systems provide strong guarantees, even against attackers with temporary physical access to the machine, by refusing to attest malicious software, and by withholding decryption keys for untrusted storage devices. In total, these systems assure that, at least at start-up time, the system is starting from a clean boot of trusted software. Unfortunately, once a system boots into a trusted software stack, the kernel almost always mounts filesystems from disks without equally-strong integrity protection. Recent studies of kernel rootkits demonstrate that, even when the integrity of kernel code is protected, compromised disk data can compromise kernel security by violating assumptions about data structure layouts [7, 12].

This is significantly more critical for cloud services where the storage server can be malicious, compromised, or simply not as diligent in ensuring regulatory compliance or security best practices. In fact, secure hybrid cloud strategies [9] based on Amazon's Virtual Private Cloud (VPC) [2] (which provides strong security guarantees through network isolation) store data on untrusted storage devices residing in the public cloud. For example, Amazon VPC file systems, object stores, and databases reside on virtual block devices in the Amazon Elastic Block Storage (EBS).

This allows numerous attack vectors for a malicious cloud provider/untrusted software running on the server. For instance, to ensure SEC-mandated assurances of end-to-end integrity, banks need to guarantee that the external cloud storage service is unable to remove log entries documenting financial transactions. Yet, without integrity of the storage device, a malicious cloud storage service could remove logs of a coordinated attack. By determining which blocks store these sensitive logs, such as by monitoring concurrent write patterns [24], a malicious cloud service can selectively replace these blocks with old, innocuous values. Such an attack on the logs can go unnoticed, potentially indefinitely.

At the root of these vulnerabilities is a lack of volume-level, read/write integrity protection. Simply encrypting the data using current block-level encryption solutions, such as dm-crypt or Bitlocker, does not achieve the necessary security guarantees. Block device encryption cannot ensure freshness of writes: an adversarial cloud storage volume may discard writes during an attack. As with checksumming individual blocks, when blocks are encrypted independently, old ciphertexts can be replayed.

Currently-available block device integrity solutions either ensure block-level integrity (dm-integrity [15]) and are prone to replay attacks (described further), or can only support read-only integrity verification (dm-verity [3]).

To address this need, we introduce **dm-x**—a new Linux block device mapper that ensures volume-level, cryptographically-strong integrity of data both *at rest and at runtime*. dm-x can detect a malicious remote storage server, normal data corruption, and physical device tampering. dm-x is efficient, with storage overheads that are constant in trusted space and logarithmic in untrusted space. To provide robustness against system crashes and power failures, dm-x **integrates metadata journaling with integrity**.

When compared with existing integrity-preserving file systems (such as [16]), a particular advantage of integrity protection at the block device level is generality. Users (especially on clouds) run a range of different file systems, as well as

APSys '17, September 2, 2017, Mumbai, India
© 2017 Association for Computing Machinery.
ACM ISBN 978-1-4503-5197-3/17/09...$15.00
https://doi.org/10.1145/3124680.3124732

databases or other software directly on the raw device [1]. Although some file systems, such as btrfs and zfs [6, 20, 28] include some checksum support, this is only designed to detect block corruption, not to resist a malicious volume. Further, integrity file systems and filesystem centric solutions [6, 20, 28], while very useful in specific applications, are often designed with significant additional compute and network resources in mind [14, 16–18]. For example, some distributed storage systems ensure strong consistency protection, such as fork consistency, in the presence of multiple clients [14, 16–18]. Such properties come at a significant computational and network cost, and are stronger than needed for a typical hybrid cloud, where one assumes that each volume is mounted by a single client at a time.

Most importantly however filesystem techniques do not work for securing boot devices, protecting volume snapshots, and applications optimized to function on top of block devices. Since cloud services expose only a raw block device (such as an EBS volume) to the client VMs, using a filesystem for integrity may introduce unnecessary performance penalties for specific applications. For example, production MySQL databases are routinely deployed on raw disks for increased performance [1]. dm-x can be deployed transparently to a legacy filesystem as well as a databases that runs directly on a raw device [1].

The contributions of this paper are as follows:

- The design and implementation of an efficient, volume-level integrity-preserving device mapper.
- A metadata journaling mechanism with integrity, which endows dm-x with robustness against system crashes and power failures.
- A host of optimizations for efficient, authenticated reads and writes.
- A thorough evaluation on both cloud and local storage, which shows read and write performance comparable to ext4 with metadata-only journaling.

2 RELATED WORK

Block-Level Integrity on Local Storage. The dm-integrity device-mapper [15] uses keyed hash-based message authentication codes (HMACs) to provide transparent read-write block-level cryptographic integrity protection for the underlying block device. Block-granular integrity prevents tampering with individual blocks, but does not prevent reordering blocks within the device.

Hou et al. [13] use a Merkle tree variant, and place subtrees with data on disk to improve locality. However, this is done under certain unrealistic assumptions (e.g., increasing standard disk block size to 536 bytes) and with asynchronous integrity checks, which allow applications to use unverified data.

Heitzmann et al. [11] use authenticated skip list to provide block-level integrity for untrusted storage servers. Although, authenticated skip lists have better asymptotic performance when compared to Merkle trees, [11] does not address the possibility of crashes and inconsistencies. Oprea et al. [23] achieve block-level integrity for cloud-hosted volumes using tweakable block ciphers and an entropy-based mechanism to detect randomness in the contents of a block. However, unlike dm-x, their scheme requires storing hashes of *all* blocks that have high entropy on the client.

File System Integrity. Both btrfs [20] and ZFS [6, 29] use checksums to detect disk block corruption at rest. ZFS also uses Merkle trees for efficiency. Neither are designed to detect tampering by an attacker with physical access (as the hashes are stored on the disk itself), and do not provide volume-level protection.

jVPFS [28] uses Merkle trees and a chained journal to store high-integrity data in a smaller file system within a larger, untrusted file system. Unlike dm-x, a block device mapper designed for bare metal, jVPFS is a nested file system that integrates the Merkle tree with the file system metadata tree and tailors updates to file system semantics.

PFS [27] is an extension of the LFFS log-structured file system [26] to add collision-resistant, chained hashes to ensure integrity of each logged write. This design is particularly elegant in a log-structured file system, as it integrates the hash chains with the log itself, but is inefficient for a typical, tree-structured file system.

Oprea and Reiter describe a MAC-tree – similar to a Merkle tree, but with MACs. They leverage MAC trees in a cryptographic file system, using features such as compressibility to store integrity metadata inline, and thereby improving locality [22].

Some distributed storage systems inherently provide integrity as a means to achieve strong consistency protection (such as fork consistency) in the presence of multiple clients [14, 16–18]. Properties such as fork consistency come at a significant computational and network cost, and are stronger than needed for a typical single-client scenario, as addressed here. Unlike dm-x, filesystem based integrity solutions do not provide block-level integrity, which is needed for certain databases [1].

3 OVERVIEW AND DESIGN

dm-x is a Linux device mapper that provides integrity protection in a filesystem-independent manner. The device mapper layer sits between the file system and the disk, and can transparently add features such as RAID, block caching, and encryption. The device mapper layer is stackable, so features such as encryption or RAID can be easily combined with integrity protection. Figure 1 summarizes the dm-x architecture and on-disk layout.

dm-x is an extension of the dm-verity device mapper [3], extended with additional guarantees and features, including journaling and read-write capability. The dm-verity module provides read-only integrity protection for a block device using a Merkle tree [21] on a separate device; once a block device is *sealed*, the block device cannot be modified again, except by resealing the filesystem—an expensive operation involving reading and rehashing all blocks, even those not in use. The main goal of dm-verity is checking system software

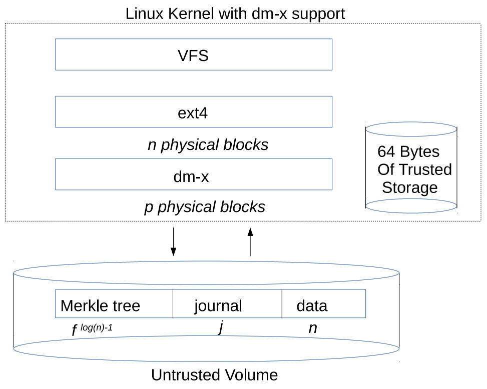


Figure 1: dm-x deployment scenario. dm-x exports a virtual device layer to the higher-level file system. The trusted storage (local disk on client VM, TPM) provides a small amount of trusted storage (64 bytes). The underlying volume itself is untrusted. The on-disk layout is illustrated, not to scale. A Merkle tree and a journal are transparently added to the front of the underlying device.

as part of boot; if system software is updated infrequently and updates are already expensive, this design makes sense. dm-x adds secure, efficient updates to an integrity-checking mapper; increases flexibility in device selection, supports a single device or multiple devices; adds journaling; and extends this benefit to all user and application data.

At a high level, dm-x can efficiently record updates to the Merkle tree by also creating an HMAC-protected journal. In dm-x, the storage device is untrusted; we posit a small amount of trusted storage, such as a trusted local disk for cloud deployment, a TPM chip, a removable storage device (e.g., a USB stick), or a trusted network service. Trusted storage keeps a 32 byte root hash of the Merkle tree and a 32 byte secret key to generate HMACs of the journal.

The dm-x mapper checks disk block integrity on demand – i.e., when a block is read from disk. If the block's hash doesn't match the Merkle tree, the file system receives an I/O error and logs a message to the kernel error log (dmesg). A production system would include a tool to help the user recover the file contents. If a block is not read, it will not be checked; if a user desires a periodic integrity check to detect sector corruption, similar to RAID scrubbing, she need only schedule a cron job that reads the contents of disk.

The dm-x module is about 2.4 kLoC (compared to 700 for the original read-only version). The userspace formatting utility is roughly 650 LoC.

3.1 Threat Model

The primary goal of **dm-x** is to detect tampering with data at rest on untrusted volumes – deployable for both local and cloud hosted storage. The dm-x adversarial model thus includes both local and remote adversaries as we detail below.

Local Adversary. dm-x protects against a local adversary that could have control of a laptop from a user temporarily, such as at a customs checkpoint, and subsequently return it to the user. The attacker does not know the user's login credentials and does not have access to the running system software. An attacker might be able to recover a secret key from the RAM of a running system using a cold boot attack [10]. Fortunately, closing this attack vector is straightforward on SGX-enabled Intel chips [19] and thus are not explicitly detailed in this paper.

Untrusted Remote Storage. dm-x protects against *untrusted* storage servers that can tamper with the stored data (such as selective data removal) and/or mount replay attacks by presenting stale data to the user (client VM) for cloud hosted volumes. Note that in this case, the client VM is trusted.

Small Trusted Storage. dm-x assumes small trusted storage to store secrets (secret keys and Merkle tree root hash, as described above) in the user's possession. The type of storage required depends on the deployment scenario. For example, to protect against a local adversary, dm-x can store secrets in a TPM. For a cloud deployment, since the client machine (VM) is not compromised, it can securely store the secrets locally.

At mount time, dm-x assures that disk contents are the same as the last time the file system was mounted by a trusted OS kernel. Thus, attacks that take control of the operating system via exploitable software are beyond the scope of our threat model, as dm-x cannot distinguish disk write requests issued by a compromised application or kernel module from uncompromised code. In our design, the hardware, BIOS, bootloader, and OS kernel are trusted.

3.2 Storage Organization

The dm-x design transparently allocates a small, logarithmic amount of disk space for its internal bookkeeping, illustrated in Figure 1. To ensure crash consistency between the data and the Merkle tree, the dm-x design includes metadata journaling to record the original hash and the new hash on every write; because the journal and tree are updated in large, contiguous writes, we stored them contiguously at the front. A portion of the Merkle tree itself, as well as the most-recently-accessed data are cached in memory at runtime.

The space overhead of integrity protection is generally low—a function of the size of the data being protected, the block size, the hash function, and the branching factor. In dm-x we use SHA256. Thus, each 4KB disk can hold up to 128 hashes (32B x 128 = 4KB). The Merkle tree in dm-x has a branching factor of 128 to ensure full utilization of the 4KB block size. In a typical use case, say a 2TB commodity drive with a 4KB block size, this requires a few GB for the Merkle tree—less than 0.8%. The journal size is also configurable; for our experiments, a 32 MB journal was sufficient; for comparison, the ext4 default journal size is 128 MB [4]. Moreover, the I/O overhead in terms of extra bytes written to the disk is very small — e.g., for every 1GB written to disk, on an average, we write extra 18.2 MB (1.77%) for journal and Merkle tree updates.

3.3 Efficient, Journaled Writes

The primary challenge in adding write support to dm-x is ensuring crash consistency among data and Merkle tree nodes. For example, in a simple Merkle tree implementation, if the system crashes between updating an interior node and the root, the Merkle tree on disk will fail validation, despite the fact that all data on disk was actually written by trusted software. To recover from crashes during a Merkle tree write, dm-x includes a replay journal. The current journal design went through many iterations. The current design stores the old and new hash of the data present at a given disk sector; dm-x ensures that journal entries are committed to the device before the corresponding data and Merkle tree nodes are written. Thus, after a crash, the data is checked to match one hash value or the other and update the Merkle tree accordingly. We note that dm-x only ensures a consistent state of the Merkle tree and cannot roll back (or roll through) transactions interrupted due to a crash. In fact, since the main purpose of deploying dm-x is integrity protection for filesystem data through the Merkle tree, the responsibily of preventing data loss lies with the filesystem. Fortunately, almost all commonly deployed filesystems, such as ext4 and NTFS, support data journaling, which allows graceful recovery from crashes and prevents data loss. Nonetheless, we consider a version of dm-x with full data journaling as future work. This is to support deployment scenarios where dm-x is deployed without an overlying filesystem, or the filesystem does not support data journaling.

A journal entry is written to disk when a `sync()` is requested, a journal buffer is full, after a maximum period (5 seconds by default), or the system needs to reclaim memory from cached, dirty pages. To improve write performance, dm-x groups dirty page writes, lowering the cost of journaling and giving the I/O scheduler more flexibility in write-back. For every read or write I/O request, the prefetch thread brings into memory all the relevant hashes at different levels, so that the I/O request serving threads are not blocked on I/O.

3.4 Verifying Reads

When the file system issues a block read request, the dm-x module verifies that block's contents hash to the same value as the last write by trusted software.

Merkle tree integrity validation executes from the top-down, as illustrated in Figure 2, with a simple tree with branching factor 2 and depth 3 (dm-x actually uses a branching factor of 128). dm-x reads the root node from disk and hashes its contents, comparing this to the root hash from trusted storage. Each non-leaf tree node stores an array of child node hashes. If a non-leaf node has the correct hash value, the appropriate child hash is selected, and the process iterates until a leaf node is hashed. If any node in the process doesn't hash to a trusted value, an I/O error is returned to the file system and an integrity violation is logged. Because multi-core CPUs are ubiquitous and often under-utilized on modern systems, we use multiple threads to parallelize block hashing, thereby increasing throughput and reducing latency.

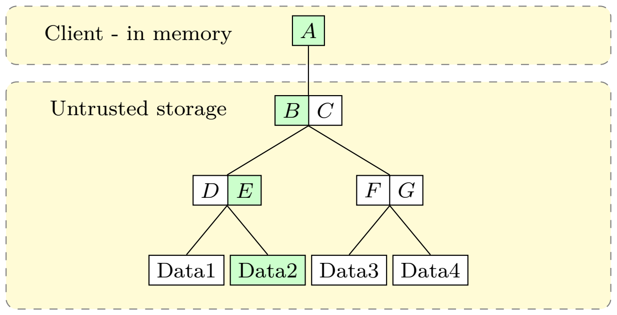


Figure 2: Merkle tree traversal, with fanout of 2. Highlighted path marks traversal to block 2. A is the root hash.

Moreover, for every read request, we first check if the data block is dirty and in memory, in which case, we return the most recent dirty data. If the read request is for a large chunk of data and only part of it is in cache, we break the large request into multiple requests so that we only read the data not present in the cache.

With a cold cache, checking a block requires reading and hashing a logarithmic number of disk blocks. For instance, on a 2 TB disk the Merkle tree would be 5 levels deep, so the first read would read and hash 5 extra blocks.

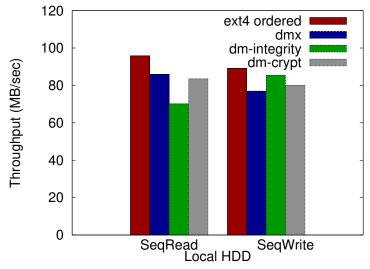
In the common, warm-cache case, reading a block incurs 0–2 extra reads and hashes, depending on the locality of access. dm-x includes a 64 MB cache of verified Merkle tree nodes, organized in a radix tree. In our 2TB disk example, the top 3 layers of the Merkle tree will almost always be in cache, but only a recently used subset of the bottom 2 layers will be cached — most of which are prefetched on the I/O request.

4 INTEGRITY DESPITE CRASHES

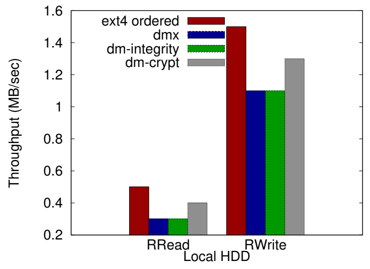
Given that the journal is replayed to the Merkle tree after a crash, the integrity of the journal itself must be assured. The Merkle tree does not protect the journal contents; rather, dm-x uses a secret key (stored in the TPM or on a trusted remote storage) to compute a chained sequence of MACs of the Merkle tree root hashes. In other words, as transactions are applied to the journal, the end of a journal transaction includes a MAC of the previous Merkle tree root hash, and the new root hash.

This design resists journal tampering, as MACs are unforgeable. If any journaled writes are modified, the MAC will not validate. When the journal is checkpointed (i.e., old entries are garbage collected), the MAC of the last transaction is stored in the trusted storage. The adversary can delete entries from the journal, but this is equivalent to the threat of zeroing block contents on the main device or losing power before the data is written to disk; the window of potential data loss can be bounded by more frequent journal checkpointing.

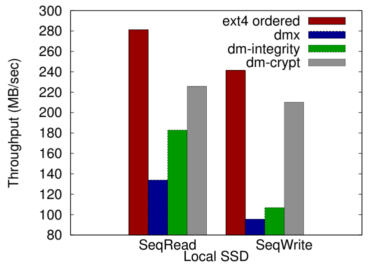
Because the journal entries are currently chained by root hashes, if the file system is ever checkpointed in exactly the same state (i.e., the root hashes match), this creates a cycle



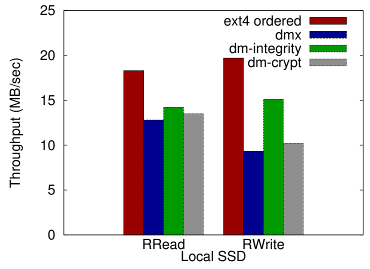
(a) Sequential Access on HDD



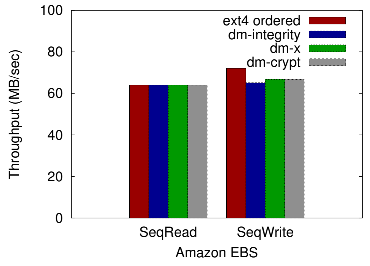
(b) Random Access on HDD



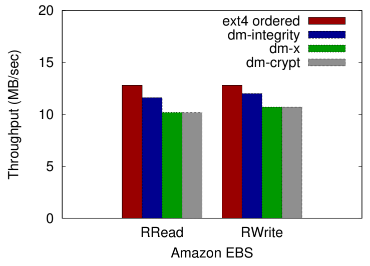
(c) Seqeuntial Access on SSD



(d) Random Access on SSD



(e) Sequential Access on EBS



(f) Random Access on EBS

Figure 3: Throughput comparison of dm-x , dm-integrity, ext4 filesystem in ordered mode and dm-crypt on an HDD, SSD and Amazon EBS volumes. Throughput is measured in MB/s (higher is better). dm-x shows comparable sequential throughput to ext4 and dm-integrity on the HDD. Ext4 outperforms both dm-x and dm-integrity on the SSD although dm-x provides stronger volume-level integrity protection compared to the block-level protection for dm-integrity. For EBS volumes, sequential and random access performances are comparable, as network performance is the limiting factor.

that would allow an attacker to replay an old journal entry. Although a repeated root hash is highly improbable with a collision-resistant hash function, this can be fixed by chaining journal entries by the MAC of the previous entry, and storing the MAC of the last checkpointed transaction in trusted storage. This change makes the probability of cycles in the journal's MAC chain negligible, even if the root hash value repeats.

5 EVALUATION

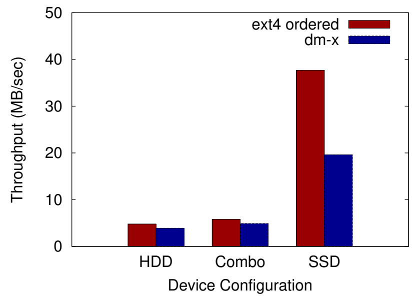
We measure the performance of dm-x on both local and cloud storage devices. For the local storage scenario, dm-x was benchmarked using Ubuntu 14.04, Linux kernel 3.16.0. All experimental results were collected on a machine with a dual-core Intel i7-3520M CPU running at 2.90GHz and 6GB RAM. For the cloud storage benchmark, we used Amazon t2.medium EC2 instances with 4GB of memory running Ubuntu 14.04, Linux kernel 3.16.0. The storage devices of choice were a 500 GB, 7200 RPM ATA Hitachi HDD, 1 TB SanDisk UltraII SSD, and a 20GB Amazon EBS "general purpose SSD (gp2)" with a max throughput of 160MB/s per volume.

Local Microbenchmarks. Local microbenchmarks were performed with sequential reads and writes of a 10 GB file, and random 1 GB of 4KB reads and writes spread across a 10GB file. dm-x was compared with dm-integrity [15], dm-crypt, and ext4 in ordered mode on the raw volume. dm-crypt is a commonly deployed disk encryption tool for the Linux kernel. Although, confidentiality is orthogonal to the goals of this paper, the benchmarks demonstrate the costs of achieving security at the block device-level.

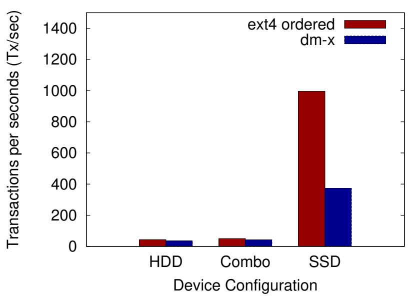
Sequential reads are comparable in most cases (Figure 3(a)), except for lower throughput on an SSD (Figure 3(c)). This is largely due to the overhead introduced by hashing the additional Merkle tree data. For example, to read or write 10 GB of data, with 4KB block size and 128 hash entries in each block, 20480 Merkle tree leaves (around 84 MB of data) and their parents needs to be read from the disk. Thus, almost 170 MB of I/O requests are made in order to read the corresponding Merkle tree nodes. Further, the Merkle tree verification requires hashing all the nodes along a path to the leaf. Thus, with depth of 5 (for a 1TB disk), almost 420 MB of Merkle tree nodes needs to be hashed. With low access latencies on SSDs, the cost of hashing dominates the cost for performing the additional I/O.

We believe that read performance can be improved on an SSD case with better read-ahead and eliminating lock contention with finer-grained synchronization.

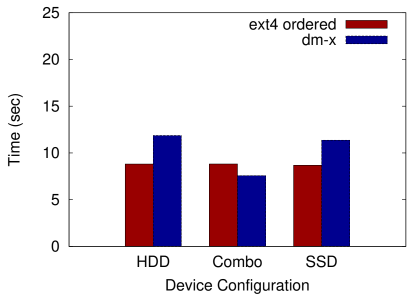
dm-x performs comparably with dm-integrity on the HDD although dm-x provides a stronger volume level protection using the Merkle tree. In contrast, dm-integrity only uses HMAC to provide block-level integrity. For the SSD, dm-integrity outperforms dm-x as it needs to only verify the



(a) Varmail



(b) Pgbench



(c) Untar Linux

Figure 4: Application benchmark comparisons of ext4 file system on an HDD, SSD, and combination of HDD+SSD (Combo), using ordered journal mode and dm-x. Varmail and pgbench measure throughput (higher is better), and untar measures latency (lower is better).

HMAC for each block but is still appreciably slower than ext4 on the raw disk. In fact, the overhead of hashing is evident also in this case.

Storing the Merkle tree on an SSD can significantly improve overall write throughput, even when data is stored on an HDD. This is especially exciting, since an SSD could be used in combination with a RAID array of rotational disks to offer even higher total bandwidth.

CPU utilization. To evaluate CPU overhead, we benchmark the SSD, which keeps the CPU busier. For sequential read and write, baseline ext4 uses 20% and 30% of one core, respectively. When adding integrity, CPU utilization increases to 60% of all four cores for reads and 40% of all cores for writes. This increase can be straightforwardly attributed to hashing Merkle tree nodes in parallel, using all the available cores. Further, a higher read throughput for the SSD entails higher CPU utilization for evaluating hashes of blocks read from the disk, compared to lower CPU utilization for a lower write throughput. In fact, sequential accesses on local SSDs demonstrate the maximum CPU utilization for dm-x. For random I/O on SSDs and both sequential and random I/O on HDDs, the CPU overheads are significantly lower as dm-x accesses are I/O bound. For CPU constrained systems, the number of parallel threads can be optimized while trading off throughput over resource utilization.

Cloud Benchmark. We measure sequential reads and writes of a 8 GB file (for 4GB RAM size), and random 1 GB of 4KB reads and writes spread across a 8GB file on the Amazon t2.medium EBS volume. The network link bandwidth is 25MB/s, as measured by *iperf* [5]. This corresponds to typical network deployment scenarios with medium to high network latency. Figure 3(e) and 3(f) reports throughput in MB/s.

Ext4 with dm-x shows comparable performance to ext4 in ordered data mode in the medium to high network latency environment, highlighting its practicality for ensuring integrity of cloud hosted volumes. Further, as shown in the local benchmarks above, dm-x performs reasonably in low latency environments and is thus suitable for deployment even in the case of low latency network storage scenarios (such as a SAN setup).

5.1 Application Benchmarks

Untar linux. As a simple application benchmark, we measure the time to untar the Linux 3.17.4 kernel (Figure 4). To demonstrate a possible deployment scenario, we additionally benchmark dm-x in "combo" mode – the Merkle tree is stored on the SSD while the data is stored on the HDD. The execution time is 24-26% slower than ordered journaling, except on the combo case, where all data points are relatively close.

Varmail. The varmail personality from Filebench [8] reads big files and forces a `sync` between relatively small writes. dm-x performs comparably with ext4 on an HDD but becomes worse for the SSD — which is not surprising given the sequential read performance of dm-x on the SSD.

PostGreSQL. For PostGreSQL pgbench, we used scaling factor 300, 4 connections, and 2 threads, which ran for 10 minutes each. This setup measures the disk performance avoiding the effect of buffers and cache on the performance. This case illustrates the benefits of group commit, as well as buffering random writes, in the dm-x journal, yielding write throughput better than ordered mode on the HDD or SSD.

Finally, in all cases, the combo out-performs the HDD alone, in several cases by a considerable margin. This is because we accelerate all synchronous writes on the SSD, and all writes to the HDD are asynchronous. Thus, this model could potentially yield very good performance on a dual SSD/HDD setup.

6 CONCLUSION

Volume-level integrity protection is essential for devices and (cloud) services that are not always under the user's control. In this paper we show that in-band, writable, cryptographically-strong integrity protections are feasible when implemented at the block device layer, and provide broad support and compatibility with legacy file systems, as well as additional features such as metadata journaling and crash recovery. The open-source implementation of dm-x is available at https://github.com/anrinch/dmx.

7 ACKNOWLEDGEMENT

The authors would like to acknowledge support from the National Science Foundation (awards 1526707, 1526102, 1161541, 1318572, 1405641 and 1228839), Office of Naval Research and VMware. We also thank our shepherd, Wenguang Chen and the anonymous reviewers for their valuable suggestions on improving the paper.

REFERENCES

[1] *Using Raw Disk Partitions for the System Tablespace in MySQL.* http://dev.mysql.com/doc/refman/5.7/en/innodb-raw-devices.html.

[2] 2017. Amazon Virtual Private Cloud. ""https://aws.amazon.com/vpc/"". (2017).

[3] 2017. dm-verity. https://code.google.com/p/cryptsetup/wiki/DMVerity. (2017).

[4] 2017. ext4 Disk Layout. https://ext4.wiki.kernel.org/index.php/Ext4_Disk_Layout. (2017).

[5] 2017. iperf. "https://iperf.fr/". (2017).

[6] Jeff Bonwick and B. Moore. 2008. ZFS: The Last Word in File Systems. http://opensolaris.org/os/community/zfs/docs/zfslast.pdf. (2008).

[7] James Butler and Greg Hoglund. 2004. VICE–catch the hookers. *Black Hat USA* 61 (2004), 17–35.

[8] filebench 2017. FileBench. http://sourceforge.net/projects/filebench/. (2017).

[9] George Leopold, Enterprise Tech. 2015. Verizon Survey: Hybrid Cloud Now Mainstream. http://www.enterprisetech.com/2015/11/09/verizon-survey-hybrid-cloud-now-mainstream/. (2015).

[10] J. Alex Halderman, Seth D. Schoen, Nadia Heninger, William Clarkson, William Paul, Joseph A. Calandrino, Ariel J. Feldman, Jacob Appelbaum, and Edward W. Felten. 2009. Lest We Remember: Cold-boot Attacks on Encryption Keys. *Commun. ACM* 52, 5 (May 2009), 91–98. https://doi.org/10.1145/1506409.1506429

[11] Alexander Heitzmann, Bernardo Palazzi, Charalampos Papamanthou, and Roberto Tamassia. 2008. Efficient Integrity Checking of Untrusted Network Storage. In *Proceedings of the 4th ACM International Workshop on Storage Security and Survivability (StorageSS '08)*. ACM, New York, NY, USA, 43–54. https://doi.org/10.1145/1456469.1456479

[12] Owen S. Hofmann, Alan M. Dunn, Sangman Kim, Indrajit Roy, and Emmett Witchel. 2011. Ensuring operating system kernel integrity with OSck. 279–290.

[13] Fangyong Hou, Dawu Gu, Nong Xiao, Fang Liu, and Hongjun He. 2009. Performance and Consistency Improvements of Hash Tree Based Disk Storage Protection. In *Networking, Architecture, and Storage, 2009. NAS 2009. IEEE International Conference on.* 51–56. https://doi.org/10.1109/NAS.2009.15

[14] Michael Kaminsky, George Savvides, David Mazieres, and M. Frans Kaashoek. 2003. Decentralized user authentication in a global file system. 60–73.

[15] Dmitry Kasatkin. 2013. dm-integrity: integrity protection device-mapper target. http://lwn.net/Articles/533558/. (2013).

[16] Jinyuan Li, Maxwell Krohn, David Mazières, and Dennis Shasha. 2004. Secure untrusted data repository (SUNDR). 9–9.

[17] Prince Mahajan, Srinath Setty, Sangmin Lee, Allen Clement, Lorenzo Alvisi, Mike Dahlin, and Michael Walfish. 2011. Depot: Cloud Storage with Minimal Trust. 29, 4, Article 12 (Dec. 2011), 38 pages. https://doi.org/10.1145/2063509.2063512

[18] David Mazières, Michael Kaminsky, M. Frans Kaashoek, and Emmett Witchel. 1999. Separating key management from file system security. 124–139.

[19] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V. Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R. Savagaonkar. 2013. Innovative Instructions and Software Model for Isolated Execution. In *Workshop on Hardware and Architectural Support for Security and Privacy.* ACM, Article 10, 10:1–10:1 pages.

[20] Amanda McPherson. 2009. A Conversation with Chris Mason on BTRfs: the next generation file system for Linux. http://www.linuxfoundation.org/news-media/blogs/browse/2009/06/conversation-chris-mason-btrfs-next-generation-file-system-linux. (2009).

[21] Ralph C. Merkle. 1988. A Digital Signature Based on a Conventional Encryption Function. In *A Conference on the Theory and Applications of Cryptographic Techniques on Advances in Cryptology (CRYPTO '87)*. Springer-Verlag, London, UK, UK, 369–378. http://dl.acm.org/citation.cfm?id=646752.704751

[22] Alina Oprea and Michael K. Reiter. 2007. Integrity Checking in Cryptographic File Systems with Constant Trusted Storage. In *Proceedings of 16th USENIX Security Symposium on USENIX Security Symposium (SS'07)*. USENIX Association, Berkeley, CA, USA, Article 13, 16 pages. http://dl.acm.org/citation.cfm?id=1362903.1362916

[23] Alina Oprea, Michael K. Reiter, and Ke Yang. 2005. Space-Efficient Block Storage Integrity. In *In Proc. of NDSS '05*.

[24] Benny Pinkas and Tzachy Reinman. 2010. Oblivious RAM Revisited. In *Proceedings of the 30th Annual Conference on Advances in Cryptology (CRYPTO'10)*. Springer-Verlag, Berlin, Heidelberg, 502–519. http://dl.acm.org/citation.cfm?id=1881412.1881447

[25] PulserG2 anfd XDA. 2015. A Look at Marshmallow Root & Verity Complications. http://www.xda-developers.com/a-look-at-marshmallow-root-verity-complications/. (2015).

[26] Margo I. Seltzer, Gregory R. Ganger, M. Kirk McKusick, Keith A. Smith, Craig A. N. Soules, and Christopher A. Stein. 2000. Journaling Versus Soft Updates: Asynchronous Meta-data Protection in File Systems. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference (ATEC '00)*. USENIX Association, Berkeley, CA, USA, 6–6. http://dl.acm.org/citation.cfm?id=1267724.1267730

[27] Christopher A. Stein, John H. Howard, and Margo I. Seltzer. 2001. Unifying File System Protection. In *Proceedings of the General Track: 2001 USENIX Annual Technical Conference, June 25-30, 2001, Boston, Massachusetts, USA*. 79–90. http://www.usenix.org/publications/library/proceedings/usenix01/stein.html

[28] Carsten Weinhold and Hermann Härtig. 2011. jVPFS: Adding Robustness to a Secure Stacked File System with Untrusted Local Storage Components. In *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference (USENIX-ATC'11)*. USENIX Association, Berkeley, CA, USA, 32–32. http://dl.acm.org/citation.cfm?id=2002181.2002213

[29] Yupu Zhang, Abhishek Rajimwale, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2010. End-to-end Data Integrity for File Systems: A ZFS Case Study. In *Proceedings of the 8th USENIX Conference on File and Storage Technologies (FAST'10)*. USENIX Association, Berkeley, CA, USA, 3–3. http://dl.acm.org/citation.cfm?id=1855511.1855514