2021 IEEE Intl Conf on Dependable, Autonomic and Secure Computing, Intl Conf on Pervasive Intelligence and Computing, Intl Conf on Cloud and Big Data Computing, Intl Conf on Cyber Science and Technology Congress

2021 IEEE Intl Conf on Dependable, Autonomic and Secure Computing, Intl Conf on Pervasive Intelligence and Computing, Intl Conf on Cloud and Big Data Computing, Intl Conf on Cyber Science and Technology Congress (DASC/PiCom/CBDCom/CyberSciTech) | 978-1-6654-2174-4/21/$31.00 ©2021 IEEE | DOI: 10.1109/DASC-PICOM-CBDCOM-CYBERSCITECH52372.2021.00026

Prevention of a DoS Attack with Copy-on-write in the Overlay Filesystem

Hirofumi Satou
Kyushu Institute of Technology
satou@ksl.ci.kyutech.ac.jp

Kenichi Kourai
Kyushu Institute of Technology
kourai@csn.kyutech.ac.jp

***Abstract*—Recently, containers are widely used for lightweight virtualization. A container usually uses a disk image that stacks a thin writable layer on top of a read-only image layer. For this layering, a filesystem called OverlayFS is often used. To modify a file in the read-only lower layer, OverlayFS first copies the entire file to the upper layer and then writes requested data to it. This *copy-on-write* suspends a container for a long time and consumes the disk space of the upper layer when the size of the target file is large. If large files are intentionally modified by attackers, a potential denial-of-service (DoS) attack can be mounted. This paper proposes a new filesystem, called *TranslayFS*, based on OverlayFS to prevent this type of DoS attack. TranslayFS creates only a special file called a *sparse file* in the upper layer when a container modifies a file in the lower layer for the first time. Using this file, it holds only modified file data in the upper layer without copying the entire file. It returns the modified part of the file from the upper layer and the unmodified part from the lower layer. We have implemented TranslayFS in the Linux kernel and confirmed that TranslayFS could dramatically reduce the latency in the first write to a file, so that the DoS attack was not possible.**

***Index Terms*—DoS attack, containers, disk image, filesystem, copy-on-write**

1. Introduction

Recently, containers like Docker [1] are widely used for lightweight virtualization. The disk image of a container is usually created by stacking a thin writable layer on top of a read-only image layer. An image layer provides the base system for container execution, while a writable layer stores files created by a container. To enable this layering, a filesystem called the Overlay filesystem (OverlayFS) [2] is often used. In OverlayFS, an image layer is called a *lower layer*, while a writable layer is called an *upper layer*. When a container reads a file, OverlayFS returns the data of a file in the upper layer if the requested file exists; otherwise, it returns the data of a file in the lower layer. In contrast, it always writes data to files in the writable upper layer.

OverlayFS uses the *copy-on-write* mechanism for the first write to a file. When a container modifies a file whose real entity exists in the lower layer, OverlayFS first copies the entire file to the upper layer using the *copy-up* operation. This is because it cannot overwrite the file in the read-only lower layer. Then, it writes requested data to the file created in the upper layer. If the size of a modified file is large, this copy-up operation increases the delay of a write operation when a container modifies the file for the first time. Since the container is suspended until the write operation is completed, e.g., 51 seconds for a 10-GB file in our experiment, this can largely affect the performance of the container. If this mechanism is maliciously used by attackers, a potential denial-of-service (DoS) attack can be mounted. Attackers could suspend a service in a container for a long time by letting it modify part of a large file. They could also make the upper layer out of space by copying large files if the size of the upper layer is limited by quota.

To prevent this type of DoS attack, this paper proposes a new filesystem called *TranslayFS*, which is based on OverlayFS. Unlike OverlayFS, TranslayFS holds only modified file data in the upper layer. It does not copy the entire file to the upper layer when a container modifies a file existing in the lower layer for the first time. Therefore, it can eliminate the root cause of the DoS attack, i.e., time- and space-consuming copy-up of the entire file. To efficiently manage only modified file data in the upper layer, TranslayFS uses a special file called a *sparse file*. This file enables scattered file data to be stored in the upper layer without a large modification to OverlayFS. When a container reads a modified file, TranslayFS returns file data in the upper layer if that data exists; otherwise, it returns file data in the lower layer.

We have implemented TranslayFS in Linux kernel 4.4. In Linux, a sparse file consists of 4-KB data blocks including real data and holes including no data. Therefore, TranslayFS copies only an unmodified part of a block from the lower to upper layer on the first modification to the block. It detects whether a file block is a hole or not to determine which layer it needs to access. We conducted several experiments to examine the performance of TranslayFS, compared with OverlayFS. It was shown that TranslayFS could complete the first one-byte write to a 10-GB file in the lower layer only in 339 μs. Instead, it could suffer from the delay by copying an unmodified part of a block for each write, but this delay was only 89 μs even for one-byte write. As a result, TranslayFS could prevent the DoS attack with copy-on-write.

The organization of this paper is as follows. Section 2

978-1-6654-2174-4/21/$31.00 ©2021 IEEE
DOI 10.1109/DASC-PICom-CBDCom-CyberSciTech52372.2021.00026

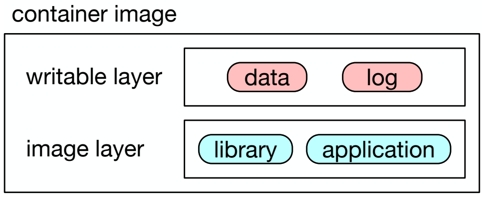


Figure 1: A container image with two layers.

describes a DoS attack with copy-on-write in OverlayFS. Section 3 proposes TranslayFS to prevent this type of DoS attack and Section 4 explains its implementation. Section 5 shows the experimental results, compared with OverlayFS. Section 6 describes related work and Section 7 concludes this paper.

2. DoS Attack with Copy-on-write

The disk image of a container in Docker is created by stacking a thin writable layer on top of a read-only image layer, as illustrated in Fig. 1. An image layer contains the base system such as libraries and applications commonly used by multiple containers. Since it cannot be modified by containers, a thin writable image is prepared for each container. It contains data and log files created and applications installed at runtime by a container. It can also contain custom libraries and applications that are not contained in the base image. A different writable layer can be further stacked on top of the existing stack of a writable layer and an image layer.

To enable this layering in a container image, a filesystem called OverlayFS is often used as a storage driver in Docker. It can combine two filesystems into one. Docker can use various storage drivers such as AUFS [3], ZFS [4], Btrfs [5], and devicemapper [6]. Among them, OverlayFS has many advantages and is preferred in Docker. Since it is simpler, the performance is better. It is supported by default in all the Linux distributions without any configuration and therefore is easy to use. In OverlayFS, a read-only image layer is called a lower layer, while a writable layer is called an upper layer.

OverlayFS handles a read request to a file from a container in a different manner, according to which layer an actually accessed file is located in, as illustrated in Fig. 2. When a read request is for a file whose real entity exists in either the lower layer (file 1) or the upper layer (file 2), OverlayFS returns file data in that layer to the container. As such, the container can read files in the base system from the lower layer and those created at runtime from the upper layer. For a file existing in both layers (file 3), it returns file data in the upper layer. In this case, the file in the lower layer is hidden from the container. After a container creates a new file with the same name, it can read that file.

On the other hand, OverlayFS handles a write request to a file differently, as illustrated in Fig. 3. When a write request is for a file existing only in the upper layer (file 4) or both layers (file 5), OverlayFS modifies the file in the upper layer. For a file existing only in the lower layer (file 6), it uses the *copy-on-write* mechanism. Specifically, it first copies the file to the upper layer because it cannot modify the file in the read-only lower layer. This operation is called *copy-up*. Then, OverlayFS modifies the copied file in the upper layer. After that, the entity of the file exists in both layers like file 5.

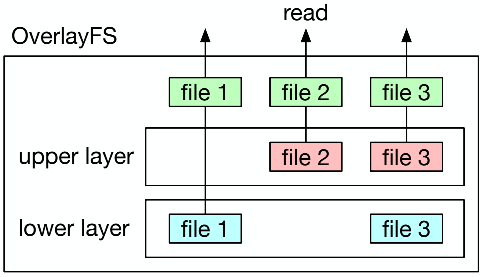


Figure 2: Read operations in OverlayFS.

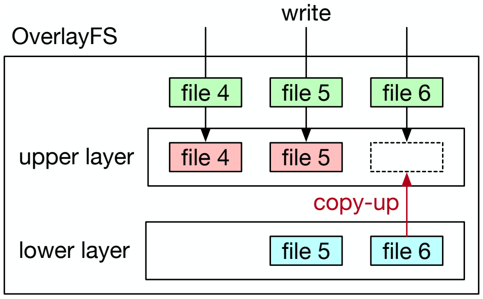


Figure 3: Write operations in OverlayFS.

As such, OverlayFS suffers from the delay due to this copy-up operation whenever a container modifies a file existing only in the lower layer for the first time. Since it always copies the entire file regardless of the size of modified data, this delay increases as the file size becomes larger. For example, a database file may become several giga-bytes. Worse, OverlayFS synchronously writes back the copied file to a disk. While OverlayFS copies a file from the lower to upper layer, the container that requests the file write is suspended until the copy-up operation completes. In addition, the copied file in the upper layer consumes the disk space assigned to the upper layer although its data is almost the same as the original one in the lower layer in general.

If attackers can intentionally cause this copy-on-write to files in a container, a DoS attack is possible against the container. Attackers could suspend a service in a container for a long time by letting it modify part of a large file. They can cause the copy-up operation once for each file existing only in the lower layer. If a container is configured for auto-scaling in a cloud, attackers can send many requests to the container so that the container is automatically scaled out. After that, they could cause copy-on-write again in the newly created containers to suspend the services. If the size

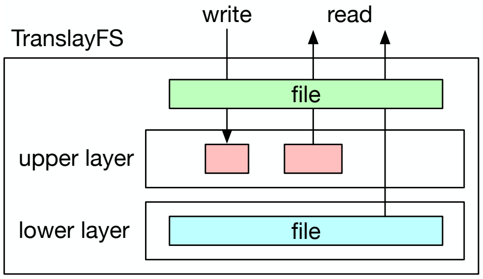


Figure 4: Read and write operations to a file in the lower layer in TranslayFS.

of the upper layer is limited by quota, a container could not write any data after many files are copied from the lower layer.

3. TranslayFS

To prevent this type of DoS attack with copy-on-write in OverlayFS, this paper proposes a new filesystem called TranslayFS. This file system is based on OverlayFS but eliminates time- and space-consuming copy-on-write. When a container writes data to a file whose real entity exists only in the lower layer, TranslayFS stores only the written data in the upper layer, as illustrated in Fig. 4. Unlike OverlayFS, it does not copy the entire file from the lower to upper layer by the copy-up operation to modify a file in the read-only lower layer. When a container modifies a previously modified part of the file again, TranslayFS modifies the data stored in the upper layer.

Using this lightweight write operation without copy-on-write, TranslayFS can decrease the delay caused by the copy-up operation. It can complete the write operation in the time proportional only to the size of written data even when a container modifies a large file existing in the lower layer. This can prevent attackers from suspending a container for a long time by a simple write to part of a large file. Attackers cannot amplify their small write request to the copy of a large file to mount a DoS attack. In addition, TranslayFS can save the disk space assigned to the upper layer because the upper layer holds only a modified part of a file. Attackers would have to write as much data as the size of the upper layer to make the upper layer out of space.

After a container modifies a file existing in the lower layer, TranslayFS handles read requests to that file by accessing both layers, as illustrated in Fig. 4. Since the upper layer holds only a modified part, TranslayFS merges file data in both layers and returns it to the container. Specifically, it returns file data held in the upper layer if a read request is to a modified part; otherwise, it returns file data held in the lower layer. This mechanism is largely different from OverlayFS. OverlayFS merges the upper and lower filesystems in a file granularity. It accesses either layer for each file and returns its file data.

TranslayFS manages a modified part of a file in a simple manner. To store only a modified part, the upper layer needs a database in general. This increases the complexity of TranslayFS, compared with OverlayFS, which simply modifies the file copied to the upper layer. Such complexity leads to less reliability. For a simpler way, TranslayFS uses a special file called a *sparse file* to store a modified part. A sparse file can hold real data only in part of a file, as illustrated in Fig. 5. Upon the first write, TranslayFS creates a sparse file in the upper layer and writes only modified data at the corresponding offset of that file. The part that contains no real data in a sparse file is called a *hole*, where disk space is not allocated. Using a sparse file, TranslayFS can quickly create a file in the upper layer because that file consists of one big hole at first. In addition, it can manage multiple modified data as only one file.

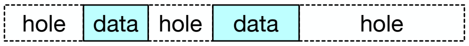


Figure 5: A sparse file with holes in the upper layer.

4. Implementation

We have implemented TranslayFS in Linux kernel 4.4. In this section, we first describe the implementation details of OverlayFS, which is used as the basis of TranslayFS. Then, we describe the implementation of TranslayFS.

4.1. Detailed Behavior of OverlayFS

Upon opening a file, OverlayFS determines which file is accessed in the upper or lower layer. When a container opens a file in a read-only mode, OverlayFS first checks the upper layer and opens a file in that layer if that file exists; otherwise, it opens a file in the lower layer. When a container opens a file in a writable mode, OverlayFS also opens a file in the upper layer if that file exists in that layer. If the file does not exist, however, OverlayFS first copies a file in the lower layer to the upper layer by the copy-up operation and opens the copied file. If there is not the directory where the copied file is stored in the upper layer, OverlayFS first creates that directory and, if necessary, its parent directories. Even if a container does not modify that file, the entire file is copied by simply opening the file in a writable mode.

After a container opens a file in either the upper or lower layer, it always accesses the opened file. Upon file reads and writes, a container bypasses OverlayFS and directly accesses either the upper or lower filesystem. Therefore, no overhead is imposed by file reads or writes.

4.2. File Open in TranslayFS

Unlike OverlayFS, TranslayFS does not copy a file from the lower to upper layer at the time of file open even when a file existing in the lower layer is opened in a writable mode. This increases the performance of file open dramatically. No file is created in the upper layer unless a container actually writes data to the opened file by the write operation.

In addition, TranslayFS does not determine which file is accessed in the upper or lower layer at the time of file open. Instead, it manages files in both the upper and lower layer if any. When a file exists in either layer, TranslayFS opens only one file like OverlayFS. It opens two files when files exist in both layers. This file open is done in the same manner regardless of opening a file in a read-only or writable mode.

As such, TranslayFS needs to open a file in the read-only lower layer even when a container opens a file in a writable mode. This does not happen in OverlayFS because OverlayFS opens a file copied to the writable upper layer. However, TranslayFS cannot open that file in a writable mode because the lower filesystem allows only read-only access. Therefore, it removes the flag for a writable mode from the ones specified for file open before opening the file in the lower layer. Specifically, it removes the O_WRONLY, O_RDWR, O_APPEND, O_TRUNC, and O_CREAT flags. The first three flags are used to specify various writable modes. O_TRUNC is used to truncate the file size to zero. O_CREAT is used to create a new file if the file does not exist.

4.3. File Write in TranslayFS

When a container writes data to a file existing only in the lower layer for the first time, TranslayFS creates a sparse file in the upper layer. That file has completely the same attributes as the file in the lower layer. To create a sparse file that contains no real data, TranslayFS first creates a file whose size is zero and then extends its size to the same size as the file in the lower layer. Then, it opens the created file and manages that file with the already opened file in the lower layer.

TranslayFS manages modified data of the file in the lower layer by the 4-KB block, as illustrated in Fig. 6. This is because the minimum size of a hole in a sparse file is 4 KB in Linux. Upon a file write, TranslayFS calculates the first and last block numbers from the specified file offset and written size. For each block, it checks whether a block is a hole or not, as described in Section 4.5. For a non-hole block, it writes data to the sparse file in the same manner as a normal file. A non-hole block means that the block has been already modified in the past.

For a hole, on the other hand, TranslayFS simply writes data to the sparse file if a container modifies the entire block like block 1 in Fig. 6. However, TranslayFS needs a partial copy-up operation if a container modifies only part of a block like block 2 and 3. It copies an unmodified part from a block in the lower layer to the sparse file. When a container modifies the middle part like block 3, TranslayFS needs two copy-up operations. This partial copy-up operation is performed only to the first and last blocks per file write at most. The other intermediate blocks are never copied from the lower layer. In addition, the partial copy-up operation copies less than 4095-byte data at a time unlike OverlayFS, which copies the entire file. Therefore, it does not suspend a container for a long time.

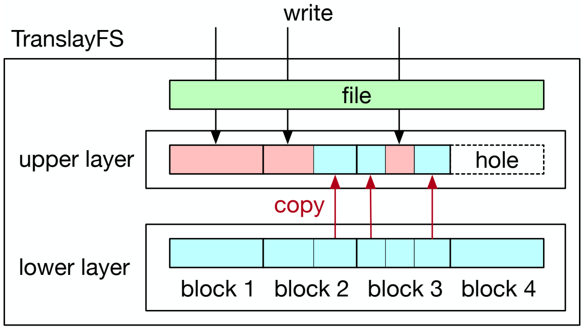


Figure 6: Write operations in TranslayFS.

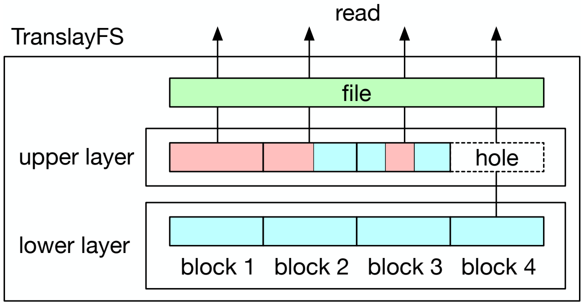


Figure 7: Read operations in TranslayFS.

As such, TranslayFS manages both layers when a container modifies a file. When a container writes data to a file whose real entity is a normal file existing in the upper layer, TranslayFS manages only the upper layer. In any case, it finally redirects the write operation to the upper filesystem. In OverlayFS, in contrast, a container can directly write data to a file in the upper layer after it has opened that file. Therefore, TranslayFS suffers from the slight overhead of this redirection to the upper filesystem.

4.4. File Read in TranslayFS

When a container reads data from a file whose real entity is a sparse file existing in the upper layer, TranslayFS reads data from both layers by the block. First, it calculates the first and last block numbers from the specified file offset and read size. For each block, it checks whether a file block of the sparse file is a hole or not. For a non-hole block like block 1–3 in Fig. 7, it reads data from the file block in the upper layer. For a hole like block 4, it reads data from the file block in the lower layer. Since each block of the sparse file contains complete data, TranslayFS does not need to merge data of blocks in both layers. It accesses only one block in either layer unlike a file write.

As such, TranslayFS manages both layers after a container modified a file. When a container reads data from a file existing in either layer, TranslayFS manages only one layer. In any case, it finally redirects the read operation to the upper or lower filesystem. Therefore, TranslayFS also suffers from the slight overhead of this redirection for the read operation.

4.5. Hole Detection

Linux provides three methods for detecting a hole in a sparse file. First, the FIBMAP ioctl enables users to translate a logical block number of a file into the corresponding physical block number of a disk. If the obtained physical block number is zero, that block contains no real data, i.e., a hole. This ioctl system call requires the root privilege. Second, the FIEMAP ioctl enables users to obtain a list of file blocks that contain real data. The blocks that are not included in the obtained list mean holes. This system call can obtain information on the entire file efficiently and does not require the root privilege. Third, the lseek system call enables users to search for the next hole using the SEEK_HOLE option or the next block that contains real data using the SEEK_DATA option.

TranslayFS uses the first method because it needs to check whether the block specified by its logical number is a hole or not. Since TranslayFS runs in the Linux kernel, it directly invokes the kernel-level bmap function, instead of issuing the ioctl system call. This function translates a logical block number into a physical one in the upper filesystem.

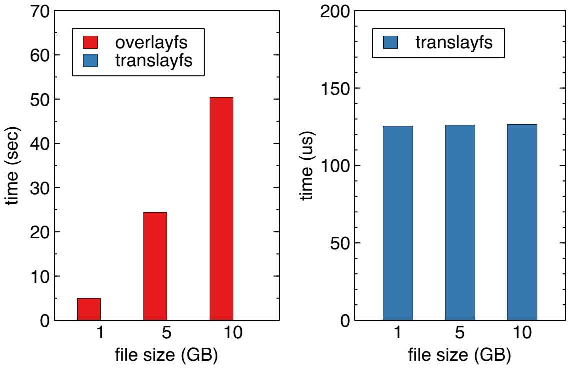
5. Experiments

We conducted several experiments to show that TranslayFS could prevent the DoS attack with copy-on-write. In addition, we compared the performance of TranslayFS with that of OverlayFS. We used a PC with an Intel Core i7-3770 processor, 8 GB of memory, a SATA3 128 GB HDD. We used Linux 4.4.0 and Docker 1.13.1. We used the ext4 filesystem as the upper and lower filesystems in TranslayFS. In these experiments, we used three files of 1 GB, 5 GB, and 10 GB.

5.1. Latency of File Open/Close

We opened a file existing only in the lower layer in a writable mode and just closed it. Then, we measured the time needed for these operations. Fig. 8(a) shows the results for OverlayFS and TranslayFS when we opened three files with different sizes. Since the latency in TranslayFS was much shorter, we also show only the results for TranslayFS in Fig. 8(b). In OverlayFS, the latency was proportional to the file size and it took 50 seconds for the file of 10 GB. This is because OverlayFS copied the entire file to the upper layer. In TranslayFS, in contrast, the latency was only 125–127 μs and almost did not depend on the file size. This dramatic performance improvement comes from the fact that TranslayFS never copies the entire file at any time. These results show that TranslayFS can prevent the DoS attack with copy-on-write.

Next, we opened a file existing only in the lower layer in a read-only mode and just closed it. Fig. 9 shows the time needed for that. The results were similar between OverlayFS and TranslayFS. This is because both filesystems opened only a file in the lower layer. This latency was almost the same as that of the file open in a writable mode in TranslayFS. Note that the latency in TranslayFS was 11–12 μs shorter than in OverlayFS. This comes from the difference in the open operation between OverlayFS and TranslayFS.



(a) Comparison (b) TranslayFS only

Figure 8: The latency of file open in a writable mode.

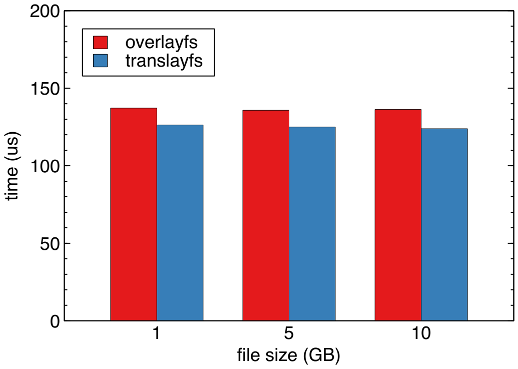
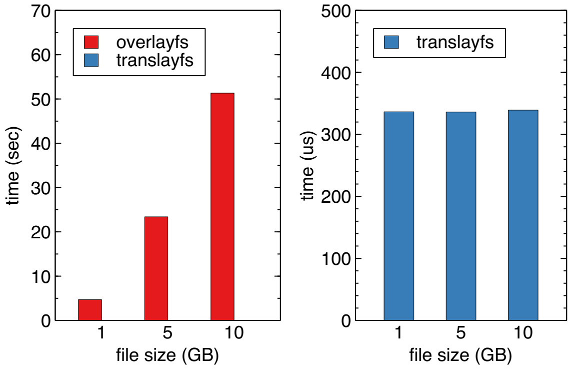


Figure 9: The latency of file open in a read-only mode.

5.2. Latency of File Write

We opened a file existing only in the lower layer, wrote one-byte data to that file, and closed it. Fig. 10 shows the time needed for these three file operations. In OverlayFS, these results were similar to Fig. 8 because file open was a bottleneck. It took 51 seconds to write only one byte to a 10-GB file. In TranslayFS, it took only 336–339 μs regardless of the file size. Among the three file operations, the write operation needed 210–213 μs. This time includes the allocation of a new data block in the sparse file and the copy of 4095-byte data from the corresponding file block in the lower layer as well as the first one-byte write.

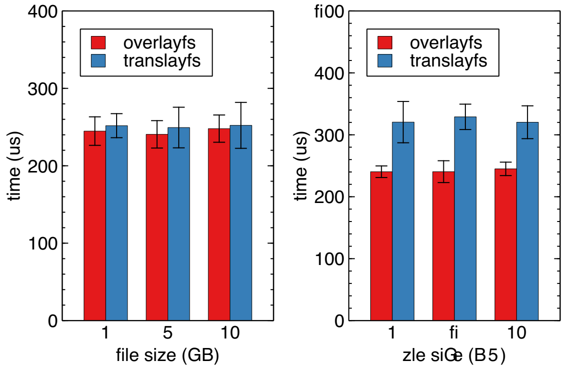
Next, we measured the time for the second one-byte write to the same or a different file block. Fig. 11(a)



(a) Comparison

(b) TranslayFS only

Figure 10: The latency of the first write.



(a) Same block

(b) Different block

Figure 11: The latency of the second write.

shows the results for the second write to the same block. TranslayFS took 4–9 μs longer time than OverlayFS. This is because TranslayFS needed to check whether the accessed block is a hole or not and redirect file access to the upper filesystem. Fig. 11(b) shows the results for the second write to a different block. In this case, TranslayFS took 75–89 μs longer time. Since it had to copy 4095-byte data of a file block in the lower layer to the upper layer, it suffered from the copy overhead. However, this overhead is too small to mount a DoS attack.

Finally, we opened a file existing only in the upper layer and measured the time for the first one-byte write. As shown in Fig. 12, TranslayFS took 8–9 μs longer time than OverlayFS. Since this overhead comes from the hole detection and the access redirection in TranslayFS, it was similar to the overhead of the second write to the same file block in Fig. 11(a).

5.3. Throughput of File Writes

We measured the throughput of writing data to a file existing only in the lower layer using the fio benchmark [7].

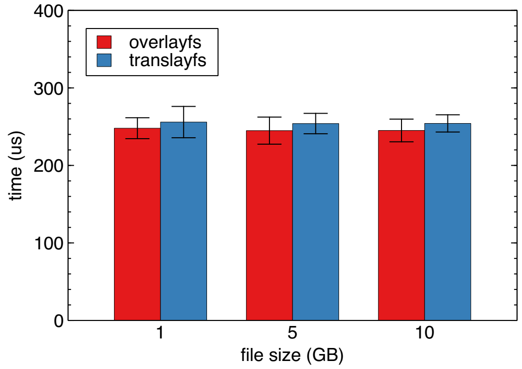


Figure 12: The latency of a write to a file only in the upper layer.

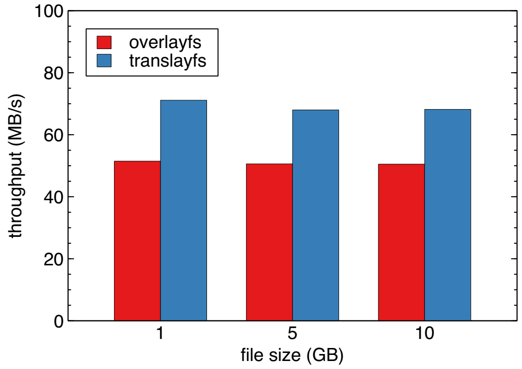


Figure 13: The throughput of sequential file writes.

First, we wrote one-byte data to each file block and skipped the remaining 4095 bytes. We sequentially performed this access until the last block. Fig. 13 shows the throughput of the sequential stride writes. TranslayFS always outperformed OverlayFS and the performance was improved by 34–38%. One of the reasons is that OverlayFS needed a long time for the first write. Another reason is that prefetching data from the file in the lower layer could hide the overhead of TranslayFS.

Next, we randomly wrote data to each file block. The size of data written at a time was 1, 41, 410, or 4096 bytes. The total size of written data was set to 1% of the file size. Fig. 14 shows the throughput of the random writes to a 10-GB file. When we wrote one-byte and 41-byte data at a time, the throughput of TranslayFS degraded by 95% and 39%, respectively. This is because file prefetching from the lower layer did not work well in random access and the overhead of TranslayFS was not hidden. However, TranslayFS outperformed OverlayFS when we wrote more than 78-byte data at a time. The throughput of OverlayFS slightly increased, while that of TranslayFS dramatically increased. When we wrote 410-byte and 4-KB data at a time, TranslayFS was 4.5 and 12 times higher in throughput, respectively. This is because TranslayFS copied a smaller

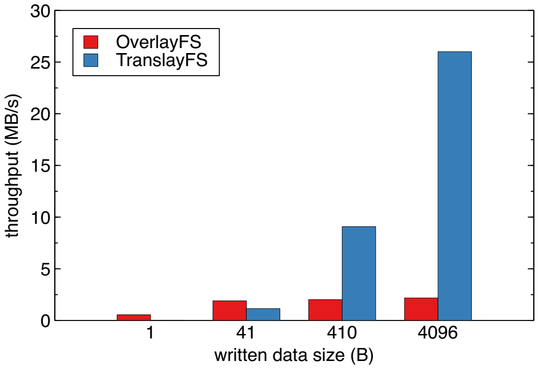
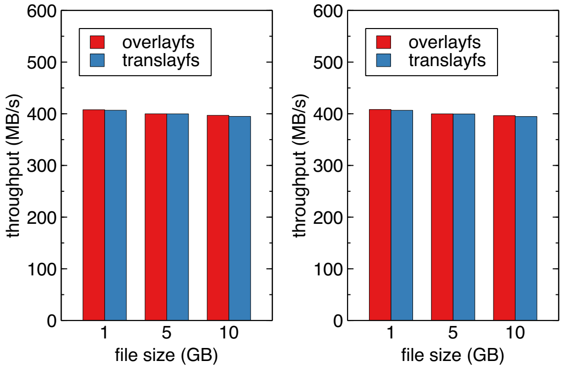


Figure 14: The throughput of random file writes.



(a) Lower layer

(b) Upper layer

Figure 15: The throughput of reads from a file in either the lower or upper layer.

amount of data, especially no data for 4-KB writes, from the lower layer per write operation.

5.4. Throughput of File Reads

We measured the throughput of reading data from a file existing in either the lower or upper layer. We sequentially read 4-KB data from each file block. Fig. 15 shows the throughput of the sequential reads. TranslayFS suffered from performance degradation by up to 0.5%, regardless of which layer a file existed in. This overhead is due to the redirection of read operations to the lower or upper filesystem in TranslayFS.

Next, we measured the throughput of reading data from a file existing in both layers. To create such a file, we first wrote only one-byte data to a file existing only in the lower layer. By this write, OverlayFS created a normal file that had real data in all the blocks in the upper layer. TranslayFS created a sparse file with only 4-KB real data and many holes. Fig. 16 shows the throughput of the sequential reads of 4-KB data to this file. The performance degradation in TranslayFS was 0.4–3.8%. This was larger than when

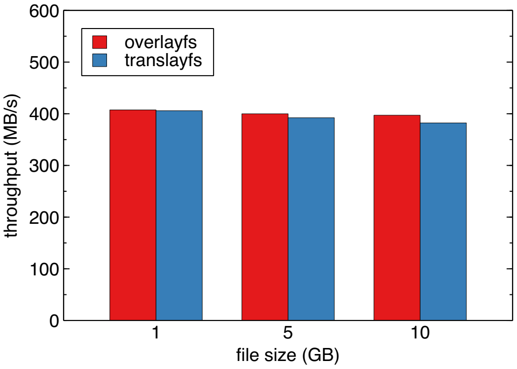


Figure 16: The throughput of reads from a file in both layers.

the entity of a file existed in either layer. This is because TranslayFS needed to first check a hole and then redirect a read operation to the lower filesystem for most of the requests.

6. Related Work

OverlayFS is one of the implementations of the union filesystem. The other implementations are the Translucent filesystem [8] in SunOS, the union mount filesystem [9] in 4.4 BSD-Lite, UnionFS [10], AUFS [3], and so on. The union filesystem performs a per-file copy-up operation, which copies the entire file to the upper layer, when a file in the lower layer is modified. OverlayFS performs this copy at the time of file open in a writable mode, while AUFS does that at the time of the first write to a file. This behavior of AUFS is similar to TranslayFS in that a new file is created in the upper layer by the first write. AUFS is also supported in Docker, but it is not often used recently because it is not merged to the Linux mainline kernel.

Mizusawa et al. revealed that the synchronization of the file cache in the copy-up operation is the root cause of performance degradation in OverlayFS [11]. To increase the I/O performance of a container, they propose a method of decreasing the frequency of this synchronization, e.g., once per 50 copy-up operations. However, this method would not be acceptable as a mechanism of the operating system in terms of fairness. Randomly selected files become reliable by the cache synchronization, while the others are kept unreliable. In addition, the cache synchronization still takes a long time for some of the files. Therefore, this method cannot prevent the DoS attack with copy-on-write completely. TranslayFS can decrease the amount of the file cache to be synchronized by reducing the size of a file created in the upper layer.

In Linux 5.10, the volatile mount option is added to OverlayFS. This option completely disables the cache synchronization on the copy-up operation. It can largely improve the performance of OverlayFS when a container modifies a file whose real entity exists in the lower layer. However, the file copied to the upper layer can be lost on a system crash because the file cache is not written back to a disk. In addition, when the cache synchronization is performed later, it could take a long time when a large file has to be written back.

There are several copy-on-write filesystems. ZFS is a filesystem that was developed for Solaris by Sun Microsystems and is currently developed in the ZFS on Linux project [4]. It provides high reliability and many features. It first creates a storage pool from physical disks and then creates a filesystem from the storage pool. Using ZFS, Docker creates a read-only snapshot from the image layer in the filesystem and then creates a writable clone from the snapshot. When a container writes data to the clone, ZFS performs copy-on-write by the 128-KB block, which is much larger than the 4-KB block used in TranslayFS. Since ZFS is licensed under CDDL, which is not compatible with GPL, it is officially supported only in Ubuntu.

Btrfs [5] is a reliable filesystem developed by Oracle, Red hat, and so on. It is based on ZFS and provides mechanisms such as copy-on-write and snapshots. Like ZFS, Btrfs first creates a storage pool and then creates a subvolume, which is part of the filesystem, from the storage pool. Using Btrfs, Docker creates a writable snapshot from the image layer in the subvolume. When a container writes data to the snapshot, Btrfs performs copy-on-write by the 16-KB block. However, Btrfs is still not stable and is not supported in RHEL 8 and later.

Devicemapper [6] is a storage driver developed with the device mapper mechanism in Linux by Red Hat. Unlike OverlayFS, it works at the block level, not at the filesystem level. It first creates a thin pool that consists of a data device and a metadata device and then creates a base device from the thin pool by thin provisioning. Using devicemapper, Docker creates a writable snapshot from the image layer in the base device. When a container writes data to the snapshot, devicemapper performs copy-on-write by the 64-KB block. Since devicemapper is not a filesystem, it is relatively difficult for users to manage container images.

Qcow2 is a file format used for the disk images of virtual machines (VMs) in QEMU [12]. When a VM writes data to a read-only base image, QEMU stores the data in a writable qcow2 file. Like TranslayFS, it incrementally allocates physical blocks in a virtual disk only for written data and saves disk space. However, the qemu-nbd tool is required to mount a disk image because qcow2 is not a file format natively supported by the operating system.

Data deduplication is a technique for eliminating duplicate copies and saving disk space. File-level data deduplication [13] was proposed earlier, but chunk-level data deduplication [14] is usually used due to better performance. The inline method deduplicates data on disk write, while the post-processing method deduplicates data later. ZFS, Btrfs, and APFS in macOS support this function to prevent real data from being duplicated in a disk on a file copy. If OverlayFS uses these filesystems as the upper and lower layers, it could reduce the overhead of copying a file in the lower layer to the upper layer. TranslayFS can be considered OverlayFS with chunk-level data deduplication.

7. Conclusion

This paper proposed a new filesystem called TranslayFS to prevent the DoS attack with copy-on-write. TranslayFS holds only a modified part of a file in the upper layer without copying the entire file from the lower layer. Upon a file read, it reads the modified part from the upper layer, while it reads the unmodified part from the lower layer. This mechanism can prevent a container from being suspended for a long time by a small write to a large file. Also, it can prevent the upper layer from being out of space by the files copied from the lower layer. We have implemented TranslayFS in the Linux kernel and conducted several experiments. As a result, TranslayFS could complete the first one-byte write only in 339 μs although OverlayFS took 51 seconds for a 10-GB file. From this result, it was shown that TranslayFS could prevent the DoS attack.

One of our future work is to merge TranslayFS and OverlayFS. TranslayFS can dramatically reduce the overhead at the first write to a file, but it imposes a slight overhead to successive file operations. We can eliminate this overhead by copying all the file blocks at some point to the upper layer and using OverlayFS after that.

Acknowlegment

This work was partially supported by JST, CREST Grant Number JPMJCR21M4, Japan.

References

[1] D. Merkel, “Docker: Lightweight Linux Containers for Consistent Development and Deployment,” *Linux J.*, vol. 2014, no. 239, 2014.

[2] M. Szeredi, “Overlay Filesystem,” https://www.kernel.org/doc/Documentation/filesystems/overlayfs.txt.

[3] J. R. Okajima, “Aufs5 – Advanced Multi Layered Unification Filesystem Version 5.x,” https://aufs.sourceforge.net/.

[4] Lawrence Livermore National Laboratory, “ZFS on Linux,” https://zfsonlinux.org/.

[5] O. Rodeh, J. Bacik, and C. Mason, “BTRFS: The Linux B-Tree Filesystem,” *ACM Trans. Storage*, vol. 9, no. 3, pp. 1–32, 2013.

[6] Red Hat, Inc., “dm-thin: Thin Provisioning,” https://www.kernel.org/doc/Documentation/device-mapper/thin-provisioning.txt.

[7] J. Axboe, “fio: Flexible I/O Tester,” https://github.com/axboe/fio.

[8] D. Hendricks, “A Filesystem for Software Development,” in *Proc. USENIX Summer 1990 Conf.*, 1990, pp. 333–340.

[9] J. Pendry and M. K. McKusick, “Union Mounts in 4.4BSD-Lite,” in *Proc. USENIX 1995 Technical Conf.*, 1995, pp. 25–33.

[10] C. P. Wright, J. Dave, P. Gupta, H. Krishnan, D. P. Quigley, E. Zadok, and M. N. Zubair, “Versatility and Unix Semantics in Namespace Unification,” *ACM Trans. Storage*, vol. 2, no. 1, pp. 1–32, 2006.

[11] N. Mizusawa, J. Kon, Y. Seki, J. Tao, and S. Yamaguchi, “Performance Improvement of File Operations on OverlayFS for Containers,” in *Proc. IEEE Int. Conf. Smart Computing*, 2018, pp. 297–302.

[12] F. Bellard, “QEMU,” https://www.qemu.org/.

[13] W. J. Bolosky, S. Corbin, D. Goebel, and J. R. Douceur, “Single Instance Storage in Windows 2000,” in *Proc. USENIX Windows System Symp.*, 2000, pp. 13–24.

[14] C. Policroniades and I. Pratt, “Alternatives for Detecting Redundancy in Storage Systems Data,” in *Proc. USENIX Annual Technical Conf.*, 2004, pp. 73–86.