VUOS: a User-space Hypervisor based on
System Call Hijacking

Published Version:
Bassi, L., Berardi, D., Davoli, R. (2024). VUOS: a User-space Hypervisor based on
System Call Hijacking. Cham : Springer [10.1007/978-3-031-68738-9_23].

Availability:
This version is available at: https://hdl.handle.net/11585/999997 since: 2025-01-04

Published:
DOI: http://doi.org/10.1007/978-3-031-68738-9_23

VUOS: a User-space Hypervisor based on System Call Hijacking

Luca Bassi[1,2][0009−0005−5191−4309], ✉ Davide Berardi[3][0000−0002−2473−2439], and Renzo Davoli[4][0000−0002−3377−8594]

[1] INFN-CNAF, Bologna, Italy luca.bassi@cnaf.infn.it
[2] GARR, Roma, Italy
[3] Universitas Mercatorum davide.berardi@unimercatorum.it
[4] Alma Mater Studiorum – Università degli Studi di Bologna renzo@cs.unibo.it

Abstract. VUOS (View based OS) is a virtual operating system that permits to give to each process a different view of the underlying system, i.e. access only some specific directories or specified system calls. This is currently obtained intercepting system call requests leveraging the `ptrace` system call. All the system call requests are forwarded to a VUOS hypervisor (`umvu`) that runs in user space. For each request, the hypervisor can use its own system call implementation or forward it to the Linux kernel. Running the hypervisor in the user space, like any other user process, increases the security because, in the case of sandbox escape bugs, the potentially malicious process obtains only limited system access. This approach can be seen as an extension to system calls of the FUSE behavior. FUSE (File system in Userspace) is a user-space file system framework that permits users to implement their own file systems without patching the Linux kernel. A kernel module forwards requests for each FUSE-mounted file system to its corresponding user process. FUSE is specific for file systems. VUOS instead permits to implement several system services in user space, like virtual devices, virtual networking and file systems (including reimplementing FUSE as a VUOS module).

Keywords: System Calls · User space · Virtualization · Security · Sandbox.

1 Introduction

In the modern world, security and sandboxing are always more important. Programs continue to become more and more complex, so the attack surface grows. Security and privacy concerns require reducing the portion of the system that a program is able to see and interact with. System calls are the means by which processes can request services from the kernel. It's possible to forward system calls to user-space processes. In this way, the user-space hypervisor can provide each process with a different *view* of the underlying environment, such as file systems, devices, networking, etc. VUOS permits the implementation of various kernel features in the user space, for example namespaces, chroot, file systems, networking support, etc. VUOS intercepts system calls and, when the access of a virtualized entity is required (e.g. file system, network, device), it forwards the system call to the specific module implementing such a virtualization (e.g. `vufuse`, `vunet`, `vudev`). This could make it easier to move features from kernel to user space.

This paper is structured as follows:

- In Section 2, we will compare the principal solutions available at the moment.
- In Section 3, we will introduce the `ptrace` system call and how it's possible to filter system calls using seccomp.
- In Section 4, we will present the implementation of the VUOS hypervisor.
- In Section 5, we will briefly introduce the `umvu` hypervisor usage and some of the modules and submodules available for VUOS.
- In Section 6, we will present some useful use cases of VUOS.
- In Section 7, we will compare qualitatively and quantitatively VUOS with other virtualization technologies.
- In Section 8, we will highlight some related works, such as FUSE.

2 State of the art

Aside from the implementation of the modules, we propose a list of use cases and a comparison with common industrial-grade virtualization software. Virtualization can be divided into four groups, namely: Language virtualization, Full virtualization, OS-level virtualization and partial virtualization.

The first one is the most distant from the implementation of VUOS, for instance Java, Python, JavaScript languages implement a virtual machine to interpret and map high-level code to low-level implementation. Security is therefore granted by the impossibility of a guest program to execute specific actions. For instance, the JavaScript virtual machine implemented in the browser cannot interact, open and modify files present on the host system.

Full virtualization shares the same concept as Language virtualization but moves from a specific, high-level language to the interpretation of native code (*opcodes*) of a CPU. Also, to create an entire virtual machine, it can implement interfaces that behave exactly like peripherals or devices. For instance, popular virtual machine implementations like Qemu, KVM, and VirtualBox fall into this category. Game Console Emulators also belong to this category. This virtualization is the technology enabler for cloud and edge computing. A lot of new approaches, first of all 5G networks and Industry 5.0 uses this technology as a core enabler. Due to this popularity, over the course of the years, a lot of optimizations were developed [1].

OS-level virtualization is a long-time theoretical [2] concept which gained a lot of traction in implementations and industry over the last decade. Docker, over all the other implementations, is a software suite to deal with containers. Containers are a concept really similar to virtual machines, the main idea behind them is that the host Kernel is in charge of replying to system requests of a process (e.g. retrieving the content of a directory from the backend storage). The system can therefore alter the result of the reply, hijacking the request to a confined environment. The set of these confined environment compose the "virtual-machine". In the Linux system, this technology is called "namespace". Aside from storage, at the moment of writing (in Linux Kernel 6.8.7), there are eight different types of namespaces. For instance, a virtual machine can now be defined in terms of storage mount points, network interfaces, time, user mappings and so on. Orthogonal to namespaces, resource limitations must be implemented (e.g. how many CPU slices a virtual machine can receive and how much Memory it can use). This limitation is implemented by a different technology called control group (*cgroup*). The combination of these two technologies enables the Linux kernel to confine process in terms of what they can see (namespaces) and what they can use (cgroups), effectively creating "virtual-machines" [3]. It is easy to see that this level of confinement and segmentation offered by containers is not extensible. If we want to extend the concept of "virtual-machine" to other parts of the system (e.g. changing the behavior of the "reboot" system call) we need to create a different namespace or extend an existing one to this behavior. This will also require modifications inside Kernel code, with devastating consequences such as Kernel Panic or security violations if the code is not analyzed in detail.

To overcome these problems, partial virtualization can be implemented. By changing the behavior of specific system calls, the application can be tricked to work in a different environment. That is, partial virtualization, as OS-level one, changes what the process sees, not what the system interacts with.

3 Background

The ptrace system call. The `ptrace` [4] system call permits one process to observe and control the execution of another process. This is useful to implement breakpoints in debuggers and to trace system calls. The process to be observed has to be connected to the tracer. The traced program stops at every signal. The tracer can use the `waitpid` or similar system calls to wait for it: the returned value contains the information with the cause of the stop. At this point, the tracer can inspect and modify the traced program using the requests of the `ptrace` system call: for example modifying the signal or ignoring it. When tracing is no longer needed, it is possible to detach the tracer program.

Filtering system calls with seccomp. Normally, a process can request all the kernel-provided system calls, but usually it doesn't use all of them. To restrict which system calls a process can request, the Linux kernel introduced the seccomp [5]. Seccomp has two modes: strict and filter. In strict mode, the process can only request `read`, `write`, `exit`, `sigreturn` system calls. If another system call is requested by the process in strict seccomp mode, it will be terminated with the `SIGKILL` signal. In filter mode [6], it is possible to write classic BPF programs that receive the system call requests and, examining the system call number and arguments, determines if each request has to be forwarded to the kernel or not. At the end of the execution, the BPF program must tell the kernel what to do with the system call request: allow the system call, skip the system call, terminate the process, send a `SIGSYS` signal to the process, or notify a `ptrace` tracer.

4 VUOS hypervisor implementation

VUOS is a partial, modular, system call virtual machine, like its predecessor View-OS [7]. This means that the processes are executed natively, but the hypervisor intercepts the system call requests and filter them: they can be processed by VUOS or be forward to the host kernel. VUOS permits to give to each process or thread a different view[5]: the execution environment seen by a process includes some resources directly provided by the host kernel and others visible only to the process. The current VUOS hypervisor implementation (`umvu`) uses a seccomp filter to forward the system call requests to a `ptrace` tracer. Initially, the VUOS hypervisor just blindly forwards all the system call requests to the underlying kernel. To virtualize some system calls, it's necessary to load modules. The hypervisor currently implements 144 Linux system calls. Each module provides up to 56 system calls, VUOS implements 121 system calls based on those provided by modules. The code of modules can be simplified in this way. 23 system calls are directly implemented by the VUOS hypervisor (e.g. calls like `getcwd` or `dup`, because the VUOS hypervisor handles the management of the current working directory and the mapping between file descriptors and files as seen by the modules). Many modules have been implemented: `vunet`, `vudev`, `vufs`, `vufuse`, `fuse`. The `fuse` module is compatible with FUSE, so it doesn't require a patched kernel. It's important to note that `umvu` is just a user process and doesn't require kernel patches or special privileges, so it doesn't increment the attack surface. In particular, `umvu` can be divided into three parts: Core hypervisor, Modules, and Submodules. The core hypervisor intercepts all the system calls using `ptrace`. If the system call request is related to a virtualized service, `umvu` forwards the request to the specific module; otherwise, the system call is processed by the kernel as usual. Each module must register its boundary of responsibility (i.e. system call numbers, path names, file descriptors, etc.) to the core hypervisor so the latter can know which module forward each system call to. Each module implements a subset of the system call API. For example, when a process issues a read system call on a virtualized file, the module registered for that path receives a system call request with the same signature. APIs between modules and submodules vary, for example the file system one is compatible with the `fuse` protocol. The `vu_insmod` command loads VUOS modules. `vu_lsmod` and `vu_rmmod` lists all the currently loaded modules or removes one or more modules, respectively. A more detailed description of the implementation is pictured in Figure 4. In this picture, it's visible how a system call (e.g. `open`) (1) is requested to the kernel, (2) it is sent by leveraging `ptrace` to the hypervisor, (3) it is dispatched to the correct module designed for the handling (if present),

[5] VU was chosen because sounds like "view".

(4) the reply is computed by the module, (5) is sent back to the kernel, and, finally, (6) it is retrieved by the application.

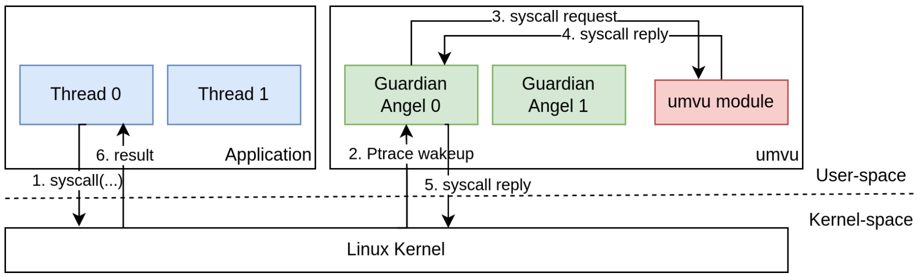


Fig. 1. Implementation details of umvu with a traced application with two threads, one module and the internal guardian angels.

We argue that this design can make the system more isolated in terms of cybersecurity. Applications can be isolated in a way similar to containers but with more flexibility, creating fine-grain containerization. As described in Section 9, we will focus on this analysis in a future work. This design requires a specific handling of `select` and `poll` system calls. That is, if a thread requests a blocking system call, how can the hypervisor continue its execution and handle the other threads? This is implemented by using a companion thread, mapped with a 1-to-1 mapping with the application thread. This thread is designed to handle blocking system calls for the application.

5 `umvu` usage, modules and submodules

The following command runs the hypervisor starting a Bash shell: `umvu bash`. The `vu_insmod` command is used to load a specific module, for example to load the `fuse` one: `vu_insmod fuse`. In UNIX and derivatives (such as Linux) "everything is a file", which means that processes can name all the resources of the system as path names and file descriptors, including files (obviously), input/output devices, sockets and also running processes. For this reason, it seemed natural in VUOS to use the `mount` system call and command to activate a virtualization, that is a layer between the process and the kernel that modifies the former "view" of the underlying system. In particular, in VUOS it's a best practice to use the `vumount` command instead of `mount`; in fact, `mount` performs unnecessary checks, resulting in unreal failure cases [8]. We will now present some of VUOS modules implemented by the VirtualSquare project [9].

FUSE support in VUOS. FUSE, described in 8, permits to implement file systems in user space. VUOS implements the `fuse` support in two modules: the `fuse` module and the `vufuse` one. The former one provides an implementation compatible with Linux FUSE kernel module and use the standard `/dev/fuse` device to communicate, but in this case no kernel support is needed because all the required code is made available by the hypervisor module. The `fuse` module is virtually compatible with any FUSE helper. `vufuse` also provides FUSE compatibility without the need of kernel support, but it requires FUSE support helpers to be compiled as custom shared libraries. This latter support can be used while performances are a concern, due to the fact that it's faster due to the removal of Kernel support. The mounted file systems are visible only by processes running in the VUOS hypervisor. There are some submodules to support different file systems: `fuse-ext2` and `vufuseext2`: for ext2, ext3 and ext4 file systems; `fusefatfs` and `vufusefatfs`: for FAT12, FAT16, FAT32 and exFAT file systems; `archivemount` and `vufusearchive`: for archives such as .tar, .zip, .iso, etc.

Changing processes' file system view. The `vufs` module can be used to move, merge, and change ownership and permissions of files and directories. This module supports four operational modes:

bind this is the default mode and permits to substitute a file/directories with another one;

merge in this mode two directories are merged, files from the first directory are added to the second one and shadows the latter if files coincide, files from the first directory are read only and all the updates are made to the second one;

cow in this mode two directories are merged, but all the updates are made to the first one, when a change to a file from the second one is made, this file is copied to the first directory before making the modification;

mincow similar to the previous one, but changes are copied from the second directory to the first only when it's impossible to apply directly to the second directory.

Virtual devices in VUOS. In UNIX, special files are used to communicate with devices. The `vudev` module permits to create virtual devices, that is, to create these special files. For example, the `vudevramdisk` submodule can be used to allocate some RAM and make it available as a virtual disk. In this memory area, it is possible to initialize a file system that appears to the virtualized process as a normal disk. Furthermore, the `vudevpartx` submodule can be used to manage MBR and GPT partitioning schemes. This can be particularly useful when you want to mount a disk image and make changes to it.

Virtual network stacks in VUOS. The `vunet` module can be used to create virtual network stacks. These are loaded with `mount` and, like other devices, they appear like special files. VUOS permits to select which stack to use with the `vustack` command. The `vunetioth` submodule permits adding all the network stacks supported by libioth [10], for example `vdestack` and `picox`. There are also two dedicated submodules for these network stacks: `vunetvdestack` for the `vdestack` and `vunetpicox` for the `picoxnet` one. The main difference in that the `picoxnet` stack is completely implemented in user space, while `vdestack` uses the TCP/IP stack of the Linux kernel through the definition of a namespace. It's also possible to completely disable the networking support by mounting the `vdenetnull` submodule.

unrealuidgid module. The `unrealuidgid` module permits to virtualize user and group identification. In particular, the `vusu` command permits to start a shell as a fake root user.

vumisc module. The `vumisc` module supports virtualization of system calls not related to specific conditions (like path names, protocol families and so on). For example, the `vumiscdate` submodule provides the processes with a virtual view on the time providing virtual implementations of `clock_getres`, `clock_gettime` and `clock_settime` system calls.The `vumiscdate` submodule permits, accessing three virtual files, to: (1) change the offset (the difference between the real and virtual time); (2) change the frequency (the speed of the clock); (3) change the base (the base time for the frequency).

6 Use cases

Avoid overuse of root. During normal interaction with the Linux system, the overuse of the root user or `sudo` is quite common. That is due to the fact that some basic system operations require elevation of permissions, even when not strictly required. One example is the `mount` command that requires almost always the privileges of the root user. For example, modifying a file inside a partition of a disk image file should not require any special privileges because it's just a sophisticated editing of the file content. On the contrary, in normal Linux usage, the standard way to navigate in the file contents using the file system abstraction is to mount the disk image and the mount operation requires root permission. The following examples show how to modify a file (`/etc/passwd`) in an ext4 partition inside a disk image in three different way: (a) using standard tools; (b) using `libguestfs`; (c) using VUOS.

Using the `losetup` requires administration privileges (root):

```
su
losetup --partscan /dev/loop0 /tmp/diskimage
fuse-ext2 -o rw+ /dev/loop0p2 /mnt
vi /mnt/etc/passwd
umount /mnt
losetup -d /dev/loop0
```

Using the root user, every mistyped command or error in a script may lead to damages. To overcome the usage of root privileges, `libguestfs` was developed. This tool permits accessing and modifying disk images. Using `libguestfs`, the previous operation can be made by doing:

```
guestfish --rw -a /tmp/diskimage
launch
```

```
mount /dev/sda2 /
edit /etc/passwd
exit
```

The main drawbacks of `libguestfs` are: (a) it is limited to some disk image formats; (b) it has a specific syntax; and (c) not all the features have been reimplemented. Using VUOS and the `vudev` and `fuse` modules, it's possible to use the same commands normally used as a root, but without the root privileges required.

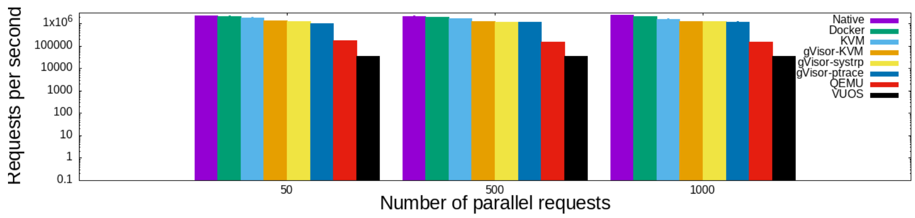
```
vu_insmod vudev fuse
vumount -t vudevpartx /tmp/diskimage /dev/sda
fuse-ext2 -o rw+ /dev/sda2 /mnt
vi /mnt/etc/passwd
vuumount /mnt
vuumount /dev/sda
```

Running a Linux distro in a virtual chroot. The `umvu` hypervisor implements the `chroot` command. It's possible to leverage that to run a Linux distribution inside a virtual chroot environment. Firstly, a rootfs of a Linux distribution is needed. The `vufs` module is used to mount a virtual root home directory and, using the `vusu` command provided by the `unrealuidgid` module, it's possible to interact with the virtualize shell as the root user. Finally, a simple `chroot` to the location of the rootfs permits running the downloaded Linux distribution.

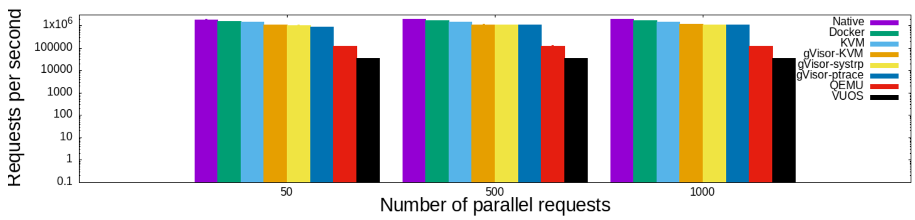
7 Results

To validate the partial virtualization approach, we propose two kinds of results: qualitative and quantitative metrics. For the qualitative one, we present it in Table 1, focusing on the comparison in terms of extensibility and results. We compared our software with reference implementation of the families of virtualization. Namely: Qemu in translation-mode (TCG) and its hardware-enabled optimized counterpart, KVM for Full Virtualization; Docker, implemented using control groups and namespaces for OS-Level virtualization; and a direct counterpart of VUOS: gVisor [11], developed by Google. It is easy to see how the modular design of VUOS is a clear advantage over the other competitors, giving developers the maximum possible extensibility over the system interfaces.

Partial virtualization is therefore a little step towards micro-kernels, which historically suffered in terms of performance. That is also true for this kind of virtualization, but not in every case. To visualize numerical data with this kind of feature, in Figure 2 we propose a comparison with the software suite presented in Table 1. These implementations were tested on an AMD EPYC 7443 CPU with 32GB of RAM for every confinement. On this machine, the various applications were installed and configured with default settings. To test performance, the hypervisors and virtualization systems were tested by using a benchmarking



(a) Get Requests



(b) Set Requests

Fig. 2. Comparison of performance results of redis benchmark provided by phoronix-test-suite. On x-axis the number of parallel threads are pictured, divided in 50, 500 and 1000 threads. On y-axis is plotted the number of requests per second in logarithmic scale.

suite called "phoronix-test-suite" [12]. This suite can run a number of different benchmarking tools. To keep our test consistent and comparable with the one used by most analysis, we focused on the "redis" test, which uses the popular NoSQL Database. This test runs a lot of different parallel requests, and calculates how many requests per second the system can handle. From Figure 2, we can evince that the systems are basically comparable, with Docker winning over other virtualizators and Qemu clearly losing due to its translation approach. gVisor, on the other hand, takes three approaches: one ptrace based, the less performant; one based on seccomp; and one based on KVM [13]. VUOS tends to be the less-performant solution due to its implementation; due to its focus on its elasticity, performance is not the core object of the project.

8 Related works

FUSE. FUSE [14] is an interface for user-space programs to export a file system to the Linux kernel. FUSE permits unprivileged users to create and mount their own file systems. It's divided into two parts: (1) a kernel module that forwards requests for each user-space file system to the corresponding user-space process; and (2) a user-space library that provides helper functions to communicate with the kernel module, e.g. mounting and unmounting the file system, receiving requests and sending responses, etc. The communication between the kernel module and the user process takes place through `/dev/fuse`. To write a

Table 1. Comparison between similar software: ○ indicates that the problem is absent and ● that the problem is present.

Problem detected	VUOS	Qemu (TCG)	KVM	Docker	gVisor
To create extensions it is required to modify critical (e.g. Kernel) code	○	○	●	●	○
To create extensions it is required to modify core part of the application	○	●	●	●	●
If compromised can lead to other guest systems control	○	○	●	●	○
It is required to implement ISA-specific code or peripherals to extend the VM	○	●	●	○	○

new file system, it's possible to implement a program to mount and respond to the requests (i.e. open, read, write and close) of the file system. This program must use the kernel-user space protocol of FUSE. `libfuse` [15] provides helper functions to simplify this communication. After the file system is mounted, all the requests are forwarded to the user-space program that handles them and sends the responses using the same protocol. This could be useful when you want to create virtual file systems, for example when you want to mount a remote file system as a local one (i.e. Rclone [16] and SSHFS [17]).

Other projects. A number of projects similar to VUOS were identified in literature. NUSE (Network Stack in Userspace) [18] exposes the Linux kernel network stack in user space in order to achieve extensibility and network stack personalization. gVisor [19] is an application kernel that permits to isolate programs from the underlying kernel. It reimplements the majority of the Linux system call interface. cVMs [20] permit to isolate applications using memory capabilities, so all share the same virtual address space, but with only the capabilities to access their own memory. It relies on CHERI capabilities hardware support [21]. DuVisor [22] is a user-space hypervisor that handles all the VM operations without kernel intervention, that is necessary only for the initialization. It relies on hardware-specific instructions available only on experimental RISC-V-based CPUs. Qemu [23] can be used for user-mode process level emulation to execute programs compiled for one architecture on a CPU with a different one. In Section 7, we pinpoint the main differences with industrial implementations, showing how VUOS can be placed in the software ecosystem of Linux-based machines.

9 Conclusions

VUOS is a modular solution for partial virtualization. Using the VUOS hypervisor (`umvu`) it's possible to give to each process a different view of the underlying system. VUOS intercepts the system call requests and, depending on the loaded modules, forwards them to the specific handler or to the kernel. In the future, we want to explore the possibility to use eBPF instead of the `ptrace` support to intercept system call requests [24,25]. Another topic that we are considering is the optimization of the performance comparing our implementation that uses `ptrace` with other projects that use different technologies. For instance, comparing our solution with the implementation of gVisor, it can be deduced that there are performance gaps that could be fulfilled to improve the tool. Analyzing the software, the performance differences can be due to the architectural requirements of guardian angels (as described in Section 4) employed by VUOS but not by gVisor. In the future, a rewrite in a memory-safe language (e.g. Rust or Go) can be done. Implementing kernel features in the user space can increase the security because the attack surface of the kernel doesn't increase. VUOS permits to avoid the overuse of the root user and fine-grain containerization. A future work under development is the analysis of the threats and attacks neutralized by VUOS. The source code of VUOS is available at `https://github.com/virtualsquare/vuos`.

Acknowledgments. This work was partially supported by project SERICS (PE00000014) under the MUR National Recovery and Resilience Plan funded by the European Union - NextGenerationEU.

Disclosure of Interests. Author Renzo Davoli is a member of committee for TOAST 2024.

References

1. Xu, Q., Patel, R.: Hardware Virtualization. In: Data Plane Development Kit (DPDK), pp. 219–227. CRC Press (2020). https://doi.org/10.1201/9780429353512-13
2. Watada, J., Roy, A., Kadikar, R., Pham, H., Xu, B.: Emerging Trends, Techniques and Open Issues of Containerization: A Review. IEEE Access **7**, 152443–152472 (2019). https://doi.org/10.1109/ACCESS.2019.2945930
3. Reshetova, E., Karhunen, J., Nyman, T., Asokan, N.: Security of OS-Level Virtualization Technologies. In: Bernsmed, K., Fischer-Hübner, S. (eds.) Secure IT Systems. pp. 77–93. Springer International Publishing, Cham (2014). https://doi.org/10.1007/978-3-319-11599-3_5
4. ptrace(2) - Linux manual page, https://www.man7.org/linux/man-pages/man2/ptrace.2.html, last accessed 2024/05/02
5. Edge, J.: A seccomp overview, https://lwn.net/Articles/656307/, last accessed 2024/05/02
6. Seccomp BPF (SECure COMPuting with filters), https://www.kernel.org/doc/html/latest/userspace-api/seccomp_filter.html, last accessed 2024/05/02
7. Gardenghi, L., Goldweber, M., Davoli, R.: View-OS: A New Unifying Approach Against the Global View Assumption. In: Bubak, M., van Albada, G.D., Dongarra, J., Sloot, P.M.A. (eds.) Computational Science – ICCS 2008. pp. 287–296. Springer Berlin Heidelberg, Berlin, Heidelberg (2008). https://doi.org/10.1007/978-3-540-69384-0_34
8. mount(8) - Linux manual page, https://www.man7.org/linux/man-pages/man8/mount.8.html, last accessed 2024/05/02

9. VirtualSquare wiki, https://wiki.virtualsquare.org, last accessed 2024/05/02
10. Davoli, R.: VXVDEX: Internet of threads and networks of namespaces. In: 2017 IEEE International Conference on Communications (ICC). pp. 1–6 (2017). https://doi.org/10.1109/ICC.2017.7996595
11. Anjali, Caraza-Harter, T., Swift, M.M.: Blending containers and virtual machines: a study of firecracker and gVisor. In: Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments. p. 101–113. VEE '20, Association for Computing Machinery, New York, NY, USA (2020). https://doi.org/10.1145/3381052.3381315
12. Deshane, T., Shepherd, Z., Matthews, J., Ben-Yehuda, M., Shah, A., Rao, B.: Quantitative comparison of Xen and KVM. Xen Summit, Boston, MA, USA (2008)
13. gVisor Platform Guide, https://gvisor.dev/docs/architecture_guide/platforms/, last accessed 2024/05/02
14. Vangoor, B.K.R., Tarasov, V., Zadok, E.: To FUSE or not to FUSE: Performance of User-Space file systems. In: 15th USENIX Conference on File and Storage Technologies (FAST 17). pp. 59–72. USENIX Association, Santa Clara, CA (Feb 2017), https://www.usenix.org/conference/fast17/technical-sessions/presentation/vangoor
15. libfuse repository, https://github.com/libfuse/libfuse, last accessed 2024/05/02
16. rclone mount, https://rclone.org/commands/rclone_mount/, last accessed 2024/05/02
17. SSHFS repository, https://github.com/libfuse/sshfs, last accessed 2024/05/02
18. Tazaki, H., Nakamura, R., Sekiya, Y.: Library Operating System with Mainline Linux Network Stack. In: netdev0.1 (2015), https://netdevconf.info/0.1/papers/Library-Operating-System-with-Mainline-Linux-Network-Stack.pdf
19. gVisor Homepage, https://gvisor.dev/, last accessed 2024/05/02
20. Sartakov, V.A., Vilanova, L., Eyers, D., Shinagawa, T., Pietzuch, P.: CAP-VMs: Capability-Based isolation and sharing in the cloud. In: 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22). pp. 597–612. USENIX Association, Carlsbad, CA (Jul 2022), https://www.usenix.org/conference/osdi22/presentation/sartakov
21. Watson, R.N., et al.: CHERI: A Hybrid Capability-System Architecture for Scalable Software Compartmentalization. In: 2015 IEEE Symposium on Security and Privacy. pp. 20–37 (2015). https://doi.org/10.1109/SP.2015.9
22. Chen, J., et al.: DuVisor: a User-level Hypervisor Through Delegated Virtualization (2022). https://doi.org/10.48550/arXiv.2201.09652
23. Bellard, F.: QEMU, a fast and portable dynamic translator. In: 2005 USENIX Annual Technical Conference (USENIX ATC 05). USENIX Association, Anaheim, CA (Apr 2005), https://www.usenix.org/conference/2005-usenix-annual-technical-conference/qemu-fast-and-portable-dynamic-translator
24. Findlay, W., Somayaji, A., Barrera, D.: bpfbox: Simple Precise Process Confinement with eBPF. In: Proceedings of the 2020 ACM SIGSAC Conference on Cloud Computing Security Workshop. p. 91–103. CCSW'20, Association for Computing Machinery, New York, NY, USA (2020). https://doi.org/10.1145/3411495.3421358
25. Jia, J., et al.: Programmable System Call Security with eBPF (2023). https://doi.org/10.48550/arXiv.2302.10366