Effectiveness Analysis of System call Sequence-based Container Security Mechanism

Somin Song, Youyang Kim, Byungchul Tak

Kyungpook National University (KNU), Daegu, Republic of Korea

한국정보과학회
KOREAN INSTITUTE OF INFORMATION SCIENTISTS AND ENGINEERS

KNU
KYUNGPOOK NATIONAL UNIVERSITY

December 21st, 2022

Rapid Growth in Cloud Adaption and Security Concern

- The total volume of data that will be stored in the cloud by 2025, which accounts for **50%** of all the data in the world

– ArcServe, 2020 [1]

- According to 74% of global IT decision-makers, **95%** of all workload will be in the cloud within the next five years

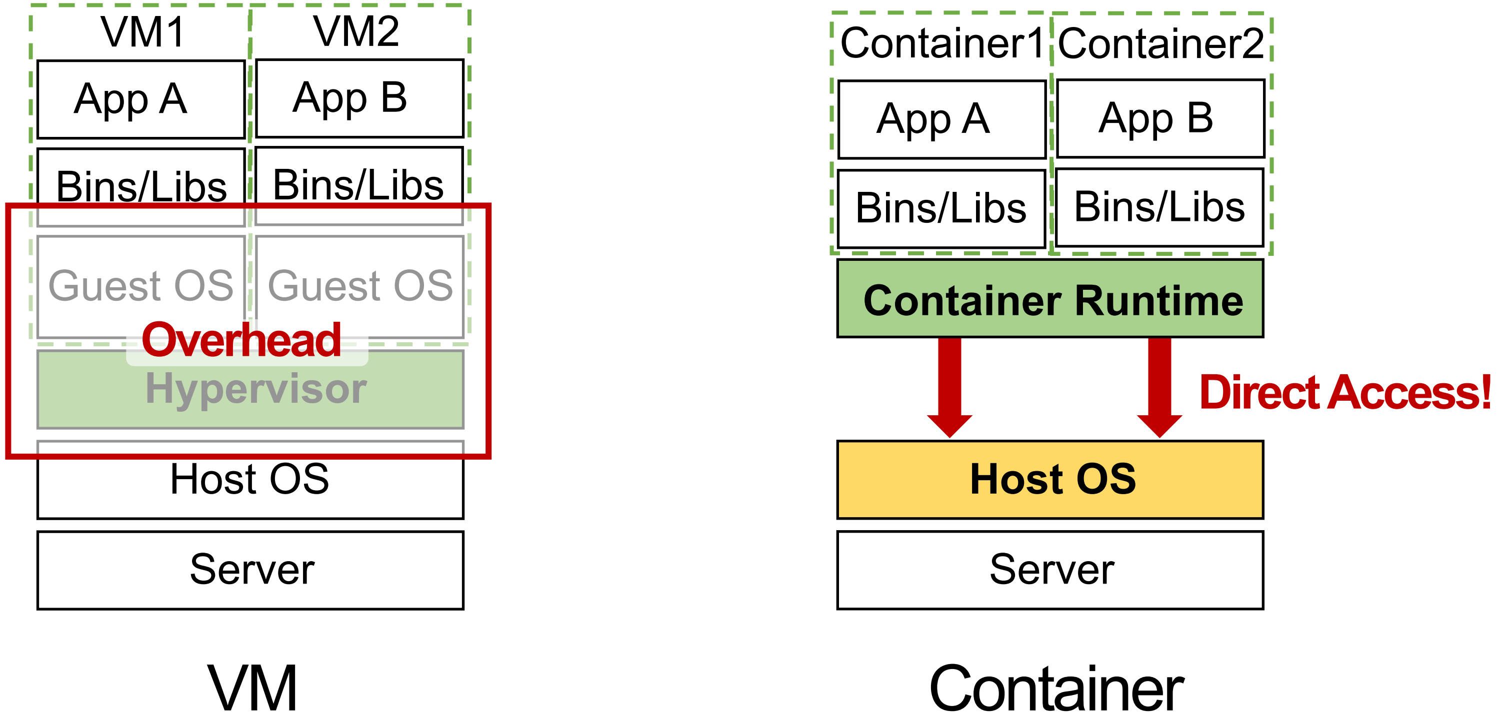
– LogicMonitor, 2020 [1]

- **75%** of enterprises and **90%** of cyber security experts agree that **security is their top concern**

– ArcServe, 2020 [1]

Container Security in Cloud Native Environment

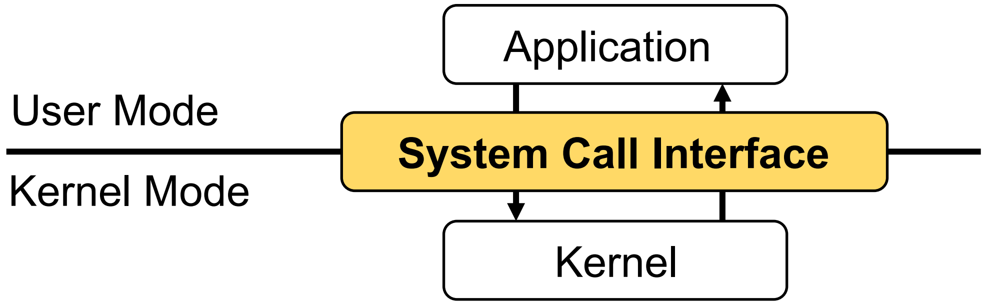
- Containers are a **key virtualization technology** in cloud computing



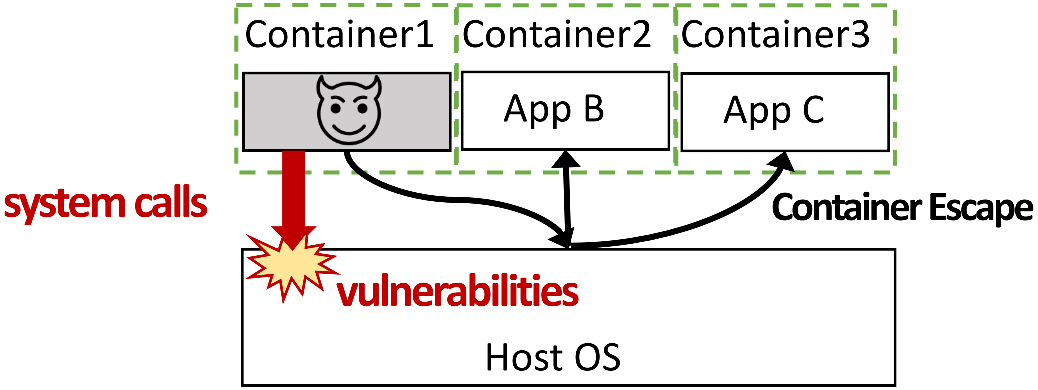
Containers **share** a Host OS!

Container Escape through System Call

- System Call



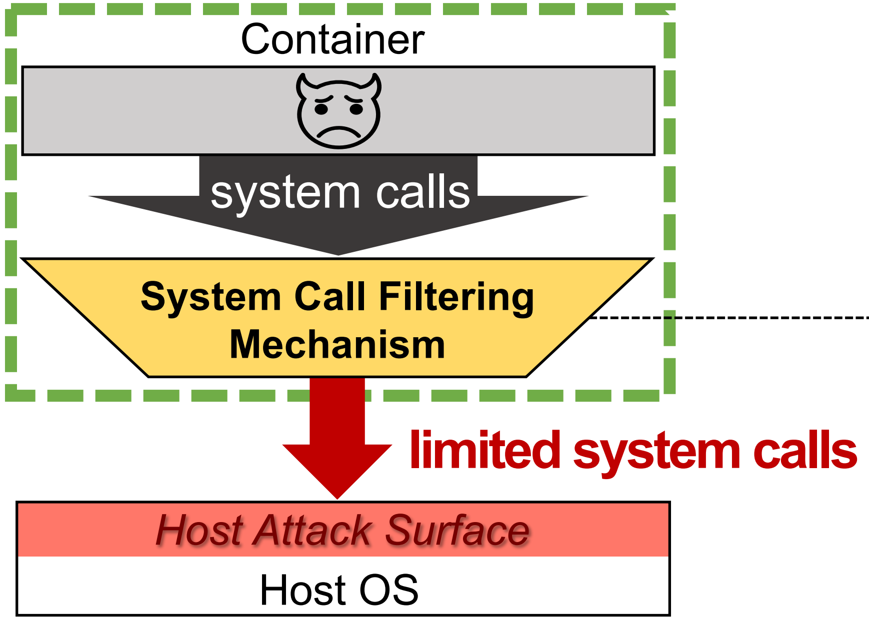
- Container Escape
  - Exploitation of Host kernel vulnerabilities through carefully **crafted system calls**



- Dirty COW Docker Escape (CVE-2016-5195)
- Runc Container Escape (CVE-2019-5736)
- Kubernetes Container Escape (CVE-2022-0185)
- Dirty Pipe Container Escape (CVE-2022-0847)
- DirtyCred Container Escape (CVE-2022-2588)

System Call Filtering Protection Mechanism

The fewer system calls, the more secure!

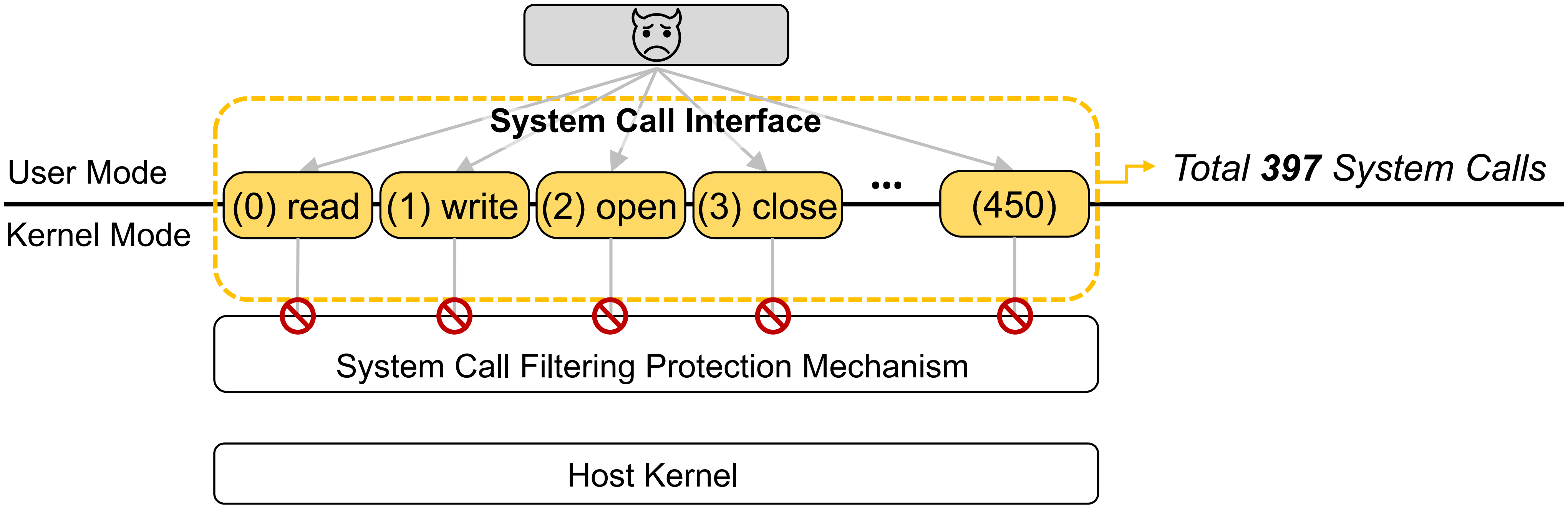


Seccomp (Secure Computing mode)

- Linux kernel feature
- Blocks the system calls
- Restrict a Container’s system calls

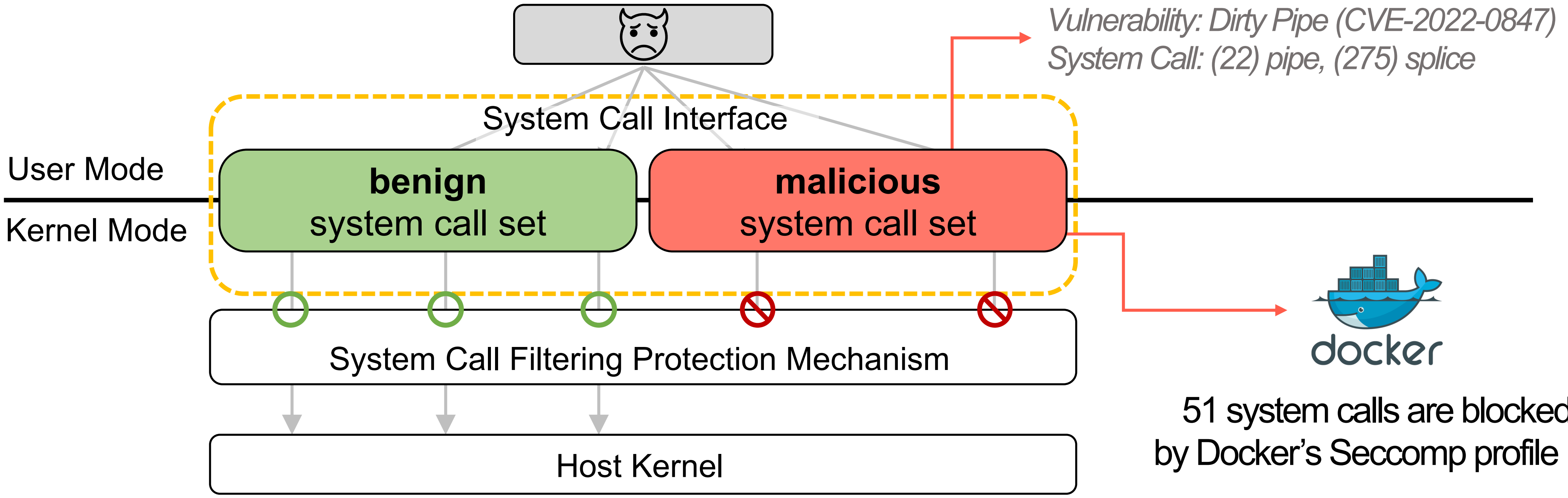
System Call Filtering Protection Mechanism

- "Which System Call should be **allowed/blocked**?"



System Call Filtering Protection Mechanism

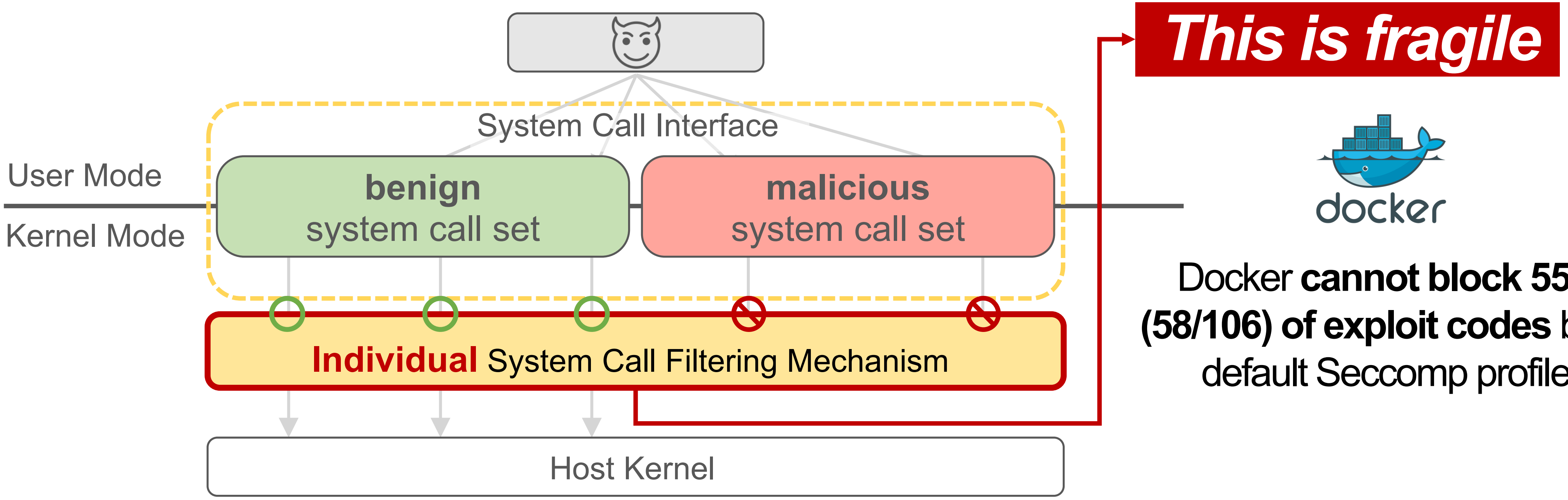
- “Which System Call should be allowed/blocked?”
  - **Which System Call is benign/malicious?**



51 system calls are blocked
by Docker’s Seccomp profile [2]

Problem

- Limitation in Individual System call Filtering Protection Mechanism



Docker **cannot block 55% (58/106) of exploit codes** by its default Seccomp profile.

Our Intuition and Approach

- System call Filtering Method: ***system call sequence***
  - without the risk of blocking normal application operations
  - It can thwart exploits that cannot be blocked by individual syscall-based filtering

- Syscall sequence blocking can mitigate Dirty Pipe vulnerability

  → Docker’s default Seccomp profile with an individual syscall blocking cannot defend it

Syscall Sequence Policy
(generated by analyzing the exploit codes)

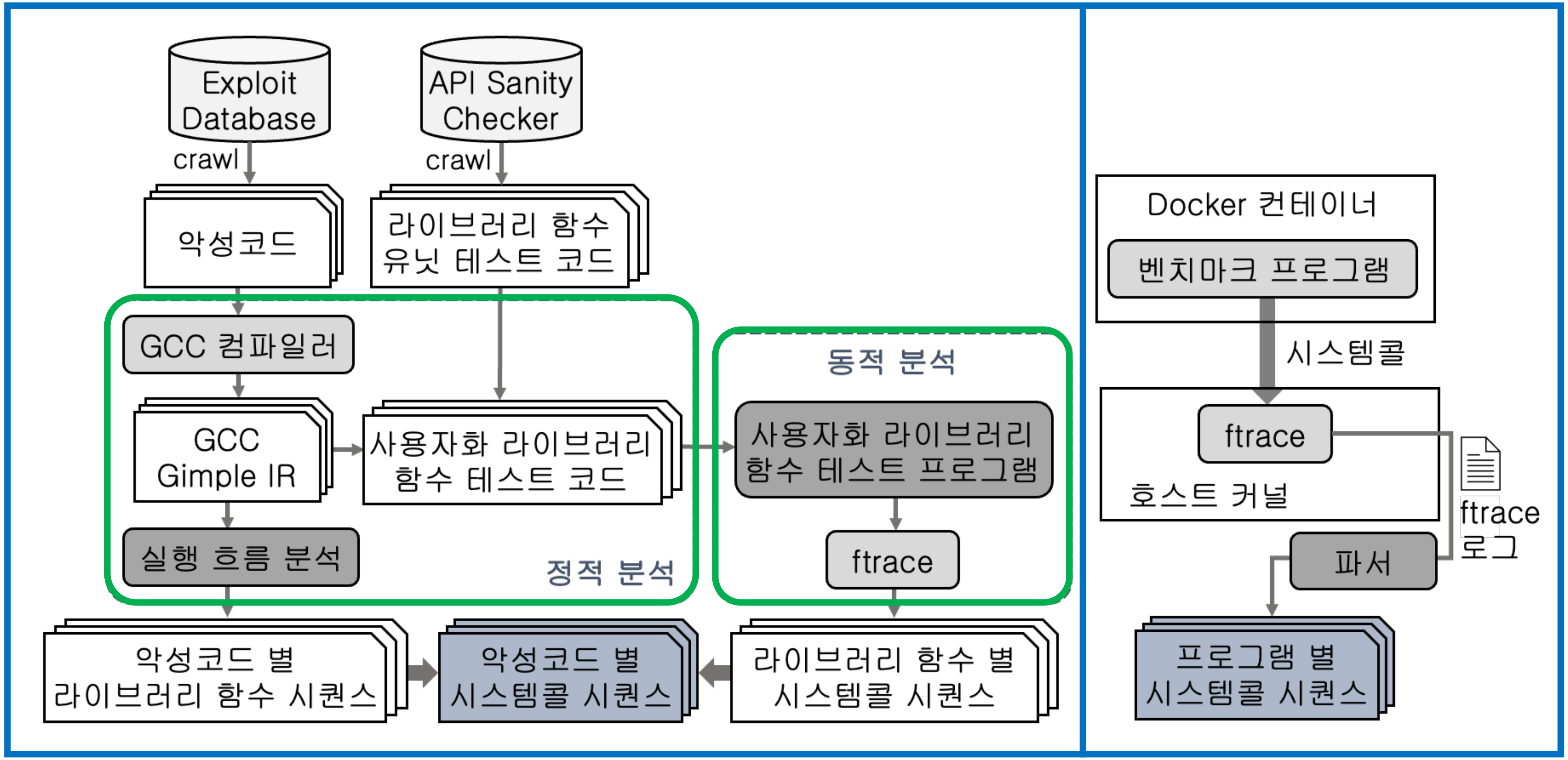
openat → fstat → pipe → fcntl → write → read → splice → write → close

Observed Syscall Sequence
(on the host when the exploit code is executed in Docker)

openat → fstat → pipe → fcntl → write → read → splice → write → close

- Goal
  1) To analyze limitations of individual system call-based filtering mechanism
  2) To evaluate the security of system call sequence-based filtering mechanism

Approach Overview: Syscall Sequence Analysis



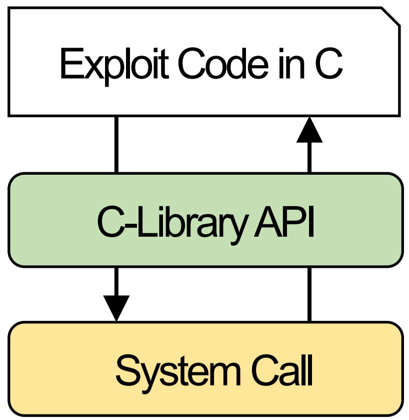
(1) Exploit Code Analysis **(2) Application Analysis**

Exploit Code Analysis Methodology

- Goal: To investigate the system calls that the exploit codes invoke

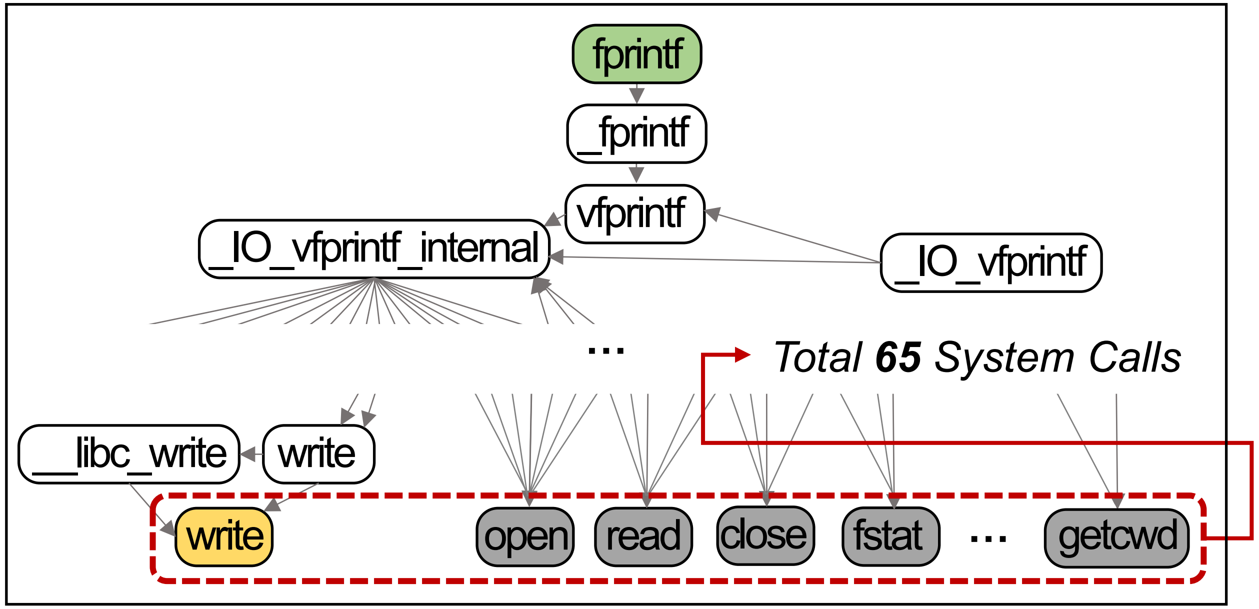
- Only Dynamic Analysis [8,9,10]
  - Method of tracing system calls while directly executing the exploit codes
  - \+ Upon successful execution, complete information can be obtained
  - – It is very difficult to set up the environment to run the exploit code

- Only Static Analysis [4,5,6,7]
  - Method of analyzing the source code/binary without executing the exploit codes
  - \+ Automated Large-scale analysis is possible
  - – C language makes static analysis difficult
    - The indirect call caused by a function pointer, and deeply nested macro code



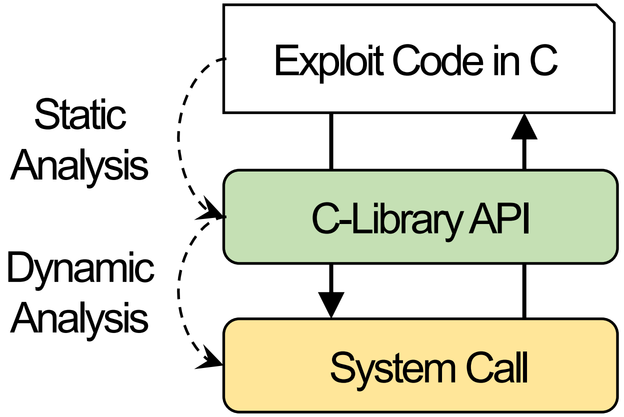
Problems in Precedent Researches

- Challenge in **Static Analysis**
  - Confine [4] (RAID '20) : built libc-to-syscall call graph
    - ► ***boating reachable system calls***



Exploit Code Analysis Methodology

- Hybrid Analysis
  - Combine static analysis and dynamic analysis to generate a system call sequence corresponding to each exploit code
  - **Static Analysis**: to extract library function sequence on all possible control flows where the exploits can be successfully triggered
  - **Dynamic Analysis**: to build a mapping between library functions and system call sequences



Static Analysis to Map Exploit Code-Glibc function

- Using GCC + GCC GIMPLE IR (Intermediate Representation)
  - Getting info on control flow of glibc function

```
void foo() {
  int i,j;
  do{
    i = getuid();
    j = geteuid();
  } while(i == j);
  bar();
}
```

Example source code

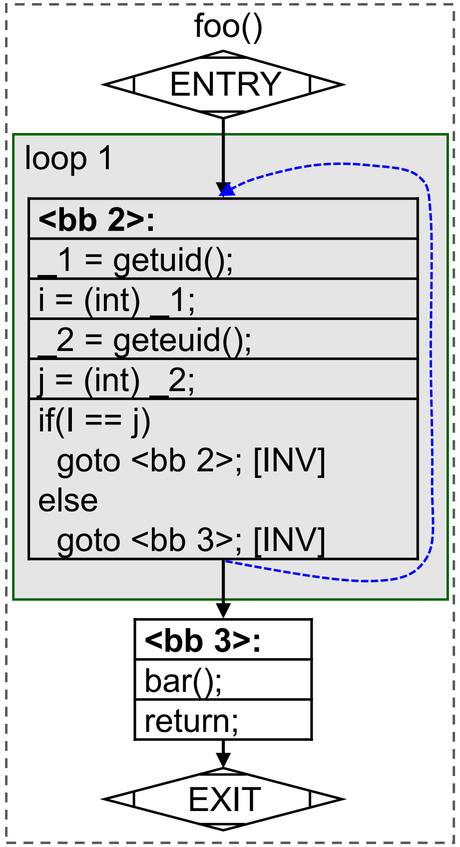
```
foo: getuid-geteuid-getuid-geteuid-getuid-geteuid-bar
```

Library function call sequence(s) for code

```
;; Function foo (…)
            …
;; Loop 1
;;  header 2, latch 2
;;  depth 1, outer 0
;;  nodes: 2
;; 2 succs { 2 3 }
;; 3 succs { 1 }
foo ()
{
             …
 <bb 2> :
 _1 = getuid ();
 i = (int) _1;
 _2 = geteuid ();
 j = (int) _2;
 if (i == j)
   goto <bb 2>; [INV]
 else
   goto <bb 3>; [INV]

 <bb 3> :
 bar ();
 return;
}
```

GCC's GIMPLE IR



Visualization

Dynamic Analysis to Map Glibc function-System Call

- Dynamic Analysis (using API Sanity Checker dataset [11] + ftrace mechanism)

(a) API Sanity Checker Dataset

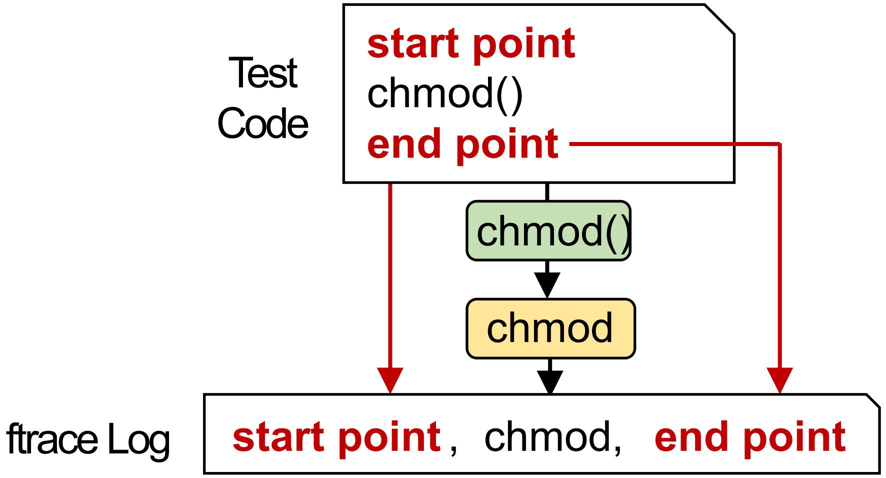
- An automatic generator of basic unit tests for a shared C/C++ library

(b) ftrace mechanism

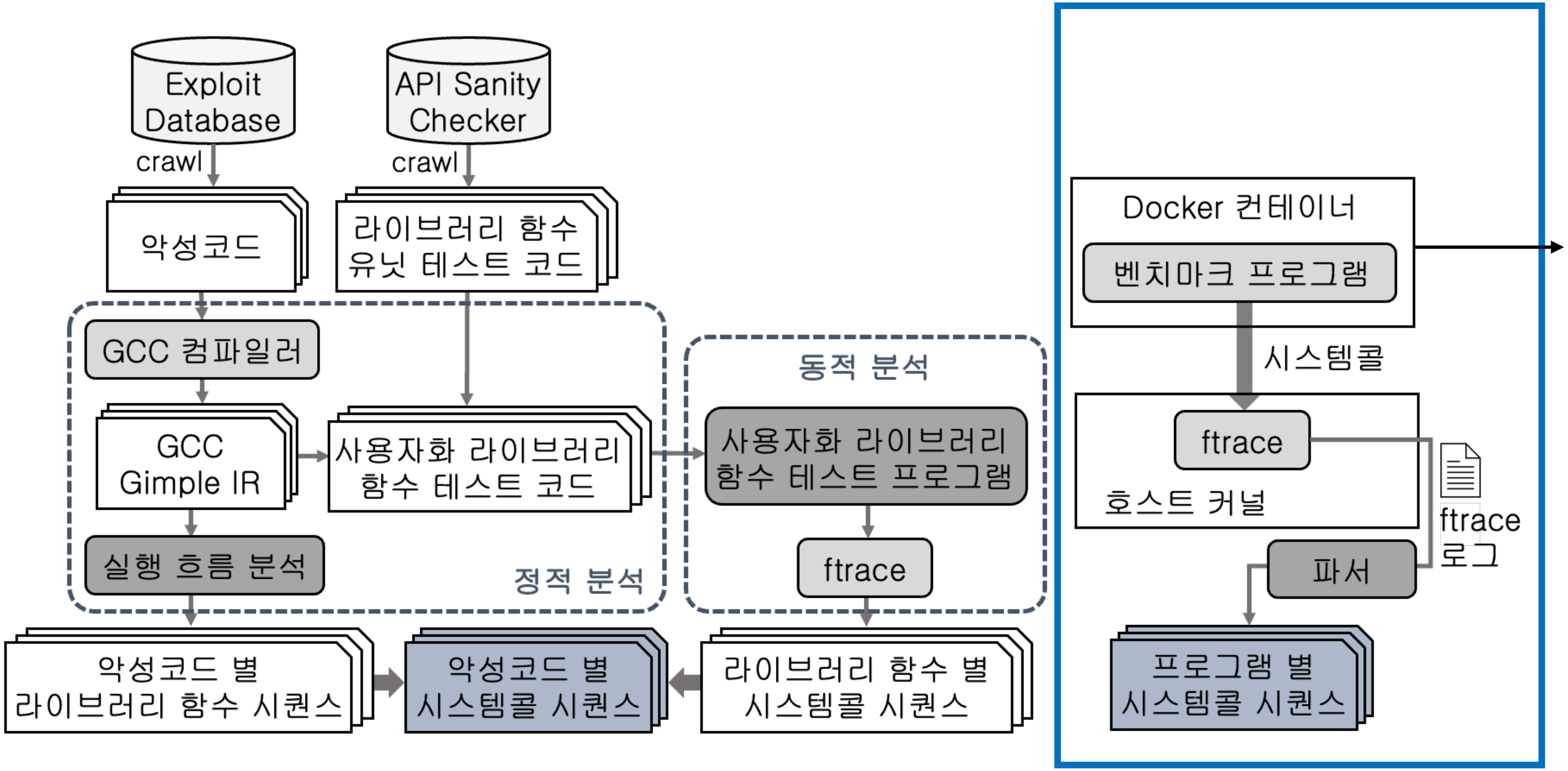
- Tracing tool in Linux kernel
- Possible to write directly to log file

```
#include <rpc/types.h>
#include <sys/stat.h>>

int main(int argc, char *argv[])
{
   __mode_t__mode =umask(0);
   chmod ((const char *) "/proc/self/exec", 3565);
   return 0;
}
```



Approach Overview: Syscall Sequence Analysis



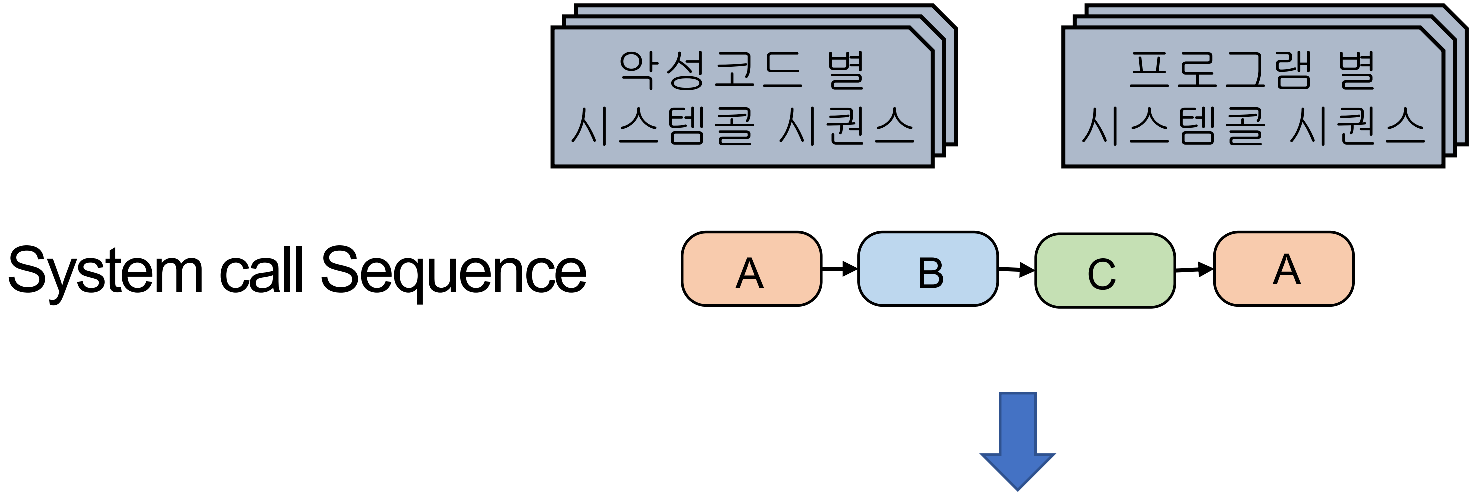
4 Server App
(nginx, httpd, tomcat, node)

4 DB App
(mongodb, mysql, mariadb, redis)

(b) Application Analysis

Approach Overview: Syscall Sequence Analysis

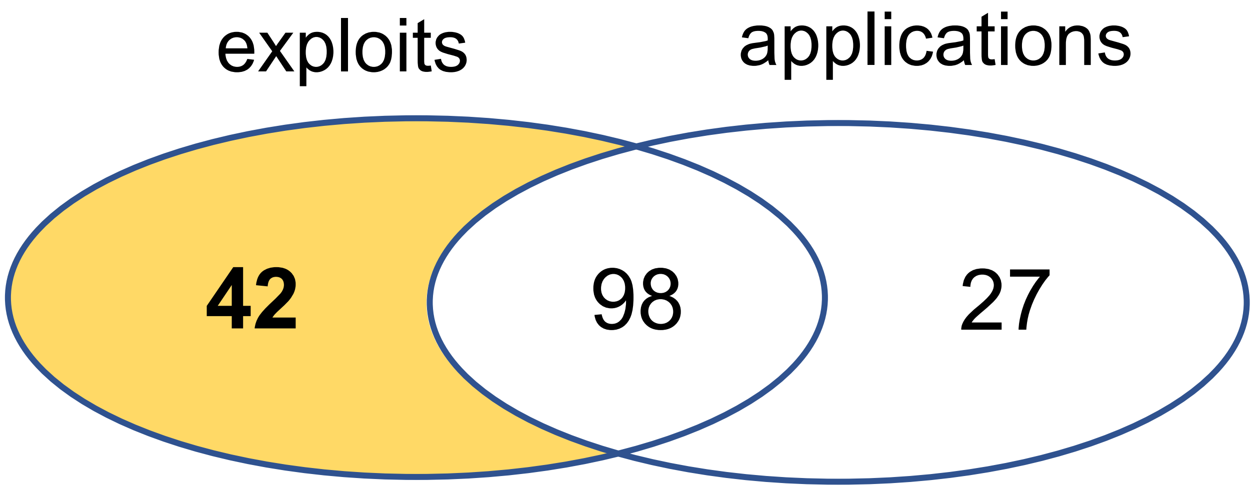
- N-gram Analysis



N	N-gram system call sequence	#
2	A-B, B-C, C-A	3
3	A-B-C, B-C-A	2
4	A-B-C-A	1

Experimental evaluation

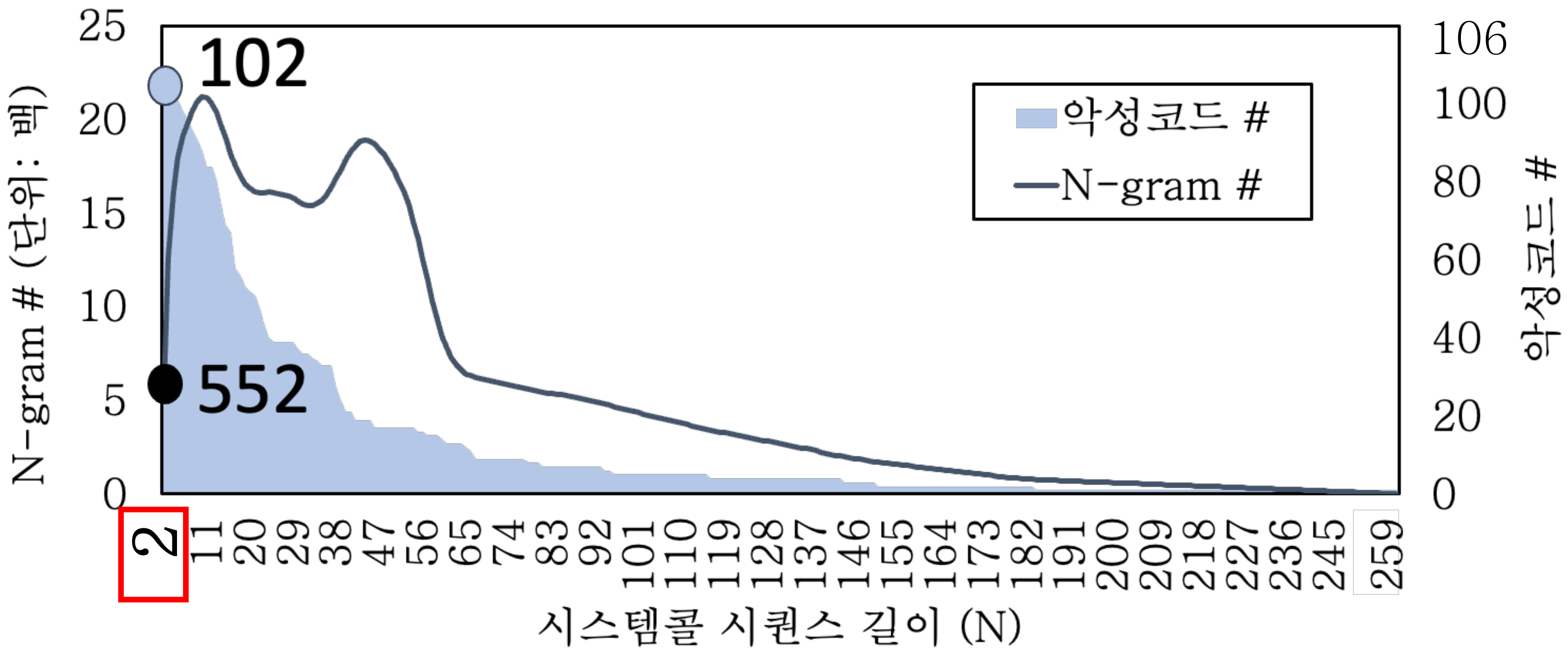
- Evaluation of **individual** system call-based filtering mechanisms



➡ **67 (about 63%)** out of 106 exploit codes could be blocked.

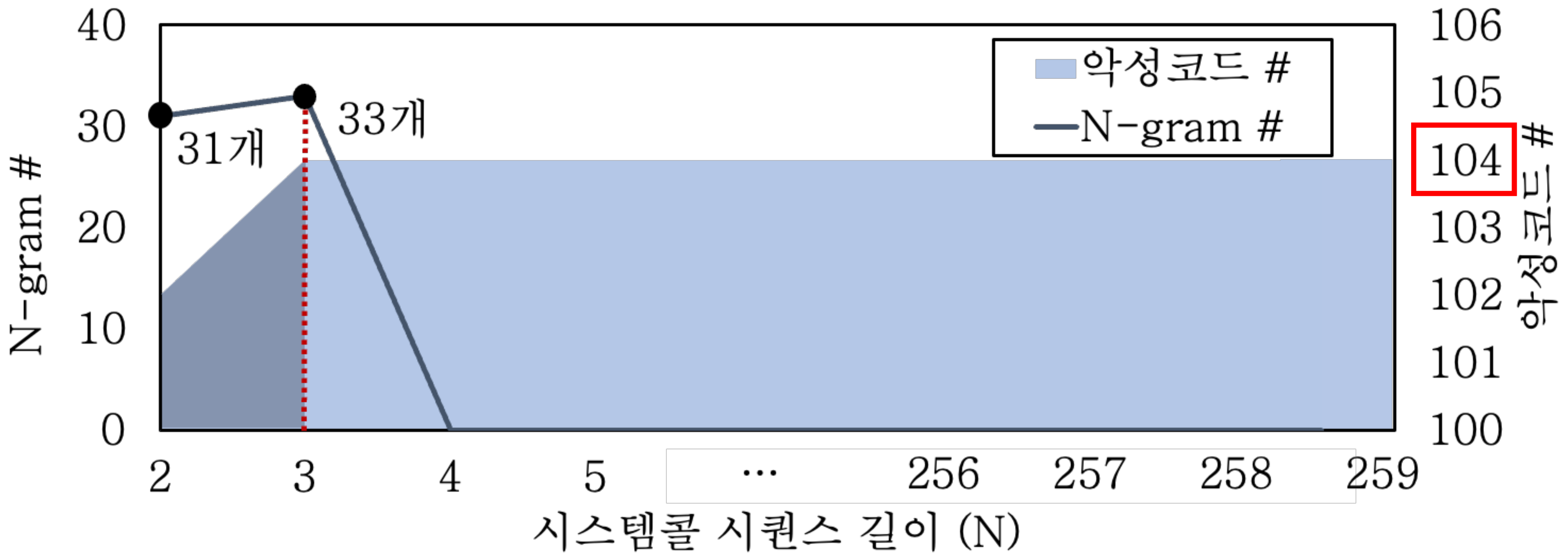
Experimental evaluation

- Evaluation of system call **sequence**-based filtering mechanisms
  - **N-gram set from exploits – N-gram set from applications**



Experimental evaluation

- Evaluation of system call **sequence**-based filtering mechanisms
  - N-gram set from exploits - N-gram set from applications



104 (about 98%) out of 106 exploit codes could be blocked.

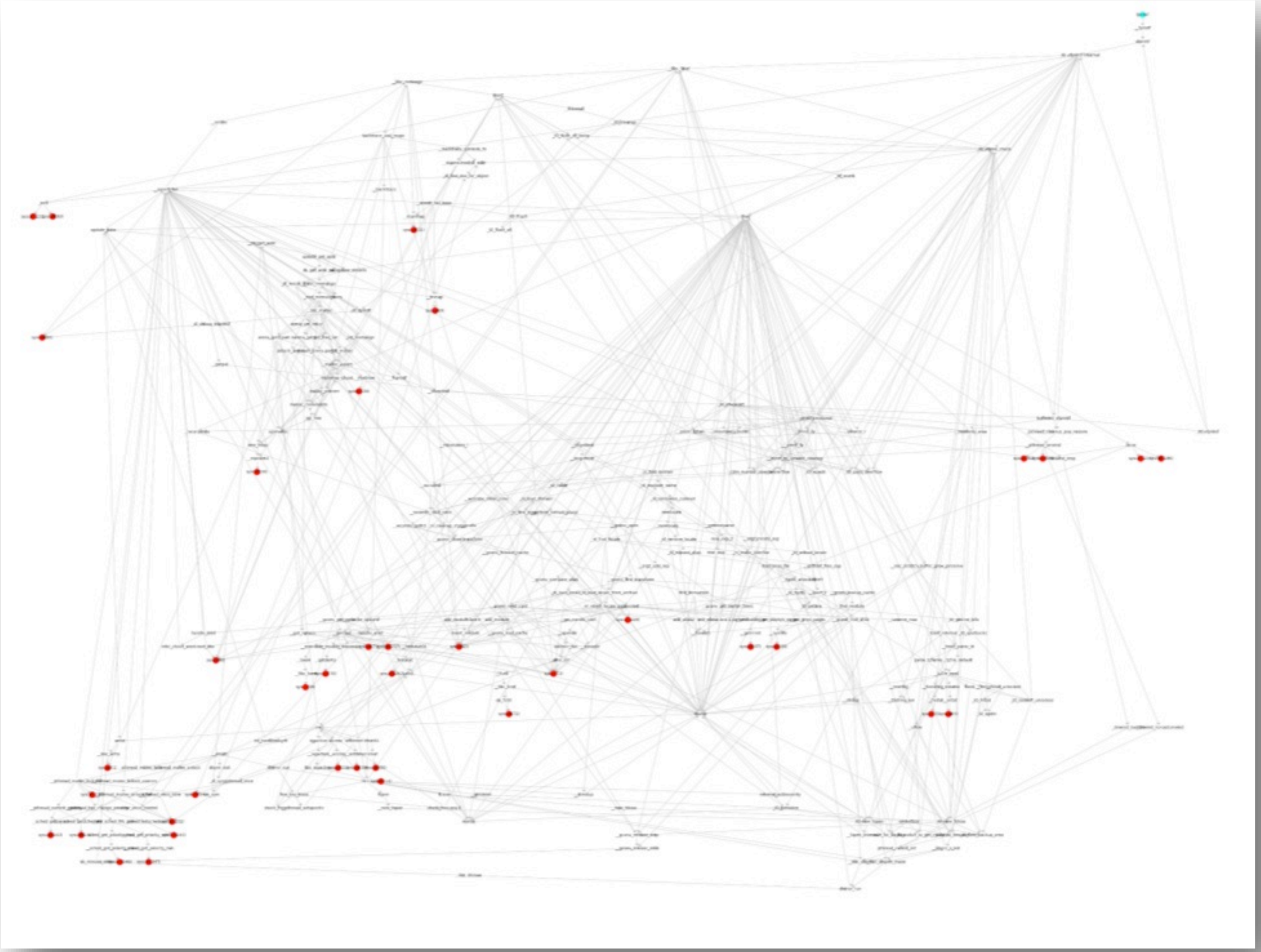
Conclusion

- System call filtering mechanism is important for protection of container environment
  - Attack surface of shared kernel reduction
  - Huge damage caused by container escape
- The current filtering mechanism is fragile and non-scalable solution
  - 67 out of 106 malicious codes (about 63%) were able to be blocked
- System call sequence blocking can compensate for the loopholes in individual system call blocking

Filtering Mechanism	# of Policy	# of exploit codes that can be mitigated (%)
Individual Syscall-based	42	67 (63%)
Syscall Sequence-based	33	104 (98%)

Thank you

Appendix. fprintf glibc library function call graph



in Confine [4]

Appendix. Docker Seccomp Porfile

- Docker uses the Seccomp kernel feature to block container access to 51 system calls
  - Docker's Seccomp profile[2] allows 346 system calls

acct	iopl	personality	swapon
add_key	kcmp	pivot_root	swapoff
bpf	kexec_file_load	process_vm_readv	sysfs
clock_adjtime	kexec_load	process_vm_writev	_sysctl
clock_settime	keyctl	ptrace	umount
clone	lookup_dcookie	query_module	umount2
create_module	mbind	quotactl	unshare
delete_module	mount	reboot	uselib
finit_module	move_pages	request_key	userfaultfd
get_kernel_syms	name_to_handle_at	set_mempolicy	ustat
get_mempolicy	nfsservctl	setns	vm86
init_module	open_by_handle_at	settimeofday	vm86old
ioperm	perf_event_open	stime	

Reference

1. https://financesonline.com/cloud-computing-statistics/.
2. “Seccomp profiles for Docker,” https://docs.docker.com/engine/security/ seccomp/.
3. Ramakrishnan Srikant et al. 1996. “Mining Sequential Patterns: Generalizations and Performance Improvements”. In Proceedings of the 5th International Conference on Extending Database Technology: Advances in Database Technology (EDBT '96). Springer-Verlag, Berlin, Heidelberg, 3–17.
4. Seyedhamed Ghavamnia, Tapti Palit, Azzedine Benameur, and Michalis Polychronakis. 2020. Confine: Automated System Call Policy Generation for Container Attack Surface Reduction. In 23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020).
5. Nicholas DeMarinis, Kent Williams-King, Di Jin, Rodrigo Fonseca, and Vasileios P. Kemerlis. 2020. sysfilter: Automated System Call Filtering for Commodity Software. In 23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020).
6. Muhammad Abubakar, Adil Ahmad, Pedro Fonseca, and Dongyan Xu. 2021. SHARD: Fine-Grained Kernel Specialization with Context-Aware Hardening. In USENIX Security Symposium.
7. Olufogorehan Tunde-Onadele, Yuhang Lin, Jingzhu He, and Xiaohui Gu. 2020. Self-Patch: Beyond Patch Tuesday for Containerized Applications. In IEEE ACSOS.
8. Nuno Lopes, Rolando Martins, Manuel Eduardo Correia, Sérgio Serrano, and Francisco Nunes. 2020. Container Hardening Through Automated Seccomp Profiling. In Proceedings of the 2020 6th International Workshop on Container Technologies and Container Clouds.
9. Wan, Zhiyuan, et al. "Mining sandboxes for linux containers." 2017 IEEE International Conference on Software Testing, Verification and Validation (ICST).
10. L. Lei et al., “SPEAKER: Split-Phase Execution of Application Containers,” in Detection of Intrusions and Malware, and Vulnerability Assessm ent.
11. API Sanity Checker. An automatic generator ofbasic unit tests for a shared C/C++ library. https://lvc.github.io/api-sanity-checker/.