LETTER

Understanding File System Operations of a Secure Container Runtime Using System Call Tracing Technique*

Sunwoo JANG[†a)], ***Nonmember***, **Young-Kyoon SUH**[††b)], ***Member***, ***and*** **Byungchul TAK**[††c)], ***Nonmember***

SUMMARY This letter presents a technique that observes system call mapping behavior of the proxy kernel layer of secure container runtimes. We applied it to file system operations of a secure container runtime, `gVisor`. We found that `gVisor`'s operations can become more expensive than the native by 48× more syscalls for open, and 6× for read and write.

key words: *secure container runtime, system call, gVisor*

1. Introduction

As containers gain wider adoption in large-scale production environments, the security of containers becomes one of the most critical concerns. Even after years of recognizing the security issues with containers, container exploits still continue to impact us. For example, Azurescape is the first cross-account container exploit in the public cloud to use the `runc` container escape vulnerability (CVE-2019-5736), and the Linux kernel integer underflow vulnerability (CVE-2022-0185) can be exploited for the Kubernetes escape. To alleviate these security problems, various security-focusing container runtimes have emerged. There are `gVisor` [1], `Kata Containers` [2], `Sysbox` [3], X-Containers [4], μ`Kontainer` [5], `SCONE` [6], and `RunD` [7].

Their common approach is to keep the host kernel away from the direct reach of applications (via system call invocations) by adding a proxy kernel layer. System calls are intercepted and replaced with user-level library functions or reassembled with compatible, yet different system calls. Consequently, when a system call is invoked from the application, no actual system calls may be invoked at the host kernel or a different set of (i.e. translated and reassembled) system calls may be invoked. This approach is thought to be effective in reducing the attack surface in terms of system calls. However, there have been few studies that analyzed to what degree these system call manipulation mechanisms are applied and how, if any, system call arguments are modified. Such visibility can be invaluable since it can provide us with understanding and new insights as to how much impact there will be on the performance due to such mechanism for what kind of reasons. These insights can further be utilized to improve the logic of the proxy kernel layer from the developer's perspective. Moreover, discovered mapping or system call translation data can be compared with the expected design of the proxy kernel logic and any discrepancies can be identified. Possible points of optimization can be identified which were not easily visible in the source code.

This letter is the *first* work to conduct an in-depth study of a popular secure container runtime, `gVisor`, as our target and investigate the inner workings of the user-level proxy kernel, `sentry`, in terms of system call handling. Then, we aim to learn the nature of the performance degradation, interesting characteristics in the system call translation pattern, and implications to the containerized applications designed to run on `gVisor`. To this end, we have implemented a system call mapping discovery tool, called `sysmap`, that enables us to investigate the system call translation behavior along with arguments occurring within the `sentry` that sits between the application and the host kernel. Our `sysmap` tool operates by invoking a predefined set of system calls via test suite within the `gVisor` container, and tracking system calls observed at the host kernel.

Our findings indicated that the performance impact of the `gVisor` in file system operations could be as high as 48× for open and about 6× for read and write in terms of the number of system calls. For file-related syscall operations, `gVisor` consumes an average of 2.97 times more cycles than in Docker. Also, system-call intercepting accounts for about 26% of the overhead of most file-related operations in the `gVisor`. This enabled us to better understand the characteristics of performance overheads introduced by the current `gVisor` architecture. Based on this understanding, we were able to compare and verify previously reported benchmark results of `gVisor` [8].

2. Background and Motivation

2.1 Secure Containers

Containers are originally designed without security considerations, and the source of weakness is that container instances share the host kernel—various attack vectors exist in the container [9]. To mitigate this weakness, security-focused container runtimes with additional sandboxing have emerged. Representative examples of such container runtimes are

Manuscript received July 1, 2023.
Manuscript revised September 25, 2023.
Manuscript publicized November 1, 2023.

[†]The author is with Amazon Korea, Korea.
[††]The authors are with School of Computer Science and Engineering, Kyungpook National University, Daegu, 41566 Republic of Korea.

*This work was supported by BK21 FOUR project (4199990214394).

a) E-mail: seonwoo@amazon.com
b) E-mail: yksuh@knu.ac.kr (Corresponding author)
c) E-mail: bctak@knu.ac.kr (Corresponding author)

DOI: 10.1587/transinf.2023EDL8039

gVisor [1] and Kata containers [2]. We refer to the container runtime that has an additional isolation layer for security purposes as the "sandboxed container."

Our work targets the gVisor among several sandboxed containers. gVisor is an implementation of a mixture of several technologies and principles; privilege separation, defense-in-depth, attack surface reduction, and system call interposition (intercept and re-implement). First, as Fig. 1 shows, gVisor consists of sentry and gofer, and each has a different role and privilege. The sentry is a user-level proxy kernel that replaces the Linux kernel per container, and the gofer is responsible for handling file accesses between the container and the host. Second, by configuring sandboxing with two processes, even if an attack occurs in the container and the sentry is compromised, it is isolated from the gofer and the host kernel. Third, sentry and gofer have seccomp profiles that allow only system calls necessary for their own operations so that the attack surface to the host is smaller than that of general containers. Last, the sentry intercepts and re-implements the system calls invoked by the application. System calls invoked by the application are not directed to the host kernel as is.

System calls invoked by the application are intercepted by the sentry in one of two modes: ptrace and kvm. The sentry executes its own implementation of the intercepted system call and returns it to the application. This re-implementation is different for each system call, and in some system calls, it may be unavoidable to invoke the actual system calls of the underlying host kernel—in most cases, there is no additional host system call invocation. However, there are operations that require communication with the host kernel, and some of them are file manipulations. The sentry communicates with the gofer to access the host file system.

2.2 Motivation

There are previous studies on the system call analysis [10], [11]. However, they concern individual system calls rather than a sequence of system calls. Also, the analysis did not take into account the system call parameters. Figure 2 shows the sequence of system calls observed from the host kernel when open system call is invoked with different parameters in gVisor. Figure 2 illustrates that system call arguments may have a large effect on changing the behavior of sandboxed containers regarding system calls.

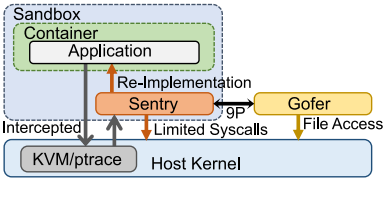

Fig. 1 Architecture of gVisor. Arrows indicate system call flows.

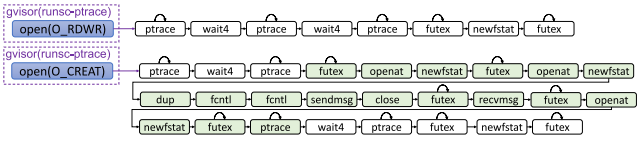

Fig. 2 A difference of host-observed system calls induced by a single open system call with different argument received by the proxy layer of gVisor, the sentry. O_RDWD and O_CREAT flags make large differences.

In gVisor, system calls invoked from the application do not directly reach the host kernel, but the user-level proxy kernel, sentry intercepts and runs a re-implemented version of the system call as much as possible. This user-level proxy kernel is divided into two processes, with separate privileges and functionalities.

For some file operations, host kernel and hardware operations are essential. The Linux file system is very complex and file-system bugs tend to be more serious than user-level bugs [12], and the application kernel of gVisor also implements its own file system. Also, it is known that privilege escalation is usually exploited through memory modification or file modification [13]. Therefore, we narrow it down to an analysis of various file-related system calls.

If source code is available, we could also apply static analysis techniques [14] to track system call invocations of operations from the codes. However, static analysis is known to suffer from missing or over-approximating system calls partly due to the challenges of following function pointers. In this work, we also aimed to develop a technique applicable to the case where source codes were unavailable.

3. Methodology

Our system call mapping discovery tool, termed sysmap, comprises three major components: *(i) Test suites* (test cases for 45 system calls), *(ii) Tracer* (the tracing logic within the host kernel), and *(iii) Trace analyzer* (trace parser and the system call mapping builder logic), as depicted in Fig. 3.

- Test suites: It is the collection of test cases, written in C, for each system call we investigate. Test case code varies the system call parameters. It also contains test cases where previously invoked system call affects the current system call under observation. For example, the write system call can behave differently depending on the flags of the previous open system call. In each test case code, we wrap the system call invocation line with trace-markers.
- Tracer: Collected system call traces are the sequence of system calls from when the tracer was turned on, not when the system call of the test was invoked. Thus, we need to identify the beginning and ending time of the target

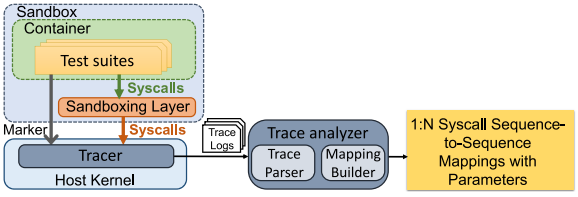

Fig. 3 Overall architecture of sysmap. The test suites consist of a set of valid system calls that reflect various parameters.

system call under test. To do so, we placed `ftrace` marker function calls before and after invoking the system call.
- Trace analyzer: This module is responsible for parsing the collected traces and outputting the mapping between the test system call and the observed list of system calls from the host kernel side.

The goal of our `sysmap` tool is to build an accurate and sophisticated 1:N system call mapping relationships reflecting the type of parameters.

4. Evaluation

4.1 Experiment Setup

Our analysis was performed on `gVisor` of release-20210906.0 in Ubuntu 20.04 (Linux kernel v5.14.12). We ran `gVisor` through Kubernetes v1.22.4—docker is also compatible. We used `ftrace` as a tracer of `sysmap`, and the marker of `sysmap` was implemented with the `trace_marker` function of `ftrace`. We additionally mounted `tracefs` (`/sys/kernel/tracing/`) when we run `gVisor` to use ftrace's `trace_marker` function.

We investigated 45 file-related system calls in eight categories for `gVisor` using `sysmap` as shown in Table 1. We primarily divided the file-related system call parameters into fd, path, flag, and mode. We focused on the flag and kept the same values for fd, path, and mode as long as there was no significant difference in mapping results. To reflect various parameters for each system call, we first conducted experiments on parameters for the most basic operation and parameters mentioned in NOTES or BUGS by referring to the Linux man pages. Then, when a result was different from the basic mapping result, we analyzed the code implementing that system call in `gVisor` and further tested other parameters that were likely to be relevant. We considered the `open` system call to be the model case while creating test cases of other system calls since it was used to specify the creation mode and state of a file. Also, in the case of a system call pair with `-at` appended such as `open` and `openat`, we included the `-at` system call in the mapping.

Table 1 List of file-related system call tested.

No	Category	System calls
1	File Operations	open(openat), close, renameat, mknodat, ftruncate, fallocate
2	Directory Operations	mkdirat, rmdir, getcwd, chdir, getdents64
3	Link Operations	linkat, symlinkat, unlinkat, readlinkat
4	File Attributes	umask, stat, fstatat, fstatfs, fchmod, fchownat, utimensat, access
5	File Descriptor	fcntl, dup, flock
6	Read/Write	read, readv, preadv, preadv2, pread64, write, writev, pwritev, pwritev2, pwrite64, lseek, sendfile
7	Synchronization	fsync, msync, sync_file_range
8	Monitoring File Events	inotify_init1, inotify_init, inotify_add_watch, inotify_rm_watch

4.2 System Call Mapping

4.2.1 `open` and `openat` System Calls

The `open` system call is a basic operation in file manipulation, and in `gVisor`, it is largely divided into two mappings—`open` and `openat` have the same implementation. The flag of the `open` system call can be divided into a creation flag and a status flag. The mapping result is affected by the creation flag in relation to fd (file descriptor) and not by the status flag. As shown in Table 2, when a new fd is needed, it communicates with `gofer` and interacts with the host kernel. But, if there is an fd to be assigned in the `sentry`, the `open` system call is handled by the `sentry` itself, and there is no interaction with the `gofer` or the host kernel. When it needs to allocate a new fd, the `gofer` invokes `openat(O_CREAT | O_RDWR | O_EXCL | O_NONFOLLOW | O_CLOEXEC)` to get the fd from the host kernel. After that, `gofer` gets the file properties through `newfstat` and `fcntl(F_GETFL)`, manipulates it for `gVisor` sandboxing with `dup(modulate fd)` and `fcntl(F_SETFL)`, and sends it to the `sentry` with `sendmsg`. The `sentry` receives a new fd with `recvmsg` and passes it to the application. Various `open` system call parameters are also mapped to these two forms. We observed that `O_RDWR` behaved the same as `O_CREAT` when `gVisor` is created and invoked for the first time—that is when there is no free fd. Moreover, when it is invoked together with the `O_TRUNC` flag, the `ftruncate` system call is additionally invoked, and when a symbolic link file is opened, `readlinkat` is additionally invoked.

4.2.2 `write` System Call

The `write` system call can be usually handled by the `sentry` itself because the `sentry` has an fd opened by the `open`

Table 2 Differences in mapping depending on parameters when invoking the `open(pathname, flags, mode)` system call. Flags followed by an |(or) sign in {}(braces) mean one of the flags.

Condition on fd existence	Flags	Syscall	Process
Present	O_RDWR, O_RDWR| {O_ASYNC| O_SYNC|O_NONBLOC| O_DIRECT|TMPFILE|O_PATH}	–	
Absent	O_CREAT, O_CREAT| {O_SYNC| O_NOFOLLOW|O_SYNC}, first invoked O_RDWR	openat	gofer
		openat	gofer
		newfstat	gofer
		openat	gofer
		newfstat	gofer
		dup	gofer
		fcntl	gofer
		fcntl	gofer
		sendmsg	gofer
		close	gofer
		recvmsg	sentry
		openat	gofer
		newfstat	gofer

Table 3 Differences in `write(fd, buf, count)` system call mapping according to previous `open` system call's parameter. Flags followed by an |(or) sign in {}(braces) mean one of the flags.

Flags	Syscall	Process	Syscall	Process
O_RDWR, O_WRONLY	pwrite64	sentry		
O_RDWR| {O_SYNC| O_DSYNC|O_DIRECT}	pwrite64	sentry		
	openat	gofer	utimensat	gofer
	close	gofer	epoll_pwait	gofer
	epoll_pwait	sentry	epoll_pwait	gofer
	fsync	gofer	getpid	sentry
	tgkill	sentry	write	sentry
	read	sentry	fsync	sentry

Table 4 Differences in `read(fd, buf, count)` system call mapping according to previous `open` system call's parameter. Flags followed by an |(or) sign in {}(braces) mean one of the flags.

Flags	Syscall	Process
O_RDWR	–	
O_RDONLY, O_RDWR| {O_SYNC| O_DSYNC|O_DIRECT}	pread64	sentry

system call as shown in Table 3 (i.e., it may not be necessary to go through the `gofer`). In `gVisor`, write-related system calls (e.g., `write`, `writev`, `pwritev`, `pwritev2`, `pwrite64`) are all implemented in `pwrite64`, and the written byte of the `write` system call invoked in `gVisor` can be known as the return value of `pwrite64`. But the write operation on the file opened with the `O_SYNC`, `O_DSYNC`, and `O_DIRECT` flags forces significantly more system calls and requires communication with the `gofer`. Although `write` is a system call without the flag parameter, the cost of the write operation may vary in `gVisor`. The `write` call in `gVisor` is affected by the state flag of the previous `open` system call.

4.2.3 `read` System Call

As shown in Table 4, the `read` system call is similar to the `write`, but it is a lighter operation. The `read` system call can also be handled by the `sentry` itself, and it is the only system call that does not require communication with `gofer` among the file-related system calls we have tested. Similar to `write`, the `sentry` calls `pread64` only when it is a read operation for a file opened with `O_SYNC`, `O_DSYNC`, and `O_DIRECT` flags. Otherwise, there is no additional host system call invoked in the read operation. Also, in `gVisor`, all read-related system calls (e.g., `read`, `readv`, `preadv`, `preadv2`, `pread64`) are implemented as `pread64`, and the return value indicates the number of bytes read.

4.3 Analysis of gVisor Performance

According to the mapping results, on average, `open (O_CREAT)` is 13× more expensive and `renameat` is 8.5× more expensive than native. There is no flag parameter for `write` and `read`, and each operation is handled by `pwrite64` and the `sentry` itself in `gVisor`. When the target file is opened with the `O_SYNC` flag, however, their mappings are different even for the same system call invocations, and

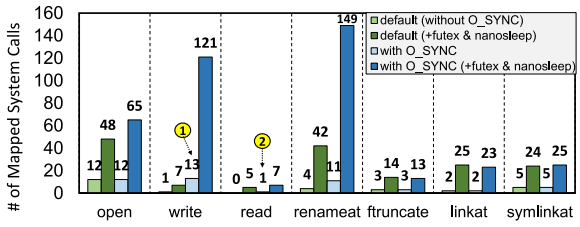

Fig. 4 Effect of `O_SYNC` flag to system call translation.

the operation becomes more expensive. This is a result of `gVisor`'s performance and security trade-off, and in file operations, it is because of the maintaining of the internal deprivileged file system in `gVisor` and the interaction between `gofer` and the host kernel for the use of the host file system. In addition, in `gVisor ptrace` mode, 17 system calls (10 `ptrace`, 2 `wait4`, 4 `futex`, 1 `newfstat` as shown in Fig. 2) are additionally invoked while intercepting system calls from the application. Although excluded from the mapping results, this overhead is non-negligible from the performance perspective. It also varies greatly depending on the parameters of the system call, and in particular, the operation on the file created with the `O_SYNC` flag resulted in more expensive overhead.

Figure 4 shows the difference in the number of system calls for files opened with and without the `O_SYNC` flag. For each, we also show two cases: *i)* with and *ii)* without counting `futex` and `nanosleep` syscalls. These two syscalls are treated separately since they commonly appear in all syscall mapping traces without being part of the core logic of the syscall and are assumed to be part of the thread management by `gVisor`. For `open`, the presence of `O_SYNC` flag does not affect `open` itself, but affects subsequently invoked system calls. The results of `write` and `read` system calls differ in `gVisor` depending on the `O_SYNC` flag of the opened file. The `write` system call invokes 13× more system calls (at ①), and `read` system call is changed from none to 1 (by `pread64` at ②). Taking into account the `futex` and `nanosleep`, the number of syscalls can reach as high as 17.3× (= 121/7) and 3.5× (= 149/42) for `write` and `read`, respectively. Compared to the native setting, `open` generates 48 syscalls (no O_SYNC) and about 6 syscalls for read and write.

5. Conclusion

In this letter, we performed an in-depth analysis of gVisor's file operations in terms of system call parameters. We built sysmap tool, which could discover the syscall mappings before and after they passes through the proxy layer of secure containers. We found that gVisor's file system had a significant volume of interaction with the host kernel. We believe our findings will provide insight into the measurement of the attack surface of secure container runtimes and facilitate future measurement studies of secure container runtimes.

References

[1] "gVisor," https://gvisor.dev/, 2023. accessed Feb. 18. 2023.
[2] "Kata Containers," https://katacontainers.io/. accessed Feb. 18. 2023.
[3] "Sysbox," https://www.nestybox.com/sysbox/. accessed Feb. 18. 2023.
[4] Z. Shen, Z. Sun, G.E. Sela, E. Bagdasaryan, C. Delimitrou, R. Van Renesse, and H. Weatherspoon, "X-containers: Breaking down barriers to improve performance and isolation of cloud-native containers," The 24th ASPLOS 2019, pp.121–135, 2019.
[5] H. Tazaki, A. Moroo, Y. Kuga, and R. Nakamura, "How to design a library os for practical containers?," Proceedings of the 17th ACM VEE, pp.15–28, 2021.
[6] S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. O'keeffe, M.L. Stillwell, et al., "Scone: Secure linux containers with intel sgx," 12th USENIX OSDI 16, pp.689–703, 2016.
[7] Z. Li, J. Cheng, Q. Chen, E. Guan, Z. Bian, Y. Tao, B. Zha, Q. Wang, W. Han, and M. Guo, "RunD: A lightweight secure container runtime for high-density deployment and high-concurrency startup in serverless computing," 2022 USENIX Annual Technical Conference, Carlsbad, CA, pp.53–68, July 2022.
[8] E.G. Young, P. Zhu, T. Caraza-Harter, A.C. Arpaci-Dusseau, and R.H. Arpaci-Dusseau, "The true cost of containing: A gvisor case study," 11th USENIX HotCloud 2019, 2019.
[9] "Containers Matrix," https://attack.mitre.org/matrices/enterprise/containers/, accessed Feb. 18. 2023.
[10] S. Forrest, S. Hofmeyr, and A. Somayaji, "The evolution of system-call monitoring," 2008 annual computer security applications conference (acsac), pp.418–430, IEEE, 2008.
[11] S. Forrest, S.A. Hofmeyr, A. Somayaji, and T.A. Longstaff, "A sense of self for unix processes," Proceedings 1996 IEEE Symposium on Security and Privacy, pp.120–128, IEEE, 1996.
[12] L. Lu, A.C. Arpaci-Dusseau, R.H. Arpaci-Dusseau, and S. Lu, "A study of linux file system evolution," 11th USENIX Conference on File and Storage Technologies (FAST 13), pp.31–44, 2013.
[13] X. Lin, L. Lei, Y. Wang, J. Jing, K. Sun, and Q. Zhou, "A measurement study on linux container security: Attacks and countermeasures," Proceedings of the 34th ACSAC, pp.418–429, 2018.
[14] S. Ghavamnia, T. Palit, A. Benameur, and M. Polychronakis, "Confine: Automated system call policy generation for container attack surface reduction," 23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID), Oct. 2020.