SecQuant: Quantifying Container System Call Exposure

Sunwoo Jang[1†], Somin Song[1†], Byungchul Tak[1⋆](✉), Sahil Suneja[2], Michael V. Le[2], Chuan Yue[3] and Dan Williams[4]

[1] Kyungpook National University, Daegu, Republic of Korea
{swjang,sominsong,bctak}@knu.ac.kr
[2] IBM TJ Watson Research Center, Yorktown Heights, NY, USA
{suneja,mvle}@us.ibm.com
[3] Colorado School of Mines, Golden, CO, USA
chuanyue@mines.edu
[4] Virginia Tech, Blacksburg, VA, USA
djwillia@vt.edu

Abstract. Despite their maturity and popularity, security remains a critical concern in container adoption. To address this concern, secure container runtimes have emerged, offering superior guest isolation, as well as host protection, via system call *policing* through the surrogate kernel layer. Whether or not an adversary can bypass this protection depends on the effectiveness of the system call policy being enforced by the container runtime. In this work, we propose a novel method to quantify this container system call exposure. Our technique combines the analysis of a large number of exploit codes with comprehensive experiments designed to uncover the syscall pass-through behaviors of container runtimes. Our exploit code analysis uses information retrieval techniques to rank system calls by their risk weights. Our study shows that secure container runtimes are about 4.2 to 7.5 times more secure than others, using our novel quantification metric. We additionally uncover changing security trends across a 4.5 year version history of the container runtimes.

Keywords: Secure container runtime · Security quantification · System call · Container escape · Exploit code analysis.

1 Introduction

Container technology has firmly established itself as an essential component of modern cloud platforms. All major cloud providers offer various kinds of container-based services either in the form of directly usable container instances, orchestrated container environment services, or as an underlying layer for serverless computing engines [5, 16, 19, 31]. However, the foremost concern of adopting containers in production firmly remains security [26]. Examples of some infamous vulnerabilities affecting containers include Dirty COW (CVE-2016-5195),

[⋆] Corresponding author, † The 1st and 2nd authors contributed equally.

RunC Container Escape (CVE-2019-5736), and Kubernetes container escape via eBPF (CVE-2021-31440).

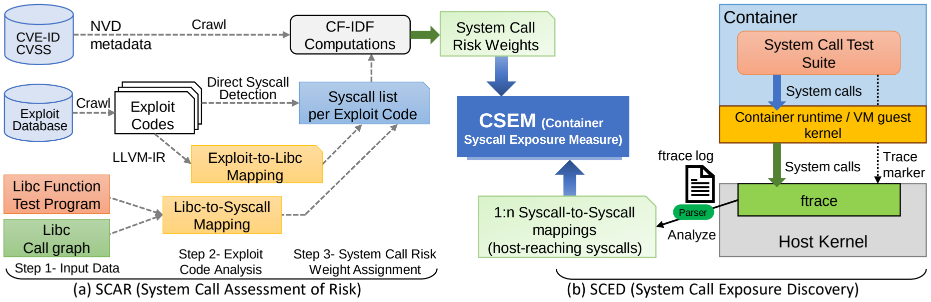
To address the security concerns, there have been recent efforts to design secure container runtimes such as gVisor [17], Kata [20], Nabla [32], Firecracker [4, 13], and Unikernels [21, 22, 29], amongst others. These focus on limiting a containerized application's system call access to the host kernel, to minimize the possibility of an exploit. The main principle is the employment of a 'surrogate or proxy kernel'. This kernel can take the form of a user-space kernel, a library OS, or a light-weight virtual machine (VM) guest kernel, which lies in front of the actual host kernel being protected, effectively sandboxing the containerized applications and preventing them from interfacing directly with the host. The role of this surrogate kernel is to handle most of the system calls directly and allow only a reduced subset of system calls to reach the host kernel.

Most claims about the security guarantees offered by the secure container runtime techniques are qualitative at best [11, 26, 36, 39]. What is missing is a metric that reflects the quantitative measures of their 'secureness', which can be a useful tool as container technologies continue to evolve. First, it can be used to track and guide the security hardening processes across the development iterations of the container runtimes [25]. Second, it enables a direct comparison of the security strengths of different, and in some cases competing, secure container runtime alternatives. Lastly, a quantification methodology that reflects the up-to-date state of vulnerabilities enables observing score changes due to external factors such as time-varying trends of threats.

Our goal in this work is to measure how well the secure container runtimes fair at policing or filtering the application-invoked system calls. One approach for a security metric could be to use the specification of the secure container runtimes, in terms of the set of system calls allowed to reach the host. However, the number of system calls in the set alone is not an accurate and sufficient metric of security. This is because the usefulness or importance of system calls in exploits differs, and changes over time. Also, it is unclear how to correctly compare sets not in a proper subset relationship with each other. So the challenge exists in translating the presence or absence of system calls into a numeric score.

In this paper, we present a novel approach for quantifying the system call exposure of container runtimes. Our technique, called SecQuant, consists of two parts: SCAR(*System Call Assessment of Risk*) and SCED(*System Call Exposure Discovery*). In SCAR, SecQuant produces numerical risk weights of system calls by analyzing existing exploit codes, and applying a variant of the information retrieval technique—TF-IDF—customized and extended for the security quantification task. In SCED, SecQuant performs comprehensive system call tests inside the target containers using a purpose-built test suite. It generates one-to-many mappings between container-invoked system calls and the ones appearing on the host in response. The outputs from SCAR and SCED are combined to produce system call exposure scores for the different container runtimes.

Analyzing the container runtimes with our quantification methodology reveals several interesting findings. First, secure container runtimes offer 4.2 to 7.5



(a) SCAR (System Call Assessment of Risk)

(b) SCED (System Call Exposure Discovery)

Fig. 1: SecQuant architecture for quantifying container system call exposure

times improved (i.e., reduced) system call exposure than their general-purpose counterparts. Second, in addition to a non-negligible number of *pass-through* system calls reaching the host kernel, the secure container runtimes also generate a significant number of *derived* system calls, appearing consistently in response to the ones invoked by the application. This presents an indirect avenue of exposure which we account for in our metric. Third, their exposure scores change across a 4.5 year version history, showing both increasing and decreasing trends across different runtimes.

2 Secure Containers and Threat Model

In the overall container security landscape, we limit the scope of this work to the specific case of system call access restriction to the host. Our analysis targets primarily the host kernel vulnerabilities, which an adversary inside a container wishes to exploit via possibly crafted system calls. We approach the security problem from the host perspective, in terms of the reachability of critical system calls. For example, if the container runtime uses light-weight virtualization instead of containerization, then the system call pass-through policy will get accounted towards the container runtime's exposure score.

We assume an untrustworthy guest container, either instantiated directly by an adversary, or compromised to gain control by them. The goal of the adversary is then to attack the host, or other colocated guests, from within this guest container, for malicious gains such as privilege escalation, denial of service, data corruption, information leak, and service theft, amongst others [18].

We set the scope of our analysis to exploits which leverage vulnerabilities exposed by system calls. For example, the exploits targeting the Dirty COW vulnerability (CVE-2016-5195) abuse a race condition in the kernel by rapidly calling `madvise` system call and `write` system call on `/proc/self/mem` for privilege escalation. One way in which secure container runtimes protect against such attacks is by policing system call invocations made from the guest. This is typically achieved by employing a *surrogate* kernel to handle a majority of the system calls and allowing only a few necessary ones to reach the host.

However, this does not necessarily imply that the system calls that do reach the host from the applications are passed down through the surrogate kernel to the host kernel as-is. The host-reaching system calls may be a 'translated' set of system calls assembled by the surrogate kernel in response to the system calls the applications invoke. During this process, the application-level system calls may be extended with additional ones, replaced by compatible ones, sanitized, or queued up for batching. For example, gVisor implements a portion of the Linux system functionality in an application kernel written in Go. Kata containers is essentially a light-weight VM with its own guest kernel. And Nabla container wraps an application with a userspace library OS, which implements almost all of the system call functionality tailored to the application.

Depending upon the system call policy enforced by a secure container runtime, an adversarial guest may succeed in its attack. For example, Kata containers have been shown to be vulnerable to the buffer overflow exploits [35]. In this work, we quantify this system call exposure of the different secure container runtimes. We trust the host system's integrity and assume that it is not already compromised. We base our measurements on publicly available exploit codes, assuming they are representative of real-world attacks.

3 Design of SecQuant

Figure 1 shows the overall architecture of our SecQuant approach to quantify system call (or *syscall*, used interchangeably) exposure of container runtimes. It consists of two independent components: (i) SCAR: System Call Assessment of Risk, and (ii) SCED: System Call Exposure Discovery. SCAR is container-independent and determines the *risk weights* associated with system calls by analyzing existing exploit codes. These per-syscall risk weights are generated by employing static code analysis as well as dynamic experiments, in conjunction with information retrieval techniques. SecQuant's second component, SCED, examines the system call reachability and host-kernel *pass-through* behavior for a given container runtime. By using a systematic exploration of syscall-level access via a custom test suite, it emits a 1:n mapping between a syscall made at the application-level to the syscalls reached the host-level. Finally, the output of the two components, per-syscall risk weights and syscalls-per-container, are combined together to produce an overall Container System call Exposure Measure (CSEM) for the container runtime. This CSEM metric essentially quantifies how well the container runtime fairs at policing the application-invoked system calls. We use it to (i) compare the security posture across different container runtime alternatives, and (ii) analyze evolution across different versions (Section 5).

3.1 SCAR: System Call Assessment of Risk

The goal of SecQuant's SCAR component is to determine a *risk weight* associated with each syscall. The intuition is to convert the occurrence frequencies of different system calls in existing exploit codes, into a numerical measure encapsulating the importance of each syscall in attacking the host. Figure 1(a) shows the steps involved in extracting syscall risk weights, described as follows.

Step 1- Input Data: SecQuant ingests three different types of input data:

1. **Exploit codes:** From ExploitDB [1], Project-Zero [2], and a few other standalone git repositories, we collected 298 exploit codes written in C and targeting the Linux kernel.
2. **Exploit Metadata:** The exploit codes extracted from the aforementioned vulnerability databases are augmented with metadata such as their CVE-ID, release date, and CVSS scores from NVD, wherever applicable.
3. **Library function Call Graph:** To facilitate the extraction of syscalls from the library functions used in an exploit, we use the library function call graphs from CONFINE [14] and refine it.

Step 2- Exploit Code Analysis: The objective of this step is to build a complete set of system calls being used by each exploit code. While it is possible to put efforts into directly running the exploits to uncover the syscalls used [26], such an approach can be challenging to properly set up and difficult to automate. Hence, we rely mostly on a static analysis approach. Since high-level code rarely invokes system calls directly, relying instead on library functions, we need to discover syscalls being invoked indirectly by these library calls. To collect the syscall set for each exploit code, we first identify the library functions it uses, and then use a mapping between the library functions and the corresponding syscalls being employed underneath. The specific details are described next.

1. Exploit-to-Libc Mapping: Instead of directly parsing the C syntax of the exploit codes, we use the LLVM Intermediate Representation (IR) to identify the library functions in the codes. LLVM IR uses the prefix character '@' for library functions, making it easier to recognize them.

2. Libc-to-Syscall Mapping: The next step is to identify the reachable system calls for different library functions extracted from the exploit codes. We use the state-of-the-art for such libc-to-syscall mapping, CONFINE, which uses static code analysis to build a function call graph from the libc source code, terminating with system calls at the leaves. The challenge, however, is the accuracy of the call graph, especially in the case of function pointers, which results in either missing system calls or reporting a bloated set as reachable. We found certain errors in CONFINE's libc-to-syscall mapping, in both over and under estimating the reachable system calls, especially when the mapping is not 1:1. In such cases, we use a custom test suite to call the corresponding library functions and collect the system calls reaching the host kernel using ftrace. Using such refinement over CONFINE, we are able to map almost 90% of the libc functions to at-most two system calls each. We limit our scope to only the libc functions and their corresponding arguments appearing in our exploit codes, and not the entire 2000+ APIs. Linking the exploit codes with another C library such as musl would possibly change the system call mappings. We leave this comparison for the future exploration. In addition to the libc, we also extract system calls from functions of other libraries used in the exploits, e.g. `libfuse`, `libecryptfs`, `libpcap`, `libaio`, etc.

3. Direct Syscall Detection: In some exploit codes, we observe direct invocations of system calls through `INT` or `SYSCALL` instructions, which bypass the libc library. Such carefully crafted system call invocations are likely to be key to the intended attack in the exploit codes. We directly parse such inline assembly codes to identify the syscalls being employed.

Step 3- System Call Risk Weight Assignment: After extracting the syscalls being used by the exploit codes, the next step is to assign a *risk weight* to each syscall based on its importance to the exploits. One approach is to use the proportion of exploit codes a syscall appears in, as an indicator of its associated risk. However, this may not appropriately capture its degree of risk. A syscall may appear in many exploit codes simply because it is part of a commonly used initialization procedure. On the other hand, it may appear in only a few exploit codes, but be critical to successfully exploit the corresponding vulnerabilities.

To address this challenge, we adopt a popular technique, TF-IDF, used in the Information Retrieval (IR) domain to compute the relevance of words to a particular topic across documents. TF, the Term Frequency, captures the idea that the more frequently appearing terms are likely more important to the content of the document. IDF, the Inverse Document Frequency, penalizes the term weights if they appear in many documents since it implies that such words are common and carry less relevance to any specific topic. Adopting TF-IDF to our security context, the *term* is mapped to a *system call*, while *document* is mapped to the *exploit code*. However, we observe that such naïve translation is insufficient. While the rationale of IDF holds true in our setting, the rationale of TF may not necessarily hold. Specifically, adhering to the IDF rationale, a syscall widely used across many exploit codes is less likely to be a syscall playing a key role in the attack logic. The observation that the most used system calls in exploit code are `close`, `brk`, `exit` and `nanosleep` also supports this. However, in contrast to the TF rationale, just because a syscall appears frequently within an exploit code (i.e., used repeatedly in a single type of attack), it does not necessarily mean that it is more important to the attack than less frequent ones.

Thus, we introduce another component—*Class Frequency* (CF)—to replace the role of TF. We define CF as the proportion of exploit codes a particular syscall appears in within a *class*. A *class* is defined as a subset of exploit codes grouped by common characteristics, such as the vulnerability being leveraged, or the attack methodology, amongst others. The intuition is that if a syscall consistently appears across the exploit codes within the same *class*, then it presumably plays a key role to the attack logic for that class. Note the contrast to IDF which considers a syscall with more appearances across the entire exploit codes as less relevant. If a syscall is indeed crucial to a specific class of exploits, it will render the CF value high. At the same time, this value will be countered by IDF if the syscall appears frequently throughout other exploit codes as well. The information we currently use to realize the *class* concept is the CVE-ID [12] associated with each exploit code.

Metric Formulation: Our metric to calculate system call risk weights is denoted as **CF-IDF**, obtained by combining the two components. Formally, let s, e and E denote a system call, an exploit code and the set of all exploit codes, respectively. We use C_e to indicate the class, subset of E, of which e is a member. The smallest size of C_e is 1. That is, an exploit code can be the sole member of a class. Also, the set of syscalls used in a specific exploit code e is denoted as σ_e.

The IDF component is represented as a vector, computed using the generic IR formulation as:

$$idf(s, E) = log_{|E|} \left(\frac{|E|}{|\{e|s \in \sigma_e\}| + 1} \right) \tag{1}$$

Note that we use the size of E as the base of log in order to keep the IDF value within 0 and 1 as well. The term $\{e|s \in \sigma_e\}$ is the set of exploit codes within which s exists. Equation 1 gives us a lower value as the system call appears in a larger number of exploit codes.

The CF component is represented as the fraction of exploit codes which contain the syscall s amongst all exploit codes belonging to the class C_e. It can be viewed as a DF (Document Frequency) metric for each class. It also contains the normalized CVSS score (to [0,1] range) to reflect the vulnerability severity of the CVE that the exploit code belongs to. Formally, CF is represented as:

$$cf(s, C_e) = \frac{1}{10} CVSS_e \times \frac{|\{e|s \in \sigma_e, e \in C_e, C_e \subseteq E\}|}{|C_e| + 1} \tag{2}$$

The overall CF-IDF value (V) is computed by multiplying both components, and represented as 2D data consisting of syscalls and exploit codes, as:

$$V(s, e) = cf(s, C_e) \times idf(s, E) \tag{3}$$

Finally, a per-syscall risk weight (W) is calculated by averaging its CF-IDF values across all exploit codes, as:

$$W_s = \frac{\sum_e V(s, e)}{|\{e|s \in \sigma_e\}|} \tag{4}$$

3.2 SCED: System Call Exposure Discovery

While SecQuant's SCAR component computes risk weights for different system calls, SCED determines which of these system calls are *accessible* under the different container runtimes (Figure 1(b)). This information is gathered by executing test programs within a given container runtime, and observing which syscalls reach the host-kernel. We developed our own test programs instead of using existing tools such as LTP [28] or Syzkaller [37]. Their focus is somewhat different (e.g. stress testing), which made the cost of extending them significantly greater than rewriting. Their heavy use of library calls, instead of direct syscall invocations, made it difficult to add necessary modifications. We thus created a custom test suite limited to the 185 syscalls found in the exploit codes we collected.

System Call Tracing: Depending upon how a container runtime handles the application-invoked system calls, different behavior can be observed on the host:

- No system calls whatsoever seen at the host kernel
- Identical system call arriving at the host kernel
- Syscalls arriving at the host kernel including the application-invoked syscall
- Syscalls arriving at the host kernel without the application-invoked syscall

We use the ftrace mechanism to detect these cases, while running our test suite inside the container(s) to exercise all target system calls. We tightly enclose all syscall statements inside the test programs with the ftrace's *trace-marker* write actions, to accurately pinpoint the begin and end markers in the ftrace logs. Then, the host-reaching syscall set is identified by locating the `sys_enter` events within these markers. We also set the `event-fork` option to include all the child processes that may be spawned along the way.

Even with the use of the *trace-marker* to narrow down the exact time range, the ftrace logs can contain syscalls from other parallel threads, which can incorrectly inflate the derived syscall set. We use domain knowledge specific to the container runtimes to identify the relevant threads, and filter the logs accordingly to collect only the syscall events triggered by the original syscall of the test program.

The *trace-marker* approach works for most container runtimes we experiment with, except Kata, sysbox, and LXC. In case of Kata, mounting the `tracefs` filesystem inside the container (VM, technically) is problematic. As a marker for Kata, we instead use the `fsync` syscall (repeated fixed number of times as a signature), which we found to be a stable and immediately responsive pass-through syscall. `getpgid` syscall similarly serves as a marker for sysbox and LXC.

Traced System Call Types: We further categorize the system calls that reach the host kernel into the following groups:

- *Pass-through*: These include the application-level system calls which eventually end up reaching the host kernel, either as-is or after any argument sanitization. We additionally include a notion of *equivalent* system calls for this category—system calls with different syscall numbers but sharing the same kernel function for execution (Appendix Table C.1). If a syscall results in triggering such equivalent syscalls, it is considered a pass-through.
- *Derived*: It refers to the system calls generated by the container runtime (and reaching the host kernel) in response to an application-invoked system call, excluding the pass-through if any. These can be differentiated into:
  - *Workload-dependent*: The set of system calls necessary to carry out an application-invoked system call. One application-invoked syscall may be translated, replaced or converted into a sequence of multiple syscalls.

Table 1: Example host-reaching syscall data for `open` as an output of SCED

runsc-ptrace		
Observed syscalls	**Count**	**Caller process**
futex	20	gofer, sandbox
ptrace	10	sandbox
openat	4	gofer
newfstat	4	gofer
wait4	2	sandbox
fcntl	2	gofer
sendmsg	1	gofer
recvmsg	1	sandbox
dup	1	gofer
close	1	gofer

kata-qemu		
Observed syscalls	**Count**	**Caller process**
futex	6	pool, virtiofsd
write	3	pool
read	3	virtiofsd
ppoll	3	virtiofsd
openat	3	pool
setresuid	2	pool
setresgid	2	pool
newfstatat	1	pool
geteuid	1	pool
getegid	1	pool

- *Architecture-dependent*: The set of system calls that are consistently generated and reached the host kernel for every application-invoked syscall, owing to the container runtime's architectural characteristics. It may be, for example, due to the specific syscall interception mechanism used, or from the auxiliary components for the container management.

SCED results in host-reachability data for each system call for a given container runtime. Table 1 shows an example for the `open` system call for two container runtimes—`gVisor` and `Kata`. The output contains a list of observed syscalls with the occurrence count and the originating process name. As can be seen, calling `open` inside the containers results in a large number of derived system calls, together with its pass-through equivalent—`openat`.

3.3 Container Syscall Exposure Measure

The complementary information gathered and computed by SecQuant's SCAR and SCED components, in terms of risk-weights-per-syscall and syscalls-per-container respectively, are finally combined together to generate the overall container system call exposure measure—CSEM. Formally, let S be the complete list of known system calls. Computing CSEM requires three inputs: (i) from SCAR: the per-syscall risk weight vector $W = \{W_i | i \in S\}$, (ii) from SCED: the entire list of observed system call sets, D, in which D_s refers to a single set of system calls observed to be induced from a system call s, and (iii) a reduction ratio r ($0 \leq r \leq 1$), controlling the risk-weight contribution of the derived system calls. Pass-through system calls are given the whole risk weight values from W_s if s exists in D_s. Derived system calls are applied the reduction ratio and averaged over the set size of D_s. Finally, CSEM is defined as:

$$CSEM(W, D, r) = \sum_{s} \sum_{d \in D_s} \Big(W_s \cdot I(s,d) + \frac{W_d \cdot r}{|D_s|}(1 - I(s,d)) \Big) \tag{5}$$

where I is an indicator function selecting the correct terms while going through syscalls in D_s, testing if two given syscalls are identical or not, as:

$$I(s,d) = \begin{cases} 1 \;\; if\; s = d \\ 0 \; otherwise \end{cases} \tag{6}$$

Table 2: CF-IDF correctly capturing similarity within exploit code groups. Similarity scores are against the first exploit code in each group.

Group	Grouping Criteria	Exploit-ID (source)	CVE-ID	Similarity
G-I	Same CVE. Keyring object reference mishandling with crafted keyctl. PrivEsc, DoS.	40003 (exploit-db)	CVE-2016-0728	-
		39277 (exploit-db)		1.0
		2016-0728A (git)		1.0
		2016-0728B (git)		1.0
G-II	Same CVE. Incorrect integer data type via crafted perf_even_open. PrivEsc.	33589 (exploit-db)	CVE-2013-2094	-
		26131 (exploit-db)		1.0
		25444 (exploit-db)		1.0
G-III	Different CVE but same vulnerability type. Improper traversal via crafted epoll_create and epoll_ctl. DoS.	35403 (exploit-db)	CVE-2011-1083	-
		35404 (exploit-db)	CVE-2011-1082	0.764533
G-IV	Different CVE but same vulnerability type. Lack of validation with bpf. PrivEsc.	2021-31440A (git)	CVE-2021-31440	-
		2020-8835A (git)	CVE-2020-8835	1.0
		2020-8835B (git)		1.0
		2021-3490A (git)	CVE-2021-3490	0.975811

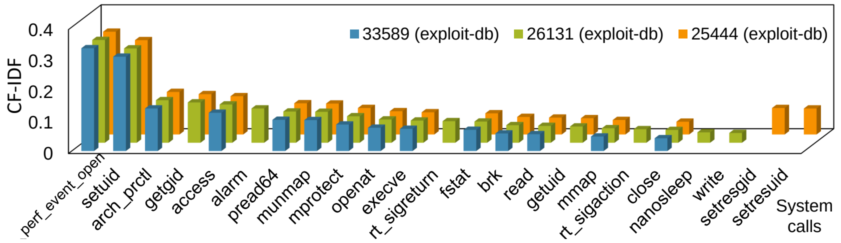


Fig. 2: Comparison of risk weight vectors of exploits in Group II showing visually very similar signatures for all three group members (Table 2). The key system call, `perf_event_open` is identified correctly to have the highest weight.

4 System Call Analysis Results

4.1 Verification of CF-IDF Metric

We first provide empirical justification supporting the soundness of the CF-IDF metric we use to assign risk weights to system calls. The CF-IDF scores in its 2D form, which is the collection of vectors with the length equal to the number of syscalls, can be considered as signatures that represent the characteristic of exploit codes. If our assumptions and intuitions behind CF-IDF are valid, it would generate similar signatures (i.e., vector of risk weights) for the exploit codes that are *truly similar* in their nature. We analyzed all exploits using domain knowledge and classified them as similar groups if the vulnerability was located in the same kernel component or the method for triggering the vulnerability was similar. We present such a similarity comparison in Table 2. It shows four example groups each containing similar kinds of exploit codes. Column 5 shows the similarity scores generated using cosine similarity between our risk weight vectors for the different exploit codes. As can be seen, the generated similarity scores are high within each group. This can also be seen in Figure 2's vector distribution, which is very similar for all exploits within a group (and different across groups; shown in Appendix Figure A.1). This corroborates the effectiveness of our CF-IDF metric in being able to capture the unique characteristics of different exploit (groups) in terms of system call composition and risk weights.

Table 3: Partial (top-70) ranked list of system calls by the risk weights obtained from the SCAR process. Full list given in Appendix Table A.1.

Rank	Syscall	Weight	Rank	Syscall	Weight	Rank	Syscall	Weight	Rank	Syscall	Weight
1	capset	0.439551	19	futimesat	0.30329	37	shmget	0.244536	55	pwrite64	0.21538
2	add_key	0.409431	19	inotify_rm_watch	0.30329	37	shmat	0.244536	55	set_mempolicy	0.21538
3	recvmmsg	0.392371	19	inotify_init1	0.30329	37	sigaltstack	0.244536	55	readv	0.21538
4	getresuid	0.388023	19	restart_syscall	0.30329	37	setxattr	0.244536	55	sched_getaffinity	0.21538
4	sendfile	0.388023	19	utimensat	0.30329	41	symlink	0.244057	55	shmdt	0.21538
4	io_uring_reg	0.388023	24	clock_nanosleep	0.303143	42	getcwd	0.244007	60	mremap	0.212287
7	shutdown	0.335366	25	umount2	0.295618	43	fchmod	0.240128	61	inotify_init	0.210391
8	settimeofday	0.334059	26	chown	0.286681	44	modify_ldt	0.237571	62	sched_yield	0.206736
9	rename	0.329819	27	link	0.284955	44	clock_gettime	0.237571	63	recvmsg	0.205074
10	creat	0.329663	28	dup3	0.272522	46	process_vm_readv	0.237358	64	getegid	0.204785
11	keyctl	0.32028	29	eventfd2	0.269163	47	writev	0.235578	65	fallocate	0.202194
12	fchown	0.316477	30	msgsnd	0.267191	48	getdents	0.234431	65	_sysctl	0.202194
12	flock	0.316477	31	sched_setscheduler	0.264745	49	sendmmsg	0.232625	65	move_pages	0.202194
12	mknod	0.316477	32	inotify_add_watch	0.261581	50	syslog	0.232287	68	shmctl	0.200075
12	mq_notify	0.316477	32	waitid	0.261581	51	mount	0.226888	69	msgctl	0.199029
12	io_setup	0.316477	34	msgget	0.254518	52	rmdir	0.224417	70	dup	0.197707
12	io_submit	0.316477	35	pipe2	0.25433	53	getgroups	0.219776	...	...	...
12	kcmp	0.316477	36	chmod	0.248783	54	select	0.216088	185	close	0.026151

4.2 System Call Risk Weights

Table 3 shows the output of SecQuant's SCAR component, in terms of a ranking of syscalls by their risk weights computed using the CF-IDF metric (partial list shown due to space constraints; full list in Appendix Table A.1). As per SCAR's analysis, the `capset` system call gets assigned the highest weight, while `close` gets ranked the last. The high weight assignment to `capset` is owing to the exceptionally high CVSS score (10/10) of the only CVE that it belongs to—CVE-2000-0506. Appendix B uses concrete examples to show how the various interdependent factors—CVSS scores, number of exploits, and CVE class size—affect the system call risk weights.

Note that the last syscall in the list should not be interpreted as a *harmless* syscall because, although ranked the last, it is given a rank because it played a part in composing the attack logic in some of the exploit codes. The fact that it is on this list already implies a substantial degree of utility to the attacks. Similarly, it is not always the case that a syscall with a higher rank is always riskier than nearby syscalls with slightly lower risk weights. This is because the final risk weight of a syscall is an average of values of a vector across exploit codes as expressed in Equation 4.

4.3 Pass-through System Calls across Containers

Table 4 shows the output of SecQuant's SCED component, in terms of the syscall handling behavior of different container runtimes. As expected, the trace results of general-purpose container runtimes (`runc, crun, sysbox,` and `lxc`) contain mostly pass-through syscalls and a small number of derived syscalls. On the contrary, Kata containers and gVisor generate a large number of derived syscalls, passing only about 20% of application-side syscalls through the runtime layer.

Table 5 shows a list of *pass-through* syscalls we captured from gVisor and Kata containers runtimes. The gVisor and Kata containers runtimes pass through different set of system calls, the risk weights of which eventually impact the

Table 4: Number of pass-through/non-pass-through system calls

Container runtime	# of Pass-through syscalls	# of Non-pass-through syscalls	# of Tested syscalls
runc	176 (93.6%)	12 (6.4%)	188
runsc-ptrace	37 (19.5%)	153 (80.5%)	190
runsc-kvm	35 (18.6%)	153 (81.4%)	188
kata-qemu	43 (22.5%)	148 (77.5%)	191
kata-clh	44 (22.2%)	154 (77.8%)	198
crun	181 (95.8%)	8 (4.2%)	189
sysbox	181 (91.9%)	16 (8.1%)	197
lxc	187 (94.9%)	10 (5.1%)	197

Table 5: Pass-through system calls for `runsc` and `kata` runtimes. Equivalent system calls are shown as X→Y. SCAR rank is given in parenthesis.

runsc-ptrace (weight rank)	kata-qemu (weight rank)
utimensat(23), futimesat(19) → utimensat(23), fchmod(43), chmod(36) → fchmod(43), pwrite64(55), write(173) → pwrite64(55), getdents64(76), getdents(48) → getdents64(76), ptrace(86), tgkill(103), ftruncate(133), wait4(157), pread64(171), munmap(172), nanosleep(174), openat(179), open(175) → openat(179), creat(10) → openat(179), fstatfs, statfs(90) → fstatfs, fstat(180) → fstatfs, renameat, rename(9) → renameat, mkdirat, mkdir(75) → mkdirat, linkat, link(27) → linkat, readlinkat, readlink(107) → readlinkat, fchownat, fchown(12) → fchownat, chown(26) → fchownat, symlinkat, symlink(41) → symlinkat, unlinkat, lchown, fsync	utimensat(23), futimesat(19) → utimensat(23), fallocate(65), getdents64(76), getdents(48) → getdents64(76), futex(89), ftruncate(133), connect(150), renameat2(166), write(173), openat(179), open(175) → openat(179), creat(10) → openat(179), close(185), pwritev2, pwritev, writev(47) → pwritev, fchmodat, chmod(36) → fchmodat, fchmod(43) → fchmodat, fstatfs, statfs(90) → fstatfs, mknodat, mknod(14) → mknodat, renameat, rename(9) → renameat, mkdirat, mkdir(75) → mkdirat, linkat, link(27) → linkat, readlinkat, readlink(107) → readlinkat, fchownat, fchown(12) → fchownat, chown(26) → fchownat, symlinkat, symlink(41) → symlinkat, unlinkat, unlink(128) → unlinkat, lchown, fsync

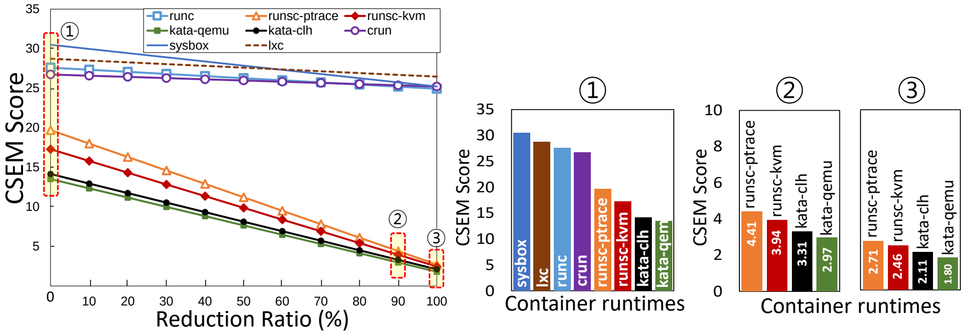
overall exposure measure for the container runtimes (Section 5.1). In determining the *pass-through* system calls, we use the concept of system call *equivalence*. If a container invokes a system call X (e.g., open), and we observe a system call Y (e.g., openat), they are not a *pass-through* in a strict sense. However, if X and Y share the code at the function level in the kernel, we view them as identical and, thus, *pass-through*. Arrows in Table 5 and 6 indicate these relationships. Note the ranking order of the lists with some syscalls out of place, e.g., `getdents` and `creat`. These syscalls get assigned the weights of their *equivalent* syscall counterparts (Section 3.2), which they are always replaced with by the container runtime.

A large part of pass-through syscalls are file I/O related since the data has to physically travel in and out of the container runtime. This holds true even for the Kata containers which is a light-weight VM with its own guest kernel.

5 Container Runtime Security Analysis

In this section, we analyze the security posture of container runtimes using our CSEM metric. We use LXC, runc, crun, and sysbox as baseline container runtimes, and gVisor and Kata containers as secure alternatives. The gVisor `runsc` uses a userspace surrogate kernel, intercepting application-level syscalls using either ptrace (`runsc-ptrace`) or KVM (`runsc-kvm`). Kata uses a light-weight VM to run the user application, relying on either QEMU (`kata-qemu`), Firecracker [13][5] or Cloud Hypervisor (`kata-clh`) [10] as VM hypervisors. Host and container runtime version information can be found in Appendix C. We do not include unikernel-inspired container runtimes such as Nabla [32] in our study, since they require varying degrees of effort to make general applications (including our test suite) run on them—a by-product of their generality vs. security trade-off.

[5] Firecracker unsupported in Kata 2.x as of conducting this study.



(a) (b)

Fig. 3: CSEM score comparison of container runtimes. Lower is better. (a) Across all reduction ratios. (b) Selected reduction ratios ①, ② and ③ from (a).

5.1 Container Syscall Exposure Measure Scores

Figure 3(a) compares the CSEM scores of different container runtimes. The graph shows the change of CSEM scores as we vary the *reduction ratio* on the x-axis. Recall that the reduction ratio governs how much the weights of the *derived* system calls contribute to the final CSEM score. A reduction ratio of 0% means that the derived system calls are given the full risk weights as determined by the SCAR analysis. On the other hand, the reduction ratio of 100% implies derived system calls are ignored, and only the *pass-through* system calls are taken into account towards CSEM scoring. Overall, as expected, gVisor and Kata have much lower CSEM scores than the baseline container runtimes, and exhibit higher sensitivity to the changes in reduction ratio owing to a higher proportion of derived system calls, as shown in Table 4.

Given that the value of the *ideal* reduction ratio is subjective, Figure 3(b) shows the CSEM score differences at three spots chosen from Figure 3(a). Spot ① is the CSEM score where derived system calls are treated equally to the pass-through system calls. This is only a theoretical configuration, not conforming to any expected practical setting. This is especially disadvantageous for the container runtimes that tend to generate many derived system calls as the CSEM score of gVisor (`runsc-ptrace`) shows. On the other extreme, CSEM score at Spot ③ is where only the *pass-through* system calls are used in the calculation. The assumption here is that the derived system calls are not useful or easily craft-able for an adversarial attack, which seems unlikely as well.

Spot ② considers a more reasonable setting of reducing risk weights of derived system calls by 90%. This reflects the intuition that derived system calls can also likely, albeit with some difficulty, be exploited and hence deserve some, but smaller, risk weights. Figure C.1 in Appendix shows the contribution of each system call types to the CSEM score. A majority of the derived system calls turn out to be architecture-dependent rather than application workload-dependent.

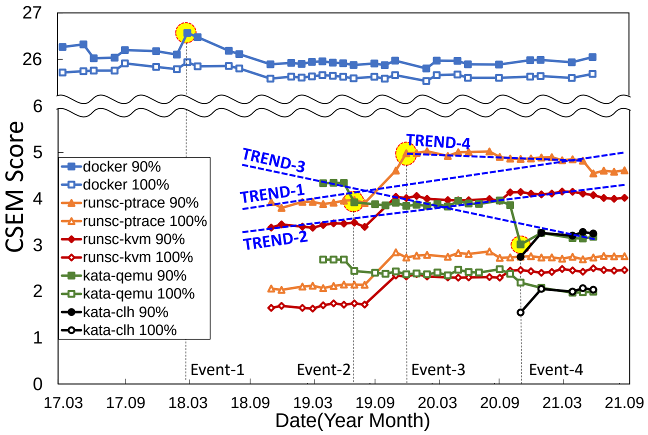


Fig. 4: CSEM score changes by container runtime versions. Two reduction ratios, 90% and 100% (latter for reference only; theoretical lower bound), are applied to 5 runtimes. Y-axis is abridged to reduce the gap.

5.2 Historical Trends Across Versions

We now present a case study of using our CSEM metric to analyze the changing trends of the security posture of container runtimes. Across a 4.5 year history, we track 31, 35 and 22 versions of runc, gVisor and Kata respectively. Figure 4 shows the trends of the changing CSEM scores for the different container runtimes:

- runc(Docker): Steady for the whole measurement period.
- runsc-ptrace: Exhibits an increasing trend in the long-term (TREND-1), but a decreasing trend in the short-term (TREND-4).
- runsc-kvm: Has a slightly increasing trend (TREND-2).
- Kata-qemu: Has a decreasing long-term trend (TREND-3).
- Kata-clh: Insufficient data points for trend analysis.

Overall, the observations from the previous Section hold across time—both of Kata and gVisor naturally being better than runc. The increasing trend of gVisor and the decreasing trend in Kata can be explained by analyzing the changes in their pass-through system calls (bigger contribution to CSEM scores—Appendix Figure C.1) across versions. This can be seen in Table 6, comparing the system calls between two opposite ends of the version history. For gVisor, the syscall count has increased over time, and the risk weights of newly added syscalls are higher than the ones removed. For Kata, the risk-weight ranks of recent version decrease and the syscall count decreases as well.

Additionally, we identify four places of interest in the figure which exhibit a sudden change of CSEM scores, labeled as Event-1 through Event-4:

- **Event-1:** Our tracking tests resulted in a weight increase in 53 syscalls in Docker v18.03.0. This was mostly because of the appearances of epoll_wait and futex syscalls. Docker release notes mentioned a version bump-up for runc, containerd and golang, which we believe to be the cause of changes in the derived syscall sets. The pass-through sets were unchanged.

Table 6: Differences of *pass-through* system calls between the first and the last versions of container runtimes

Runtime	**Version**	**Count**	**Syscalls unique to this version (Risk weight rank)**
runsc-ptrace	20180610	33	getrandom, sched_setaffinity(96)
	20210906	37	utimensat(23), futimesat(19) → utimensat(23), openat(179), munmap(172), nanosleep(174), fstat(180) → fstatfs
kata-qemu	1.5.4	46	gettimeofday(102), newfstatat(no rank), recvmsg(63), sendmsg(114)
	2.1.1	44	connect(150), fallocate(65), nanosleep(174), renameat2(166), unlink(128) → unlinkat

- **Event-2:** The score drop in Kata v1.8.0 is due to the disappearance of `write` and `futex` from the derived syscall set, which otherwise contribute significantly to the overall CSEM score (18% and 9% respectively). On cross-verification with the release notes, this was a result of QEMU upgrade in Kata.
- **Event-3:** Starting with `runsc-ptrace` v20191210, one architecture-dependent derived syscall disappeared—`getcpu`. It makes the CSEM score larger by causing the denominator smaller in the Equation 5 for all syscalls. This can be traced to `getcpu` being replaced by a golang API, as per the `gVisor` patch note.
- **Event-4:** A second score drop in Kata, this time in v1.12.0, is due to the disappearance of `gettimeofday` and `clock_gettime`, leading to a further reduction in derived system calls from 5 to 3.

Historical analysis presented in this section, made possible by SecQuant, demonstrates the important utility of quantifying the syscall exposure. It enables us to compare the degree of syscall exposure between container runtimes and uncover hidden trends. Associating the score changes with production events provides us with deeper insights.

6 Related Work

Attack Surface and Risk Metrics. Existing research proposes ways to measure the attack surface of a host kernel and how to quantify its security risk. Kurmus et al. [23] define attack surfaces of a kernel as a set of entry functions, the associated call graphs, and a set of barrier functions. Williams et al. [41, 40] use ftrace-based system call coverage as a proxy for attack surface. Li et al. [24] developed a risk metric based on popular paths through the kernel to evaluate their LibOS based scheme for securing privileged kernels. These works do not discuss how to assess the risks of system calls with respect to how they are used by exploits. Nayak et al. [33] utilize a large collection of real-world exploits as a basis for their analysis on the vulnerabilities of systems and their attack surfaces. Cheng et al. [9] developed three security metrics: the vulnerable host percentage, CVSS severity score, and compromised host percentage to evaluate the general security of an enterprise network based on vulnerability assessment. However, these works do not offer a methodology to statistically analyze the risk of system calls. Bernaschi et al. [7] presented a system call classification based on function and threat level. However, their analysis is limited to buffer overflow-based attacks, and their 4-level threat classification method is difficult to expand and update due to subjective criteria.

Measuring Container Security. Works that attempt to evaluate the security mechanisms of containers and their runtimes by analyzing the design and architecture of the respective containerization technology also exist [34, 6]. Other researchers have deployed known vulnerabilities and exploits to assess the isolation and security promises of containers and their runtimes [30, 26, 42]. For example, Lin et al. [26] deployed 88 known exploits to measure and analyze how Docker containers fare against them. This collection of exploits includes attacks against both the applications in the container as well as the host kernel. Wu et al. [42] evaluated five cloud-based container offerings against selected attacks that were chosen to exploit specific security mechanisms that were identified to be lacking in those particular offerings. However, none of these works developed a methodology for scoring the risks of system call usage found in exploits and their associated effects on the security of container runtimes.

System Call Extraction. We build upon works that devise mechanisms for extracting system calls from programs for the purpose of debloating/specializing the kernel or automatically generating seccomp profiles. Ghavamnia et al. [14, 15] have used static analysis to build out system call mappings for libc and target applications as well as whole containers create tight seccomp policies. They evaluated the security benefit of their works using a list of critical system calls derived from exploit programs as well as system calls linked to CVEs. Unfortunately, they do not specify the methodology in which the criticality of the system calls was derived nor do they provide any ranking among the security-critical/affected system calls. Similarly, works from Abubakar et al. [3] and Olufogorehan et al. [38] generate system call lists from target applications using a static analysis approach. Bulekov et al. [8] build a mapping between system calls and PHP APIs. Lopes et al. [27] perform dynamic analysis by running the target application and using unit testing combined with fuzzing to come up with their system call list. These works all have common elements with our work in how we extract the system call list.

7 Considerations for Improvements

As this is the first attempt, to the best of our knowledge, to quantify the syscall behavioral aspect of secure containers, there are several promising improvements and challenges yet to be addressed. Although each topic may lead to in-depth discussion, we only briefly outline them here due to space constraints.

- *Benign application*: Incorporating known benign applications into the syscall analysis can further improve the validity of our syscall risk weights. For example, if some system calls are found to be used heavily in exploit codes, but not in benign applications, this may be a ground to increase the risk weights. However, the challenge is to select benign applications that are *representative*.
- *Argument checking of system call tracing and the need for systematic argument fuzzing*: Current SCED process for testing the syscall path-through behavior can be made more accurate and comprehensive by extending test cases with systematic argument fuzzing. Our experience suggested that syscall pass-through behaviors can vary by syscall argument values. In addition, we need

to enhance our implementation to better observe argument values and how one syscall translates into another while passing through the proxy kernel.
- *Validity of using exploit codes publicly available*: Exploit codes we used are all publicly available ones. This may raise a concern that some of these *open* exploit codes may not resemble the *real, unknown* ones. Our current assumption is that the core part of the attack logic remains similar since they are based on the same principle. However, as we find more exploit codes, we can easily incorporate them into our automated analysis and adjust scores.

8 Conclusion

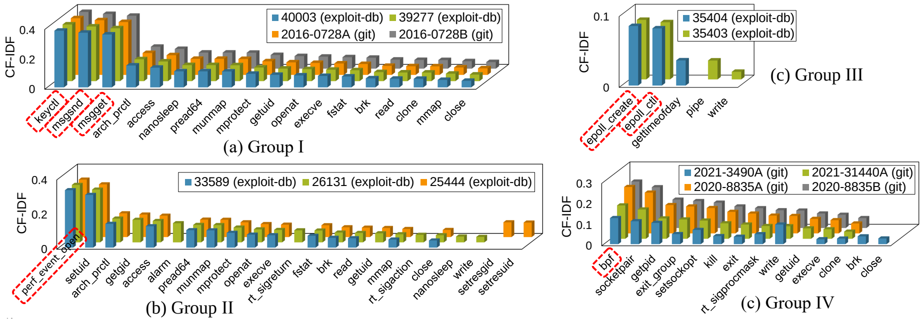
In this work, we presented a novel syscall exposure quantification technique, SecQuant, for secure container runtimes. SecQuant works by combining the system call risk weights obtained from IR-based analysis on a large set of exploit codes, and the system call pass-through/filter behavior of runtimes through extensive experimentation. Our analysis revealed several interesting syscall pass-through behaviors with varying types and numbers of syscalls reaching the host kernel. According to our metric, secure container runtimes have 4.2 to 7.5 times smaller syscall exposure. We have also found that there exist both increasing and decreasing trends in syscall exposures of container runtimes. SecQuant can further be improved by employing more accurate syscall-to-exploit mapping techniques and more general and accurate pass-through test platforms in the future. Especially, a technique for comparing the arguments of application and host-arrived syscalls can enable more sophisticated quantification.

Acknowledgement

This work was supported by the National Research Foundation of Korea(NRF) grant funded by the Korea government(MSIT) (No. NRF-2021R1A5A1021944). Authors S. Jang and S. Song contributed equally to this work. Corresponding author is B. Tak.

A Complete Ranking of System Calls by Risk Weights

We provide a complete list of system calls ranked by the risk weights calculated by our CF-IDF methodology in Table A.1.



(a) Group I

(c) Group III

(b) Group II

(c) Group IV

Fig. A.1: Visual comparison of the similarity between risk weight vectors of exploits within different groups

Table A.1: List of syscalls ranked by risk weights obtained by SCAR

rank	system call	weight	rank	system call	weight	rank	system call	weight	rank	system call	weight	rank	system call	weight
1	capset	0.439551	37	shmat	0.244536	75	mkdir	0.193169	112	pipe	0.139035	149	setgid	0.104811
2	add_key	0.409431	37	sigaltstack	0.244536	76	getdents64	0.188096	113	getppid	0.138948	150	connect	0.103625
3	recvmmsg	0.392371	37	setxattr	0.244536	77	_llseek	0.187949	114	sendmsg	0.136639	151	prlimit64	0.101779
4	getresuid	0.388023	41	symlink	0.244057	77	getpriority	0.187949	115	setresgid	0.136633	152	seccomp	0.101097
4	sendfile	0.388023	42	getcwd	0.244007	79	setns	0.187368	116	uselib	0.136261	153	stat	0.100942
4	io_uring_register	0.388023	43	fchmod	0.240128	80	msgrcv	0.185928	116	msync	0.136261	154	setitimer	0.092306
7	shutdown	0.335366	44	modify_ldt	0.237571	81	getsockname	0.184515	118	mincore	0.135555	155	setsockopt	0.091907
8	settimeofday	0.334059	44	clock_gettime	0.237571	82	setrlimit	0.175967	119	uname	0.135125	156	lseek	0.087384
9	rename	0.329819	46	process_vm_readv	0.237358	83	getrlimit	0.175651	120	pause	0.134133	157	wait4	0.084849
10	creat	0.329663	47	writev	0.235578	83	sync	0.175651	121	vmsplice	0.130688	158	exit_group	0.080511
11	keyctl	0.32028	48	getdents	0.234431	85	splice	0.174387	122	alarm	0.128849	159	getpid	0.078393
12	fchown	0.316477	49	sendmmsg	0.232625	86	ptrace	0.169211	123	setresuid	0.128281	160	ioctl	0.077965
12	flock	0.316477	50	syslog	0.232287	87	setpriority	0.167943	124	gettid	0.128004	161	arch_prctl	0.073672
12	mknod	0.316477	51	mount	0.226888	88	userfaultfd	0.167438	125	epoll_create1	0.126591	162	rt_sigaction	0.072893
12	mq_notify	0.316477	52	rmdir	0.224417	89	futex	0.166319	125	setgroups	0.126591	163	kill	0.069353
12	io_setup	0.316477	53	getgroups	0.219776	90	statfs	0.165307	125	umask	0.126591	164	access	0.068628
12	io_submit	0.316477	54	select	0.216088	91	dup2	0.164186	128	unlink	0.126579	165	exit	0.06426
12	kcmp	0.316477	55	pwrite64	0.21538	92	accept	0.164048	129	time	0.123122	166	renameat2	0.063323
19	futimesat	0.30329	55	set_mempolicy	0.21538	93	perf_event_open	0.163461	130	socketpair	0.122321	167	sysinfo	0.061537
19	inotify_rm_watch	0.30329	55	readv	0.21538	94	poll	0.15674	131	geteuid	0.121388	167	setreuid	0.061537
19	inotify_init1	0.30329	55	sched_getaffinity	0.21538	95	getsockopt	0.15602	132	setuid	0.120857	169	socket	0.059526
19	restart_syscall	0.30329	55	shmdt	0.21538	96	sched_setaffinity	0.155237	133	ftruncate	0.1204	170	rt_sigprocmask	0.058141
19	utimensat	0.30329	60	mremap	0.212287	97	timerfd_create	0.154563	134	mlock	0.119328	171	pread64	0.055834
24	clock_nanosleep	0.303143	61	inotify_init	0.210391	97	timerfd_settime	0.154563	135	setsid	0.119298	172	munmap	0.055819
25	umount2	0.295618	62	sched_yield	0.206736	99	unshare	0.153808	136	epoll_ctl	0.116779	173	write	0.055482
26	chown	0.286681	63	recvmsg	0.205074	100	fcntl	0.153628	137	sendto	0.115332	174	nanosleep	0.055054
27	link	0.284955	64	getegid	0.204785	101	madvise	0.152935	138	setpgid	0.113589	175	open	0.05293
28	dup3	0.272522	65	fallocate	0.202194	102	gettimeofday	0.151181	139	getgid	0.113317	176	getuid	0.052282
29	eventfd2	0.269163	65	_sysctl	0.202194	103	tgkill	0.150511	140	getresgid	0.113173	177	mprotect	0.049306
30	msgsnd	0.267191	65	move_pages	0.202194	104	personality	0.150292	140	adjtimex	0.113173	178	execve	0.04722
31	sched_setscheduler	0.264745	68	shmctl	0.200075	105	listen	0.148191	140	timer_create	0.113173	179	openat	0.043193
32	inotify_add_watch	0.261581	69	msgctl	0.199029	106	prctl	0.146587	140	memfd_create	0.113173	180	fstat	0.040249
32	waitid	0.261581	70	dup	0.197707	107	readlink	0.144822	144	epoll_wait	0.108171	181	clone	0.035285
34	msgget	0.254518	71	io_uring_enter	0.194227	108	chroot	0.142477	145	set_tid_address	0.107822	182	brk	0.034785
35	pipe2	0.25433	71	io_uring_setup	0.194227	109	bpf	0.142335	145	set_robust_list	0.107822	183	read	0.032339
36	chmod	0.248783	73	chdir	0.19396	110	recvfrom	0.140137	147	bind	0.106639	184	mmap	0.03019
37	shmget	0.244536	74	iopl	0.193403	111	epoll_create	0.139559	148	rt_sigreturn	0.1059	185	close	0.026151

B Break-down of Sample Risk Weights

To gain better understanding of ranks and scores presented in Section 4.2, we provide four sample system calls `shmdt`, `capset`, `add_key` and `io_uring_register` with details of how the scores are computed.

The first example shows the impact of CVSS scores for the same CVE class size. Although both `shmdt` and `capset` appear only in one exploit code each (CVE-2019-15666 and CVE-2000-0506, respectively), they have very different risk-weight rankings—55 vs 1. For `shmdt`, the IDF value is 0.88, but the CF is only 0.245 because of the low CVSSv2 score 4.9. On the other hand, `capset` has the same IDF value but CF of 0.5 because of the high CVSSv2 score 10.

In the case of `add_key` and `io_uring_register`, the size of the class affected the rank (2 vs. 4) following the Equation 2. Since `add_key` appear in three exploit codes belonging to CVE-2016-8655 and `io_uring_register` appear in two exploit codes belonging to CVE-2020-29534, it grants the IDF score of about 0.76 to `add_key` and about 0.81 to `io_uring_register`. The CVEs of `add_key` and `io_uring_register` have the same CVSS score (7.2) but the class size (3 vs. 2) is different. Thus, 0.75 and 0.67 are multiplied by the normalized CVSS score. Eventually, `io_uring_register` has a higher IDF score than `add_key`, but the total weight is lower.

C Experiment Setup

runc v1.0.0-rc10. gVisor v20210906. Kata v2.1.0. Host: Ubuntu 20.04 / Linux 5.11. For historical trends, running older versions of container runtimes required setup of compatible environments including older OS versions, e.g. Ubuntu 16.04 / Linux 4.4 for Docker. glibc v2.33 used for extracting libc-to-syscall mapping.

Table C.1: System call groups that share kernel functions

Syscall Group	Shared Kernel Fxn
open, openat, creat	do_sys_open
link, linkat	do_linkat
mkdir, mkdirat	do_mkdirat
mknod, mknodat	do_mknodat
symlink, symlinkat	do_symlinkat
readlink, readlinkat	do_readlinkat
read, pread64	vfs_read
readv, preadv, preadv2	vfs_readv
rename, renameat renameat2	do_renameat2
unlink, unlinkat	do_unlinkat
chmod, fchmod, fchmodat	chmod_common
statfs, fstatfs	do_statfs_native
utimensat, utime utimes, futimesat	do_utimes
fchownat, chown lchown, fchown	chown_common
write, pwrite64,	vfs_write
writev, pwritev, pwritev2	vfs_writev
clone, fork	kernel_clone

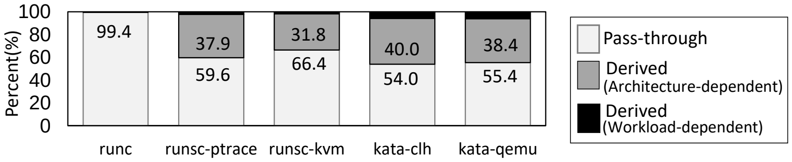


Fig. C.1: CSEM score break-down by system call types at 90% reduction ratio

References

1. Exploit Database. https://www.exploit-db.com., online; accessed 12 October 2021
2. Project Zero. https://bugs.chromium.org/p/project-zero/issues/list., online; accessed 12 October 2021
3. Abubakar, M., Ahmad, A., Fonseca, P., Xu, D.: Shard: Fine-grained kernel specialization with context-aware hardening. In: USENIX Security Symposium (2021)
4. Agache, A., Brooker, M., Iordache, A., Liguori, A., Neugebauer, R., Piwonka, P., Popa, D.M.: Firecracker: Lightweight virtualization for serverless apps. In: NSDI'20
5. AWS: Lambda. https://aws.amazon.com/ko/lambda/ (2014), accessed Oct 2021
6. Babar, A., Ramsey, B.: Understanding container isolation mechanisms for building security-sensitive private cloud. Technical Report CREST (2017)
7. Bernaschi, M., Gabrielli, E., Mancini, L.V.: Operating system enhancements to prevent the misuse of system calls. In: Proceedings of the 7th ACM Conference on Computer and Communications Security. p. 174–183 (2000)
8. Bulekov, A., Jahanshahi, R., Egele, M.: Saphire: Sandboxing php applications with tailored system call allowlists. In: 30th USENIX Security Symposium (2021)
9. Cheng, Y., Deng, J., Li, J., DeLoach, S.A., Singhal, A., Ou, X.: Metrics of security. In: Cyber defense and situational awareness, pp. 263–295. Springer (2014)
10. Cloud Hypervisor: https://github.com/cloud-hypervisor/cloud-hypervisor., online; accessed 12 October 2021
11. Combe, T., Martin, A., Di Pietro, R.: To docker or not to docker: A security perspective. IEEE Cloud Computing **3**(5), 54–62 (2016)
12. CVE: https://cve.mitre.org., online; accessed 12 October 2021
13. Firecracker: https://firecracker-microvm.github.io., online; accessed 22 June 2022
14. Ghavamnia, S., Palit, T., Benameur, A., Polychronakis, M.: Confine: Automated system call policy generation for container attack surface reduction. In: The 23rd International Symposium on Research in Attacks, Intrusions and Defenses (2020)
15. Ghavamnia, S., Palit, T., Mishra, S., Polychronakis, M.: Temporal system call specialization for attack surface reduction. In: USENIX Security Symposium (2020)
16. Google: Cloud Function. https://cloud.google.com/functions (2016), online; accessed 10 October 2021
17. gVisor: https://github.com/google/gvisor/., online; accessed 17 May 2022

18. Hunt, P., Hansman, S.: A taxonomy of network and computer attack methodologies. Computers and Security **24**(1), 31–43 (2003)
19. IBM: IBM Cloud Functions. https://cloud.ibm.com/functions/ (2016), online; accessed 10 October 2021
20. Kata Containers: https://katacontainers.io/., online; accessed 17 May 2022
21. Kuenzer, S., Bădoiu, V.A., Lefeuvre, H., Santhanam, S., Jung, A., Gain, G., Soldani, C., Lupu, C., Teodorescu, Ş., Răducanu, C., et al.: Unikraft: fast, specialized unikernels the easy way. In: EuroSys (2021)
22. Kuo, H.C., Williams, D., Koller, R., Mohan, S.: A linux in unikernel clothing. In: EuroSys (2020)
23. Kurmus, A., Tartler, R., Dorneanu, D., Heinloth, B., Rothberg, V., Ruprecht, A., Schröder-Preikschat, W., Lohmann, D., Kapitza, R.: Attack surface metrics and automated compile-time os kernel tailoring. In: NDSS (2013)
24. Li, Y., Dolan-Gavitt, B., Weber, S., Cappos, J.: Lock-in-pop: securing privileged operating system kernels by keeping on the beaten path. In: USENIX ATC (2017)
25. Lie, D., Satyanarayanan, M.: Quantifying the strength of security systems. In: USENIX HOTSEC (2007)
26. Lin, X., Lei, L., Wang, Y., Jing, J., Sun, K., Zhou, Q.: A measurement study on linux container security: Attacks and countermeasures. In: ACSAC (2018)
27. Lopes, N., Martins, R., Correia, M.E., Serrano, S., Nunes, F.: Container hardening through automated seccomp profiling. In: Proceedings of the 2020 6th International Workshop on Container Technologies and Container Clouds. pp. 31–36 (2020)
28. LTP: Linux Test Project. https://github.com/linux-test-project/ltp., online; accessed 12 October 2021
29. Manco, F., Lupu, C., Schmidt, F., Mendes, J., Kuenzer, S., Sati, S., Yasukata, K., Raiciu, C., Huici, F.: My vm is lighter (and safer) than your container. In: Proceedings of the 26th Symposium on Operating Systems Principles (2017)
30. Martin, A., Raponi, S., Combe, T., Pietro, R.D.: Docker ecosystem - vulnerability analysis. In: Computer Communications 122. p. 30—43 (2018)
31. Microsoft: Azure Function. https://azure.microsoft.com/en-us/services/functions/
32. Nabla Containers: A new approach to Container Isolation. https://nabla-containers.github.io/., online; accessed 12 October 2021
33. Nayak, K., Marino, D., Efstathopoulos, P., Dumitraş, T.: Some vulnerabilities are different than others. In: Workshop on Recent Advances in Intrusion Detection'14
34. Reshetova, E., Karhunen, J., Nyman, T., Asokan, N.: Security of os-level virtualization technologies: Technical report. Secure IT Systems (2014)
35. Sahil Suneja: The choices we make: Impact of using host filesystem interface for secure containers. https://nabla-containers.github.io/2018/11/28/fs/ (2018)
36. Sultan, S., Ahmad, I., Dimitriou, T.: Container security: Issues, challenges, and the road ahead. IEEE Access **7**, 52976–52996 (2019)
37. Syzkaller: Kernel Fuzzer. https://github.com/google/syzkaller, accessed Oct 2021
38. Tunde-Onadele, O., Lin, Y., He, J., Gu, X.: Self-patch: Beyond patch tuesday for containerized applications. In: IEEE ACSOS (2020)
39. Viktorsson, W., Klein, C., Tordsson, J.: Security-performance trade-offs of kubernetes container runtimes. In: IEEE MASCOTS (2020)
40. Williams, D., Koller, R., Lucina, M., Prakash, N.: Unikernels as processes. In: Proceedings of the ACM Symposium on Cloud Computing. p. 199–211 (2018)
41. Williams, D., Koller, R., Lum, B.: Say goodbye to virtualization for a safer cloud. In: 10th USENIX Workshop on Hot Topics in Cloud Computing (2018)
42. Wu, Y., Lei, L., Wang, Y., Sun, K., Meng, J.: Evaluation on the security of commercial cloud container services. In: ISC (2020)