A Lightweight and Fine-grained File System Sandboxing Framework

Ashish Bijlani
Georgia Institute of Technology
Atlanta, GA
ashish.bijlani@gatech.edu

Umakishore Ramachandran
Georgia Institute of Technology
Atlanta, GA
rama@cc.gatech.edu

ABSTRACT

File system sandboxing is a useful technique for protecting sensitive data from untrusted binaries. However, existing approaches do not allow fine-grained control over policy enforcement, require superuser privileges, or incur high performance overhead. This paper proposes SandFS, a lightweight and fine-grained file system sandboxing framework for unprivileged users and applications. We have designed SandFS as a stackable in-kernel file system that can be safely be extended at runtime from the user-space to enforce custom security policies in the kernel and offer near-native performance. With SandFS, multiple sandboxing layers could be stacked on top of each other, with each higher layer further enforcing its own policies to provide a restricted view of the lower. Our evaluation of SandFS with real-world workload shows that it imposes less than 10% performance overhead.

CCS CONCEPTS

• **Security and privacy → File system security**;

KEYWORDS

Security, Sandboxing, File System

ACM Reference Format:
Ashish Bijlani and Umakishore Ramachandran. 2018. A Lightweight and Fine-grained File System Sandboxing Framework. In *APSys '18: 9th Asia-Pacific Workshop on Systems (APSys '18), August 27–28, 2018, Jeju Island, Republic of Korea.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3265723.3265734

APSys '18, August 27–28, 2018, Jeju Island, Republic of Korea

ACM ISBN 978-1-4503-6006-7/18/08…$15.00
https://doi.org/10.1145/3265723.3265734

1 INTRODUCTION

File system sandboxing is a useful technique for enforcing security policies following the principal of least privilege, and restricting access to sensitive data from untrusted program execution. Discretionary Access Control (DAC) policies offered by mainstream operating systems allow users to specify permissions on files and directories, which are enforced by the kernel to allow or deny access based on the identifier of the requesting user. While DAC policies can protect the confidentiality and integrity of the data belonging to one user from other potentially malicious users on the system, they are inadequate to provide a sandboxed environment against untrusted binaries. An executable launched inherits all the permissions of the user, and, therefore, can freely access user's private data.

Applications often rely on third-party binaries to extend their functionality. For instance, web browsers allow users to install third-party extensions [4]. Mobile applications monetize by including third-party advertisement libraries [2, 7]. However, DAC policies fail to protect the private data files from buggy or malicious third-party binaries that share the execution environment with the host application (i.e., are executed under the same user identifier).

Alternative mechanisms, such as Mandatory Access Control (MAC) policies and Capabilities are often employed to achieve a fine-grained sandboxing of the host file system against untrusted binaries. For example, the Linux Security Module (LSM) framework provides hooks to perform rule-based MAC checks by invoking registered callback handlers from security kernel modules, such as SELinux during file system operations. However, they require superuser privileges, and, thus are only meant for system administrators. Additionally, leveraging such mechanisms to sandbox binaries require careful and detailed analysis of their execution and corresponding tuning of rules so as to not unnecessarily restrict functionality.

Consequently, System Call Interposition (SCI) is a commonly used technique for sandboxing untrusted applications by monitoring the system calls and mediating access to kernel resources [10]. However, SCI suffers from a number of limitations. First, SCI-based tools mostly employ `ptrace`

```
$ sandfs -s sandfs.o -d /home/user /bin/bash
```

Figure 1: Example usage of SandFS to sandbox user's home directory. `sandfs.o` bytecode file represent runtime kernel extensions to enforce custom security checks.

framework to intercept system calls in the user-space and inspect user-supplied arguments at runtime [10, 11, 13, 15, 18]. However, `ptrace` imposes a high runtime performance overhead, rendering such tools impractical. Second, SCI-based sandboxing techniques are vulnerable to TOCTTOU attacks [8] due to non-atomic security policy enforcement, even with no possibility of path manipulation in user supplied arguments [1]. Finally, not all file system requests are performed using system calls. For example, `mmap` I/O does not result in system calls. Hence, SCI-based sandboxing cannot provide fine-grained control over enforcing access policies.

Our Solution. In this paper, we present SandFS, a light-weight file system sandboxing framework for UNIX-like commodity operating systems to overcome the limitations of existing SCI-based sandboxing techniques. It is designed for unprivileged users and applications to limit the attack surface resulting from security bugs or malicious untrusted binaries. SandFS framework offers the following advantages:

- Fine-grained access control over kernel objects,
- Low performance overhead,
- High-level language support for defining checks,
- Stackable (layered) protection, and
- Runtime dynamic (programmatic) enforcement.

Applications of SandFS. Users can mount SandFS on the host file system to restrict access to sensitive data (e.g., private keys) in their workspace when executing untrusted applications. For example, machine learning models can be evaluated without inspecting the code for malicious file system behavior. Figure 1 shows its usage. Applications can further stack SandFS to protect against untrusted third-party helper binaries (e.g., browser extensions).

Overview. We have designed SandFS as an extensible in-kernel file system framework. That is, the user-space can safely extend its functionality at runtime by inserting custom security checks (called *extensions*) in the kernel that are performed on the requests issued by the upper file system (e.g., VFS). Being a file system, SandFS complements existing MAC-based sandboxing mechanisms (e.g., SELinux).

SandFS consists of three core components: 1) a stackable file system kernel driver that exports low-level file system APIs to applications, 2) a user-space helper library that allows applications to define custom security checks using a subset of C-language, and 3) an in-kernel virtual machine runtime that safely executes the checks in the kernel at runtime to offer access protection at near-native performance. Security functionality is extensible through familiar file system interfaces, which export a set of abstractions (e.g., `dentry`) and APIs (e.g., `lookup`, `open`, etc.) to applications. Applications can register a callback for some or all of the file system APIs as necessary to achieve the desired security functionality.

Contributions. In this paper, we:

- propose a light-weight and fine-grained file system sandboxing framework for unprivileged users,
- present the design and architecture of SandFS and its usage model, and
- evaluate the performance of SandFS with real-world workload to demonstrate its practicality.

2 RELATED WORK

In this section, we summarize approaches adopted by past works that focused on creating a secure file system environment for untrusted binaries and compare them with ours.

Isolation. A number of tools and techniques have been proposed [3, 11, 17] that leverage mechanisms, such as `chroot`, namespaces, and virtualization to create an isolated file system environment for untrusted applications. In contrast, SandFS does not create an isolated file system, but enforces checks that restrict access from untrusted executables to shared files within the same file system environment. As such, SandFS can complement existing isolation techniques to further add a layer of protection.

Sandboxing. A few tools have explored System Call Interposition technique to build a file system sandboxing layer [9–11, 13, 15, 18]. These tools rely on `ptrace` to intercept system calls in the user-space and inspect their arguments for runtime access policy enforcement. However, not only `ptrace` incurs high performance overhead [15], but interposing at runtime in user-space is also vulnerable to TOCTTOU attacks [8]. Furthermore, not all file system requests are performed using system calls. For instance, `mmap` I/O does not result in system calls. Therefore, SCI-based sandbox fails to provide support for fine-grained access policy enforcement.

SCI-based tools also require careful replication of kernel file system state in user-space [15]. For example, by interposing on system calls, one can only filter `read()/write()` requests based on file descriptors. Therefore, file descriptor to path mappings have to be maintained in the user-space as well, which may lead to further synchronization issues and race conditions. SandFS, on the other hand, does not inspect user-supplied arguments to system calls; instead, it is designed as a stackable kernel file system layer in to apply security checks directly on low-level kernel abstractions (e.g., dentry) atomically, thereby avoiding TOCTTOU risks or replicating file system state.

`Seccomp-bpf` framework lets applications define and install their system call filtering (allow or deny) rules at runtime, but requires hand-coded BPF filter assembly programs. However, the filters cannot be removed once installed. SandFS provides a user-space helper library that exports a familiar

Technology	Dynamic Policies	Unprivileged Users	Fine-grained Control	Performance Overhead
chroot	×	×	×	-
LD_PRELOAD	✓	✓	×	Low
Ptrace	✓	✓	×	High
SELinux	×	×	✓	Low
Seccomp	✓	✓	×	-
Namespaces	×	✓	×	-
FUSE [19]	✓	✓	✓	High
SandFS	✓	✓	✓	Low

Table 1: Existing file system sandboxing approaches.

file system interface and allows custom security checks to be written in a subset of C language constructs, thereby making it easy for developers to write portable checks, compared to system calls that are specific to a particular architecture.

An important point to note here is that `seccomp-bpf` can only filter system calls based on their type and raw arguments; it cannot inspect user-supplied arguments (e.g., memory contents of `path` argument in `open()` system call). Therefore, applications cannot rely on `seccomp-bpf` alone to enforce custom access policies. MBox [15] utilizes `seccomp-bpf` framework only as an auxiliary tool to reduce `ptrace` overhead by filtering out unnecessary system calls and interposing only on the few relevant ones (e.g., `open()`, `read()`, etc.). `pledge` [6] on OpenBSD, similar to `seccomp-bpf`, allows applications to also filter out multiple predefined set of system calls. Working at the system call level, both `pledge` and `seccomp-bpf` suffer from TOCTTOU races.

Capsicum [21] addresses the limitations of `seccomp-bpf` and `pledge` frameworks by allowing applications to enforce access policies on individual system resources (e.g., file descriptors) as opposed to system calls. However, it requires modifying applications. In contrast, being a file system, SandFS is transparent to applications. As with Capsicum, developers can further compartmentalize their applications to enable fine-grained sandboxing with SandFS where each compartment acts as an independent sandbox §5.2.

Capabilities [5] and Mandatory Access Control (MAC) policies, such as SELinux and AppArmor are built into commodity operating systems and are often employed to achieve a fine-grained sandboxing of the host file system against untrusted binaries. However, they require admin privileges. As SandFS has been designed as a lightweight file system sandboxing framework for unprivileged users and applications, it complements existing MAC-based sandboxing mechanisms.

`LD_PRELOAD` facility could be used by applications as a lightweight technique to intercept file system requests in user-space by overloading C library function calls (e.g., `read`, `write`) and enforce custom checks [14]. However, malicious executables could bypass C library functions and attempt to make direct raw system calls to the kernel.

User-space File Systems. FUSE [19]-based security user file systems also export a low-level file system interface for fine-grained control over resources and allow applications to define their custom policies. However, FUSE framework incurs a high performance overhead [20].

3 DESIGN AND ARCHITECTURE

3.1 Threat Model

Under the UNIX DAC policies, applications launched by a user inherit their access permissions, and therefore, are granted full access to all the files owned by the user (ambient authority). As such, the DAC model fails to protect the private data of the user against buggy and malicious applications.

Modern applications, such as web browsers that large and complex typically rely on third-party helper binaries (e.g., extensions) to extend their functionality. However, malicious third-party binaries can exploit ambient authority of the host application to steal sensitive data.

SandFS is a file system sandboxing framework for unprivileged users and applications on UNIX-like systems to limit the attack surface resulting from security bugs or potentially malicious untrusted binaries.

3.2 Goals

We have built SandFS to overcome the limitations of existing SCI-based sandboxing tools. Specifically, we identify the following operative goals.

Fine-grained access policy enforcement. SandFS must allow users to enforce custom security access policies in a fine-grained manner by mediating all file accesses.

Stackable sandboxing layers. SandFS framework must allow multiple sandboxing layers to be stacked on top of each other, with each layer enforcing its own security policies.

Avoid TOCTTOU risks and OS state replication. Unlike existing SCI-based sandboxing tools, SandFS must avoid common pitfalls that lead to TOCTTOU races or replication of file system state in the user-space.

Low Performance Overhead. Finally, SandFS must incur low performance overhead for the developers to adopt it.

3.3 Challenges and Mechanisms

To achieve the aforementioned goals, SandFS has been designed as an extensible kernel file system framework. It presents itself as a file system to the user-space, which when mounted on top of the host file system can transparently intercept and mediate all low-level file system requests from applications to the host file system. Its functionality can be safely extended at runtime from the user-space by inserting security checks (extensions) in the kernel to enforce custom access policies on the intercepted file system requests. While this design allows the user-space to perform fine-grained

checks directly on low-level file system abstractions (e.g., dentry), it raises a number of challenges.

Safety. SandFS functionality of the kernel at runtime to be able to perform custom security checks. extensions must not be able to access arbitrary memory addresses or leak pointer values to the user-space. SandFS relies on eBPF extension framework to offer the necessary safety guarantees by restricting extensions to only a few well-defined helper functions in the kernel §3.5.

Isolation. Since malicious third-party binaries in applications may exploit their ambient authority to corrupt or manipulate the sandboxing data structure, SandFS only allows the owner (main) thread to create/modify the data structures. To further reduce the attack surface, SandFS leverages `namespaces` to limit the sandboxed view of the file system to only the owner process and its children.

3.4 Architecture

Figure 2 depicts the architecture of SandFS. It consists of three core components: kernel driver, in-kernel Virtual Machine (VM) runtime, and a helper library. The driver registers itself as a file system to interface with the VFS operations and acts as thin stackable interposition layer. That is, when mounted on a host (lower) file system, it neither performs any I/O nor does it implement any new functionality of its own; it directly forwards all requests issued by the upper file system (e.g., VFS) to the lower file system.

The VM allows unprivileged users and applications to safely extend the functionality of the driver at runtime by integrating their custom security checks (extensions) directly into the file system stack in the kernel virtual address space. As a result, unlike existing SCI-based approaches that inspect user-supplied input arguments, SandFS works with actual low-level kernel objects (e.g., dentry) and is not vulnerable to TOCTTOU risks [8].

The library provides a familiar set of file system APIs and abstractions on UNIX-based systems. The user-space can register callbacks for some or all of the APIs to implement security checks in a subset of C language. The checks are complied into VM bytecode, loaded into the kernel as extensions, and safely executed by the driver closer to the lower file system using the VM runtime.

3.5 eBPF

We use eBPF [12] framework as the in-kernel VM runtime environment to achieve the safety guarantees. The reasons are many-fold. Extended Berkeley Packet Filters (eBPF) framework is an extension of classic BPF [16], a pseudo machine architecture for packet filtering. eBPF enhances classic BPF by including 64-bit support and richer programming constructs such as call, load, store, and conditional jumps.

High-level language support. eBPF enables code to be written in a subset of C language for much larger expressiveness while still being safe and small. We tap into this feature of eBPF to export a familiar file system interface to the user-space for implementation of security checks in a high-level language. The C code is then compiled into BPF bytecode using `clang` compiler and loaded into the kernel.

Runtime Safety. eBPF provides safe bytecode execution environment. The inserted code is statically verified by checking for infinite loops and illegal memory references. Although BPF bytecode is loaded into the kernel, it is prohibited from accessing arbitrary kernel memory regions; instead, the framework expects a whitelist of kernel helper functions. For instance, standard helper function `bpf_get_current_uid_gid` allows the bytecode to retrieve the user and group identifiers of the current process.

eBPF is a generic runtime kernel extension framework. Nevertheless, it is currently only used by the networking and system profiling subsystems. To adopt it for SandFS, we added a set of kernel helper functions to assist in stacking security functionality. Specifically, we added support for `sandfs_bpf_read` and `sandfs_bpf_write` that allows SandFS extensions to read and modify parameters passed by the driver, respectively (e.g., `path` and `mode` parameters in `sandfs_open` API). The safety guarantees provided by the eBPF framework enables SandFS to be used by unprivileged users and untrusted applications without exposing a large attack surface. The verified code can further provide native performance through JIT compilation.

Maps. eBPF framework allows user-space to create opaque key-value data structures called *maps* for bookkeeping, as needed. Maps are created using `bpf_map_create` system call. Once created, user-space can operate on them using file descriptors. The framework provides system call APIs to lookup, update, and delete map entries. Maps are also accessible to the eBPF bytecode in the kernel, and thus provide a channel between the user-space and bytecode to share execution state or data. Since map entries can be arbitrary data blobs, it is up to the user-space and the eBPF bytecode to define their data types.

3.6 SandFS APIs and abstractions

SandFS library provides a set of APIs and abstractions to the developers for easy implementation of their security extensions, hiding the complex details. Table 2 lists the APIs in detail. `sandfs_ops` interface exports all Virtual File System (VFS) APIs, (e.g., `lookup` for path to inode mapping) to offer fine-grained access control over kernel abstractions (e.g., dentry). Furthermore, many file system operations in APIs are optional. For example, callbacks for data I/O APIs (e.g.,

Interface	API(s)	Description
VFS	`sandfs_ops`	FS Ops (e.g., `lookup`, `open`, `read`, etc.)
eBPF Map	`CRUD`	Hosts arbitrary data blobs.

Table 2: APIs and abstractions provided by SandFS. It provides a high-level file system interface for easy adoption. CRUD (create, read, update, and delete) APIs are offered for map data structures to operate on Key/Value pairs.

`read/write`) can be empty if checks have been performed in meta-data APIs (e.g., `open`).

To be able to configure kernel extensions and deploy new or modify existing access policies at runtime from the user-space, SandFS library provides a eBPF `HashMap` data structure that is capable of hosting arbitrary key/value blobs. However, since SandFS targets unprivileged users and applications, the hashmap is only accessible to the thread that created it, typically the thread that mounts SandFS. This design disables arbitrary accesses to the map and protects it from malicious binaries that may attempt to disable checks or corrupt its state. SandFS library further abstracts low-level details of `HashMap` and provides high-level `CRUD` APIs to create, read, update, and delete entries.

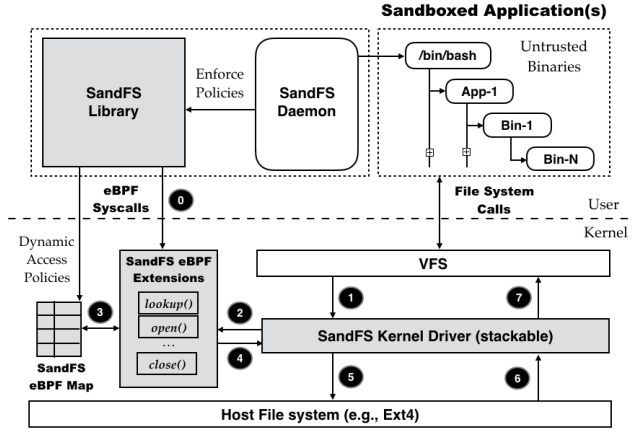

Figure 2: Overview of SandFS framework. The components introduced have been highlighted in grey.

3.7 Workflow

To understand how SandFS provides a sandboxing file system layer, we describe its workflow in detail below.

Before mounting SandFS, the user-space must load the eBPF program containing the security extensions into the kernel and register them with the SandFS driver ❶. This is achieved by performing `bpf_load_prog` system call, which invokes eBPF verifier in the kernel to check the integrity of the extensions. If the verification fails, the extensions are discarded and the user-space is notified of the errors. If the verification step succeeds and the JIT engine is enabled, the handlers are processed by the JIT compiler to generate machine assembly code ready for execution.

As requests are issued to the driver from the upper file system (e.g., VFS) ❶, each request is first delivered to the corresponding extension handler for enforcing security policies ❷. The handler may refer to the shared data maintained the in eBPF map ❸ to allow or deny the request and may even manipulate the arguments as necessary (e.g., change the value of `mode` argument in `open()`). Security check result from the handler is propagated back to the driver ❹. Denied requests are delivered immediately to the user-space with the corresponding error code (`errno`) ❼, whereas permitted requests are forwarded to the lower file system ❺. It is important to note that the SandFS driver only acts as a thin interposition layer between the lower file system and registered security kernel extensions from user-space. As such, it does not perform any I/O operation or attempts to serve requests on its own. Results from the lower file system ❻ are simply delivered to the user-space ❼.

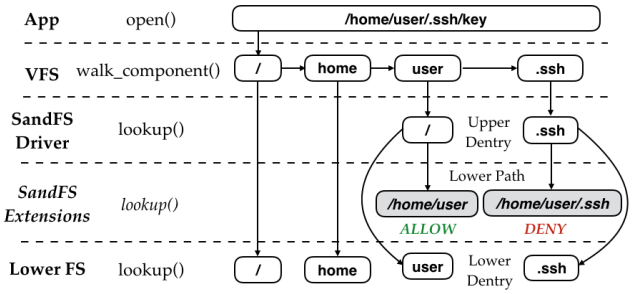

Figure 3: This figure shows how SandFS extensions enforce access policies during VFS `lookup` operation.

To understand how SandFS enforces user-defined security policies in the kernel and avoids TOCTTOU race conditions, we illustrate its working by taking `lookup` operation as an example in Figure 3. It is the most common file system operation issued by the VFS internally to perform `path` to `inode` mapping, once for each `path` component under a RCU lock when serving system calls, such as `open()`, and `mkdir()`.

When SandFS is mounted on `/home/user` to sandbox the working space of the user, each VFS `lookup` request is first delivered to `lookup` implementation in SandFS driver. The driver `lookup` function first obtains the lower dentry and its (lower) inode of the requested component (e.g., `.ssh`) by calling into the lower file system, and subsequently invokes `dentry_path()` to build its full path (e.g., `/home/user/.ssh`). During this process any symbolic links or special names (e.g., `.` and `..`) are resolved. The computed full path is then passed as an argument to `sandfs_lookup` extension to perform registered checks. Upon successful return value (allow) from the extension, SandFS driver creates an (upper) inode for the path component and returns to the VFS.

Unlike existing SCI-based file system sandboxing techniques that attempt to enforce security policies at system call boundary where user-supplied arguments (e.g., `path`)

Component	Loc New
SandFS kernel driver	3890
SandFS user-space library	1598

Table 3: Engineering effort required to support SandFS functionality.

have not been processed to get the corresponding kernel objects (e.g., `dentry`), SandFS extensions are invoked from deep within the kernel (below VFS) and work directly with low-level `path` components and kernel `dentry` objects, thereby avoiding any TOCTTOU races.

4 IMPLEMENTATION

We have implemented SandFS in Linux version 4.10. SandFS driver is based on wrapfs [22], which is a stackable no-ops file system. That is, its file system handlers do not perform any I/O operations; they simply call into the handlers of the underlying file system. Being a stackable file system, a malicious app could stack a number of SandFS file system layers on top of each other and cause kernel stack overflow. To guard against such an attack, we limit the number of SandFS layers that could be stacked on a mount point. Specifically, we rely on `s_stack_depth` field in the superblock to track number of stacked layers and check it against `FILESYSTEM_MAX_STACK_DEPTH`, which is limited to two. Table 3 reports the number of lines of code for SandFS.

5 USE CASES

Below we describe three important real-world usecases of SandFS that make it desirable over existing approaches.

5.1 Hiding private user data

Many applications require access to user's home directory to work properly. However, at the same time malicious applications can steal private user data, such as `$HOME/.ssh` keys. With SandFS framework, unprivileged users can construct a sandbox to protect private files while executing untrusted applications. For example, by mounting SandFS on `$HOME` directory, users can enforce custom security checks, thereby hiding private directories and files in their workspace and only allowing access to a few absolutely necessary files when executing untrusted applications (see §3.7).

5.2 Designing secure applications

Applications that rely on untrusted third-party binaries to extend their functionality can use SandFS framework to protect their private data. For instance, web browsers that execute third-party extensions as separate processes to achieve memory isolation can define and enforce fine-grained custom security access checks using SandFS to protect access to sensitive data from malicious third-party extensions. However, since these extensions may need persistent storage, browsers can restrict access to their private files using eBPF maps, while providing full access to the files that are created at runtime by these binaries. Figure 5 shows an example.

```
int sandfs_open(void *args) {

    /* get mode */
    u32 mode;
    ret = sandfs_bpf_read(args, PARAM1, &mode, sizeof(u32));
    if (ret) return ret;

    /* example check: file creation not supported */
    if (mode & O_CREAT) return -EPERM;

    /* example enforcement: rewrite arg to force RDONLY mode */
    mode = O_RDONLY;
    ret = sandfs_bpf_write(args, PARAM1, &mode, sizeof(u32));
    if (ret) return ret;

    return 0; /* allow access */
}
```

Figure 4: Example SandFS checks to deny creation of new files and rewriting of arguments at runtime to create a sandboxing file system layer.

```
int sandfs_lookup(void *args) {

    /* get path */
    char path[PATH_MAX];
    ret = sandfs_bpf_read(args, PARAM0, path, PATH_MAX);
    if (ret) return ret;

    /* lookup in map if the path is marked as private */
    u32 *val = bpf_map_lookup(&access_map, path);

    /* example check: prohibit access to private files */
    if (val) return -EACCES;

    return 0; /* allow operation */
}
```

Figure 5: Example SandFS checks that can be enforced by Android apps to limit access by untrusted advertisement libraries to private files.

5.3 Hardening Containers

Containers package a number of interdependent applications and their libraries to achieve a desired functionality. Each application poses its own set of security requirements. SandFS can be used to harden containers, by stacking a separate layer of SandFS for each application. Figure 4 shows an example of how requests to new file creation could be denied.

6 EVALUATION

To measure the performance of SandFS, we run the same real-world workloads as used in Mbox [15] and Apiary [17] under SandFS sandbox and compare it with native mode. Our testbed included Ubuntu 16.04.3 environment with Intel Quad-Core i5-3550 3.3GHz, 16GB RAM, and EXT4 formatted Samsung 850 EVO 250GB SSD. Results presented in Table 4 indicate that SandFS incurs less than 10% overhead.

Benchmark	Native	SandFS	Overhead	Description
Tar	61.05s	63.84s	4.57%	Compress (tar.gz) linux-4.17 source files.
Untar	5.13s	5.63s	9.75%	Decompress linux-4.17 gzipped (tar.gz) tarball.
Build (-j4)	27.15s	29.67s	9.28%	Compiling linux-4.17 (tinyconfig) kernel with 4 parallel jobs.

Table 4: Results from benchmarking SandFS for performance. We measure and report the total execution time of each benchmark with and without SandFS (native). Machine used for benchmarking contains four cores.

7 CONCLUSION

In this work we presented a lightweight file system sandboxing framework called SandFS. It has been designed as a stackable file system for unprivileged users and applications to enforce custom security checks to protect private data from untrusted and malicious binaries. SandFS incurs less than 10% overhead. It's source code is available on GitHub.

8 ACKNOWLEDGMENT

We would like to thank our shepherd, Dr. Adam Morrison, and all anonymous reviewers for their insightful feedback and suggestions, which substantially improved the content and presentation of this paper. We also would like to acknowledge the editorial efforts of Dr. Gargi Sawhney. This work was funded in part by NSF CPS program Award #1446801, and a gift from Microsoft Corp.

REFERENCES

[1] Jonathan Anderson. 2017. A Comparison of Unix Sandboxing Techniques. *FreeBSD Journal* (2017).

[2] Michael Backes, Sven Bugiel, and Erik Derr. 2016. Reliable Third-Party Library Detection in Android and its Security Applications. In *Proceedings of the 23rd ACM Conference on Computer and Communications Security (CCS)*. ACM, Vienna, Austria, 356–367.

[3] Michael Backes, Sven Bugiel, Christian Hammer, Oliver Schranz, and Philipp von Styp-Rekowsky. 2015. Boxify: Full-fledged app sandboxing for stock android. (Aug. 2015), 27–38.

[4] Adam Barth, Adrienne Porter Felt, Prateek Saxena, and Aaron Boodman. 2010. Protecting Browsers from Extension Vulnerabilities. In *Proceedings of the 17th Annual Network and Distributed System Security Symposium (NDSS)*. IEEE, San Diego, CA, 1–17.

[5] Andrew Berman, Virgil Bourassa, and Erik Selberg. 1995. TRON: Process-specific File Protection for the UNIX Operating System. In *Proceedings of the 1995 USENIX Annual Technical Conference (ATC)*. USENIX Association, New Orleans, Louisiana, 14–14.

[6] Theo de Raddt. 2015. pledge() a new mitigation mechanism. (2015). http://openbsd.org/papers/hackfest2015-pledge/mgp00001.html

[7] Ruian Duan, Ashish Bijlani, Meng Xu, Taesoo Kim, and Wenke Lee. 2017. Identifying Open-Source License Violation and 1-day Security Risk at Large Scale. In *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)*. ACM, Dallas, Texas, 2169–2185.

[8] Tal Garfinkel et al. 2003. Traps and Pitfalls: Practical Problems in System Call Interposition Based Security Tools. In *Proceedings of the 10th Annual Network and Distributed System Security Symposium (NDSS)*. IEEE, San Diego, CA, 163–176.

[9] Tal Garfinkel, Ben Pfaff, Mendel Rosenblum, et al. 2004. Ostia: A Delegating Architecture for Secure System Call Interposition. In *Proceedings of the 11th Annual Network and Distributed System Security Symposium (NDSS)*. IEEE, San Diego, CA, 187–201.

[10] Ian Goldberg, David Wagner, Randi Thomas, and Eric A. Brewer. 1996. A Secure Environment for Untrusted Helper Applications Confining the Wily Hacker. In *Proceedings of the 6th USENIX Security Symposium (Security)*. USENIX Association, San Jose, CA, 1–1.

[11] Philip J Guo and Dawson R Engler. 2011. CDE: Using System Call Interposition to Automatically Create Portable Software Packages. In *Proceedings of the 2011 USENIX Annual Technical Conference (ATC)*. USENIX Association, Portland, OR, 21–21.

[12] IOVisor. 2017. eBPF: extended Berkley Packet Filter. (2017). https://www.iovisor.org/technology/ebpf

[13] Kapil Jain and R Sekar. 2000. User-level infrastructure for system call interposition: A platform for intrusion detection and confinement. In *Proceedings of the 7th Annual Network and Distributed System Security Symposium (NDSS)*. IEEE, San Diego, CA, 19–34.

[14] Michael B Jones. 1993. Interposition agents: Transparently interposing user code at the system interface. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles (SOSP)*. ACM, Asheville, NC, 80–93.

[15] Taesoo Kim and Nickolai Zeldovich. 2013. Practical and Effective Sandboxing for Non-root Users. In *Proceedings of the 2013 USENIX Annual Technical Conference (ATC)*. USENIX Association, San Jose, CA, 139–144.

[16] Steven McCanne and Van Jacobson. 1993. The BSD Packet Filter: A New Architecture for User-level Packet Capture. In *Proceedings of the Winter 1993 USENIX Annual Technical Conference (ATC)*. USENIX Association, San Diego, CA.

[17] Shaya Potter and Jason Nieh. 2010. Apiary: Easy-to-use Desktop Application Fault Containment on Commodity Operating Systems. In *Proceedings of the 2010 USENIX Annual Technical Conference (ATC)*. USENIX Association, Boston, MA, 8–8.

[18] Niels Provos. 2003. Improving Host Security with System Call Policies. In *Proceedings of the 12th USENIX Security Symposium (Security)*. USENIX Association, Washington, DC, 18–18.

[19] M. Szeredi. 2005. Filesystem in Userspace. (February 2005). http://fuse.sourceforge.net

[20] Bharath Kumar Reddy Vangoor, Vasily Tarasov, and Erez Zadok. 2017. To FUSE or Not to FUSE: Performance of User-Space File Systems. In *15th USENIX Conference on File and Storage Technologies (FAST) (FAST 17)*. USENIX Association, Santa Clara, CA, 77–90.

[21] Robert NM Watson, Jonathan Anderson, Ben Laurie, and Kris Kennaway. 2010. Capsicum: Practical Capabilities for UNIX. In *Proceedings of the 2010 USENIX Annual Technical Conference (ATC)*. USENIX Association, Boston, MA, 2–2.

[22] E. Zadok, I. Bădulescu, and A. Shender. 1999. Extending File Systems Using Stackable Templates". In *Proceedings of the 1999 USENIX Annual Technical Conference (ATC)*. USENIX Association, Monterey, CA, 57–70.