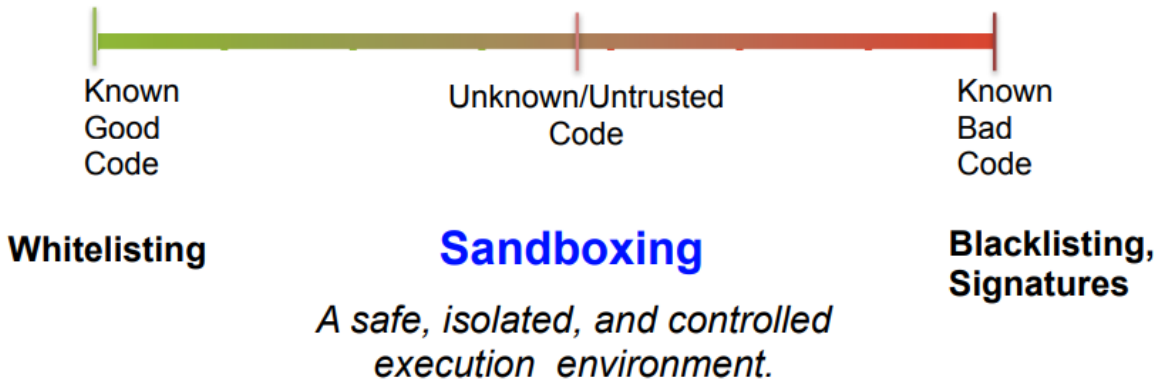
A Lightweight and Fine-grained File System Sandboxing Framework

（一个轻量级和细粒度的文件系统沙箱框架）

主要目标：以安全的方式运行来自互联网的不受信任的第三方代码

不受信任的第三方举例：

- 第三方网页浏览器插件
- 评估机器学习模型等



有很多使用情况需要运行不受信任的第三方代码，而不会冒险损坏系统文件的内容。其中两个例子就是：从可能不可靠的互联网站点获得的 Web 浏览器插件和可能希望评估的机器学习代码。有一个代码运行的范围，从已知良好的代码到已知恶意的代码；在这两者之间是未知的、不受信任的代码。已知良好的代码可以加入到白名单中，已知恶意的代码可以加入到黑名单中，而沙盒技术则用于处理中间的代码。而且沙盒只是一个安全的、被隔离的和可控的执行环境。

Bijlani （本论文的第一作者）关注的是特定类型的沙盒：文件系统沙盒。其思想是在运行这些不受信任的程序时限制对敏感数据的访问。由于这种限制可能需要根据正在运行的程序进行一些更改，因此需要的是动态规则。他举了一些需要进行限制的一些例子，比如限制访问 ~/.ssh/id_rsa* 文件（SSH协议的私钥文件），或者仅允许访问特定类型的文件（例如仅允许 PDF 阅读器访问 *.pdf 文件）。

现有的一些文件系统沙盒技术

1、UGO、ACL、DAC、MAC、RBAC

- UGO是最基本的访问控制策略，它只能控制资源的所有者、组和其他用户的访问权限。ACL、DAC、MAC和RBAC都是UGO的扩展和升级版本。UGO的用户组到其他所用的户之间的范围过于大，难以实现更加灵活的文件数据的访问。
- ACL是Access Control List的缩写，可以更细粒度地控制资源访问权限，例如可以对单个用户或组分别设置不同的权限。但是ACL只能通过用户自主设置其他的用户或用户组对文件的访问权限，不适合大量文件访问权限的设置。
- DAC是Discretionary Access Control的缩写，是一种资源所有者为基础的访问控制策略，资源所有者可以自由决定资源的访问权限。DAC简单易实现，且用户可以控制自己的资源访问，但用户拥有了对资源的完全控制权，容易造成安全风险，而且DAC也难以实现对资源的细粒度访问控制（例如启动的可执行文件会继承用户的所有权限，因此可以自由访问用户的私有数据）。

- MAC（SELinux）是Mandatory Access Control的缩写，是一种系统管理员为基础的访问控制策略，管理员定义了一组安全策略，并强制在系统级别上实施。MAC可以实现高度安全的访问控制，以保护敏感数据，并且容易扩展到大规模系统中。但同时MAC的实现和管理复杂，访问权限设置也不灵活，难以适应变化的环境。

```
//MAC实例：
$ ls -dZ - /etc/    //-Z：在列表中显示SELinux安全上下文
drwxr-xr-x. root root system_u:object_r:etc_t:s0 /etc
//system_u:object_r:etc_t:s0：表示SELinux的安全上下文，system_u表示此目录属于system_u用户，object_r表示此目录的SELinux对象为etc_t类型，s0表示此目录的SELinux安全级别为默认级别。
```

- RBAC是Role-Based Access Control的缩写，是一种基于角色的访问控制策略，通过将访问权限分配给角色而不是用户，管理员可以更好地管理和维护安全策略。RBAC可以实现更细粒度的访问控制，可以基于用户的职责和角色进行授权，可以更好地管理复杂的权限关系。但它需要更复杂的角色和权限管理，且难以适应变化的环境，需要频繁更新角色和权限。

总结：UGO和ACL是Linux系统中比较简单的访问控制策略，而DAC、MAC和RBAC则更为复杂和高级。DAC是较为灵活的，但安全性较低，而MAC和RBAC安全性更高，但也更复杂和难以管理。

2、Chroot、Namespaces

chroot和命名空间（Namespaces）是Linux系统中的两种隔离机制，可以用来限制进程的访问权限，提高系统的安全性。

chroot（Change Root）机制可以将进程的根目录改变为一个新的根目录，使进程在这个新的根目录下运行时只能访问到这个新的根目录和其子目录，而不能访问到原来的根目录和其他目录。这种机制常用于提供一个独立的运行环境，例如用于运行某些服务进程。

命名空间（Namespaces）是Linux中的一个内核特性，用于将一些系统资源隔离开来，例如进程、网络、文件系统、IPC（进程间通信）等。通过将不同的进程放置在不同的命名空间中，就可以避免它们之间的冲突和干扰，提高系统的安全性和可靠性。

这两种机制都是隔离进程的运行环境，提高系统的安全性和稳定性。

```
//Chroot/Namespaces实例：
$ unshare -m /bin/bash
//该命令将创建一个新的 mount namespace，并在新的 mount namespace 中启动一个新的 /bin/bash shell。由于新的 mount namespace 是隔离的，因此该 shell 将只能看到该 mount namespace 中挂载的文件系统，而看不到其他 mount namespace 中的文件系统。-m 选项表示要创建一个新的 mount namespace。
```

3、LD_PRELOAD

LD_PRELOAD 是一个环境变量，它允许用户在运行程序之前指定一个共享库的路径，该共享库将优先于系统默认的库加载到程序中。这意味着用户可以通过 LD_PRELOAD 来劫持一些函数的调用，替换系统默认的函数实现，从而实现一些特殊的目的。

LD_PRELOAD 在一些场景下被广泛使用，例如进行程序调试、记录系统行为、实现钩子函数等等。需要注意的是，LD_PRELOAD 劫持函数调用的行为具有一定的风险，可能会导致程序出现不可预料的问题，因此需要谨慎使用。

```
//LD_PRELOAD实例：
LD_PRELOAD=./wrapper.so /bin/bash
e.g., ssize_t write_wrapper(int fd, …) { return -EACCES; }
//LD_PRELOAD=./wrapper.so：设置环境变量LD_PRELOAD，指定要预加载的共享库文件为当前目录下的wrapper.so文件。
///bin/bash：运行/bin/bash程序，即启动一个新的bash shell。在启动时，因为设置了LD_PRELOAD环境变量，系统会先加载wrapper.so文件中的函数，再加载bash程序中的函数。
```

4、PTRACE

PTRACE 是 Linux 内核提供的一组用于进程跟踪的系统调用，它允许一个进程监视另一个进程的执行过程，读取或修改它的寄存器和内存等状态。PTRACE 通常被用于进程调试、安全检测、进程间通信等场景。

PTRACE 提供了一些基本的功能，包括：

- PTRACE_ATTACH：将一个进程附加到另一个进程上，使得附加进程可以对目标进程进行跟踪。
- PTRACE_DETACH：将一个进程从另一个进程上分离，停止跟踪目标进程。
- PTRACE_SYSCALL：让被跟踪的进程执行下一个系统调用，并暂停在系统调用返回前。
- PTRACE_GETREGS / PTRACE_SETREGS：读取或修改目标进程的寄存器状态。
- PTRACE_PEEKDATA / PTRACE_POKEDATA：读取或写入目标进程的内存数据。
- PTRACE_GETSIGINFO / PTRACE_SETSIGINFO：读取或修改目标进程的信号处理器状态等等。

```
//PTRACE实例：
ptrace(PTRACE_TRACEME,…); ptrace(PTRACE_PEEKUSER,…);
ptrace(GET/SETREGS)
```

5、FUSE

FUSE将所有的文件系统操作放在用户空间中进行，可以提高系统的安全性和可扩展性。由于内核空间的代码难以调试和验证，而且内核空间的代码一旦崩溃，就会导致整个系统崩溃。因此，将文件系统操作放在用户空间中，可以更容易地进行调试和测试，也可以减少系统崩溃的风险。

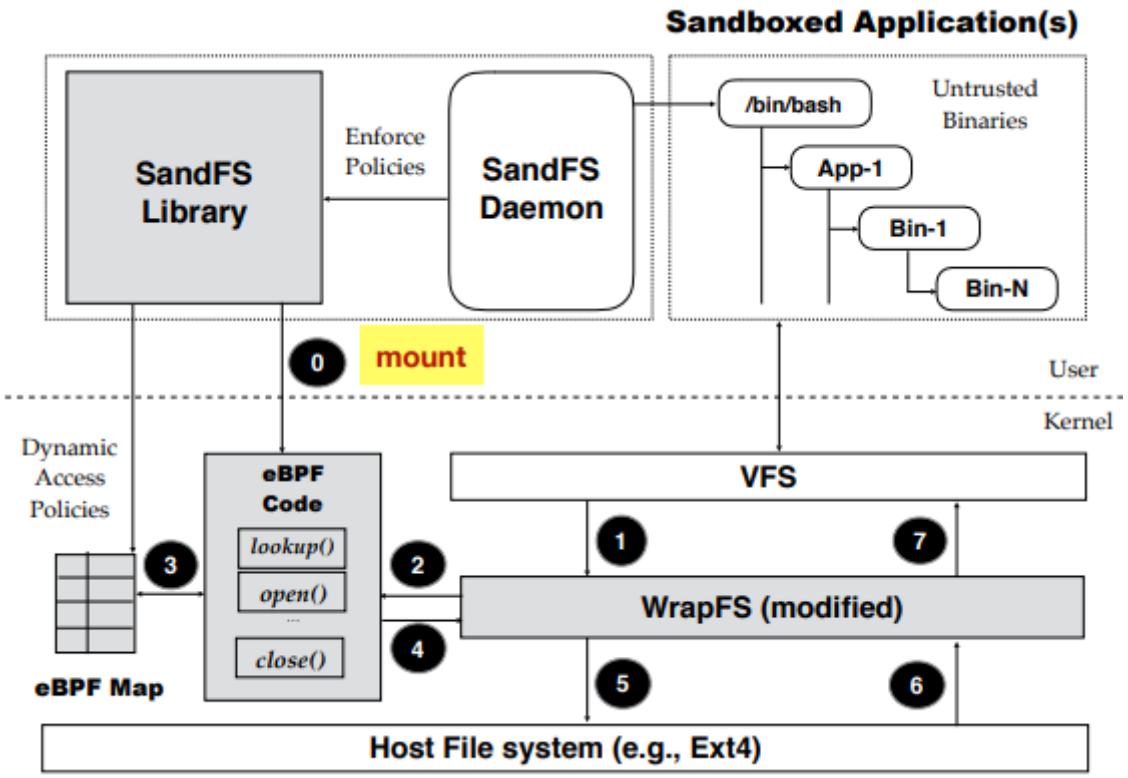
另外，将文件系统操作放在用户空间中还可以提高系统的可扩展性。由于内核空间的资源有限，当需要处理大量文件系统操作时，容易出现内存耗尽、死锁等问题。而将文件系统操作放在用户空间中，则可以根据需要动态分配资源，提高系统的并发性和吞吐量。

File System Sandboxing Techniques	Dynamic Policies	Unprivileged Users	Fine-grained Control	Security Needs	Performance Overhead
UNIX DAC	X	✓	X	Inadequate	–
SELinux (MAC)	✓	X	✓	✓	–
Chroot/Namespaces	X	X	X	Isolation	–
LD_PRELOAD	✓	✓	X	Bypass	Low
PTRACE	✓	✓	X	TOCTTOU	< 50%
FUSE	✓	✓	✓	✓	**< 80%**

SandFS的创新点

- 非特权用户和应用程序
- 细粒度的访问控制
- 动态（编程式）自定义安全检查
- 可堆叠（分层）保护
- 低性能开销

SandFS的架构



1、工作流程:

在挂载SandFS之前，用户空间必须将包含安全扩展的eBPF程序加载到内核并在SandFS驱动程序中进行注册【0】。这是通过执行bpf_load_prog系统调用来实现的，该系统调用会在内核中调用eBPF验证器以检查扩展的完整性。如果验证失败，则会丢弃扩展，并通知用户空间发生错误。如果验证步骤成功，并且启用了JIT引擎，则处理程序将由JIT编译器处理以生成准备执行的机器汇编代码。

当上层文件系统（例如VFS）向驱动程序发送请求【1】时，每个请求首先被传递到相应的扩展处理程序以强制执行安全策略【2】。处理程序可能会参考在eBPF映射中维护的共享数据【3】来允许或拒绝请求，如果需要的话（例如更改open（）中的mode参数的值）甚至可以操纵参数。处理程序返回的安全检查结果传回驱动程序【4】。被拒绝的请求将立即以相应的错误代码（errno）交付给用户空间【7】，而允许的请求将转发到较低的文件系统【5】。需要注意的是，SandFS驱动程序仅充当较低文件系统和来自用户空间的已注册安全内核扩展之间的薄层拦截层。因此，它不执行任何I/O操作或尝试自己处理请求【6】。来自较低文件系统的结果只是传递给用户空间【7】。

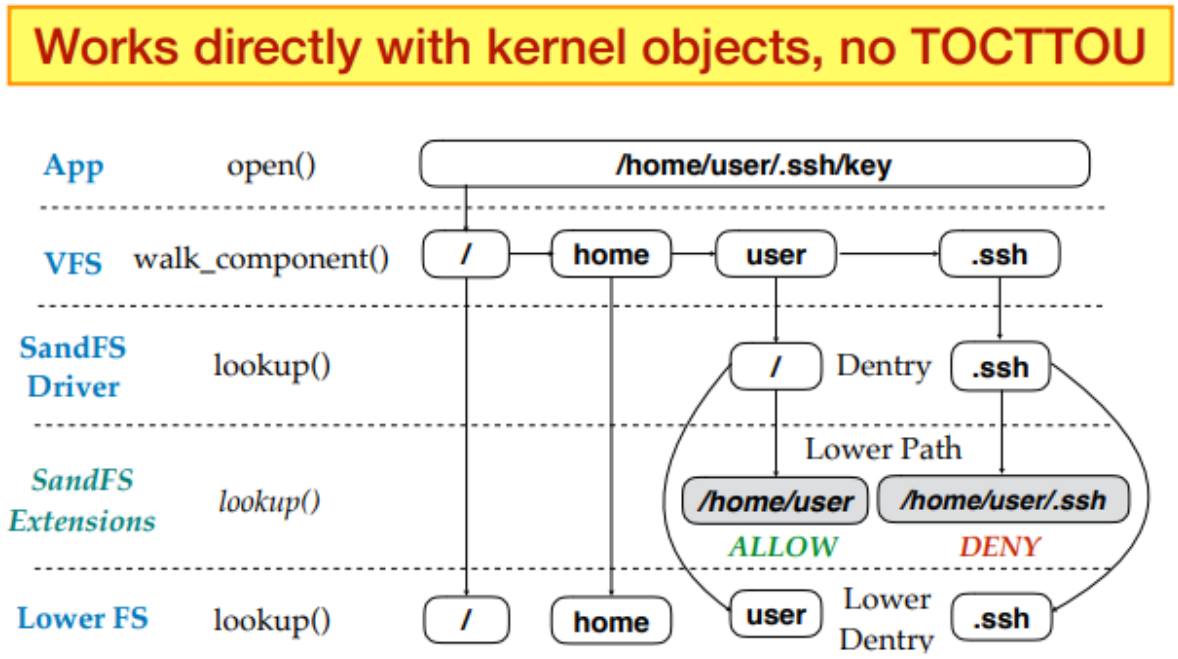
注意：基于WrapFS的SandFS驱动程序

- 可堆叠的文件系统包装器层

  不执行I/O

  将请求转发到较低的文件系统（例如Ext4）
- 限制可堆叠层的数量（堆叠层数限制为2，避免栈溢出）
- 调用SandFS扩展以执行策略

2、SandFS扩展如何在VFS执行lookup操作期间执行访问策略的实际过程:



当在/home/user上挂载SandFS以隔离用户的工作空间时，每个VFS查找请求首先会传递到SandFS驱动程序中的查找实现。驱动程序的查找函数首先通过调用底层文件系统来获取所请求组件（例如.ssh）的下层dentry和其（下层）inode，然后调用dentry_path()来构建其完整路径（例如/home/user/.ssh）。在此过程中，任何符号链接或特殊名称（例如.和..）都会被解析。然后，将计算出的完整路径作为参数传递给sandfs_lookup扩展来执行注册的检查。如果扩展成功返回值（允许），SandFS驱动程序将为路径组件创建一个（上层）inode并返回给VFS。

3、实际代码实例：

```
1 int sandfs_lookup(void *args) {
2
3 /* 获取路径 */
4 char path[PATH_MAX];
5 ret = sandfs_bpf_read(args, PARAM0, path, PATH_MAX);// 从参数中读取路径到path数组
6 if (ret) return ret;// 如果读取出错，返回错误码
7
8 /* 在映射表中查找该路径是否标记为私有 */
9 u32 *val = bpf_map_lookup(&access_map, path);
10
11 /* 示例检查：禁止访问私有文件 */
12 if (val) return -EACCES;// 如果该路径被标记为私有，返回访问权限错误码
13
14 return 0; /* 允许操作 */
15 }
```

sandfs_lookup将查询BPF映射以获取正在使用的路径，作为lookup()文件系统操作的一部分；如果在映射中找到了该路径，它将返回-EACCES，从而为用户空间提供了一种限制访问沙盒目录的方法。

```
1 int sandfs_open(void *args) {
2
3 /*  获取文件打开模式 */
4 u32 mode;
5 ret = sandfs_bpf_read(args, PARAM1, &mode, sizeof(u32));// 从参数中读取模式到mode变量
6 if (ret) return ret;// 如果读取出错，返回错误码
7
```

```
8 /* 示例检查：不支持文件创建 */
9 if (mode & O_CREAT) return -EPERM;// 如果模式中包含O_CREAT标志，返回无权限错误码
10
11 /* 示例强制：重写参数以强制使用只读模式 */
12 mode = O_RDONLY;
13 ret = sandfs_bpf_write(args, PARAM1, &mode, sizeof(u32));// 将只读模式写回参数
14 if (ret) return ret;// 如果写回出错，返回错误码
15
16 return 0; /* 允许访问 */
17 }
```

sandfs_open将检查open()操作中指定的模式，拒绝带有O_CREAT的模式，并将其余模式更改为O_RDONLY。

SandFS结果评估

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.

作者比较了在ext4和SandFS上执行几种不同类型的操作所需的时间。

- 创建一个4.17内核的.tar.gz文件显示了最低的开销（4.57%，61.05秒与63.84秒）。
- 解压缩和展开tar文件的开销最大（9.75%，5.13秒与5.63秒）。
- 而编译内核（make -j 4 tinyconfig）的开销为9.28%（27.15秒与29.67秒）。

总结

本文介绍了一种轻量级的文件系统隔离框架，称为SandFS。它被设计为一个可堆叠的文件系统，供非特权用户和应用程序使用，以实施自定义安全检查，保护私有数据免受不受信任和恶意二进制文件的攻击。SandFS的开销不到10%。它的源代码可在GitHub上获得。