Research on System Call Filtering Technique for Defense against Host Kernel Exploits in Container Environment

Somin Song (Advisor: Byungchul Tak)

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

July 13rd, 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 <u>within the next five years</u>

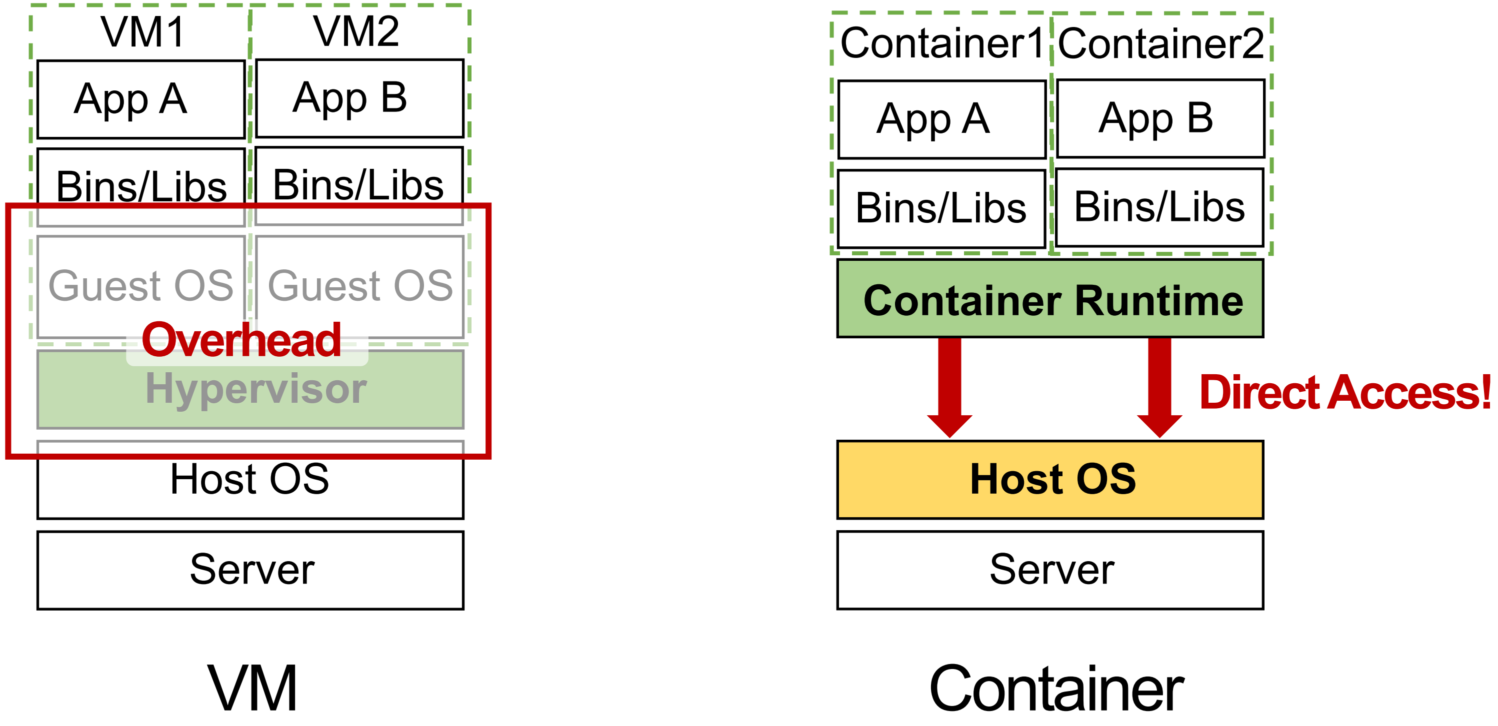
– LogicMonitor, 2020 [1]

- **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

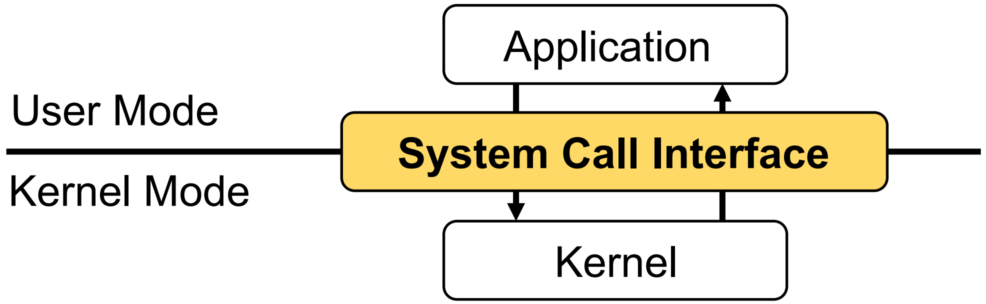
- Containers are a **key virtualization technology** in cloud computing



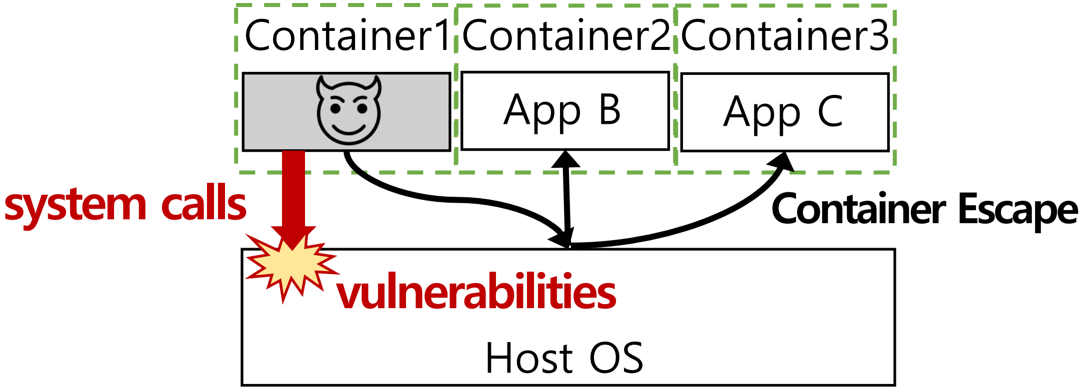
Containers share a Host OS!

Container Escape through System Call

- System Call



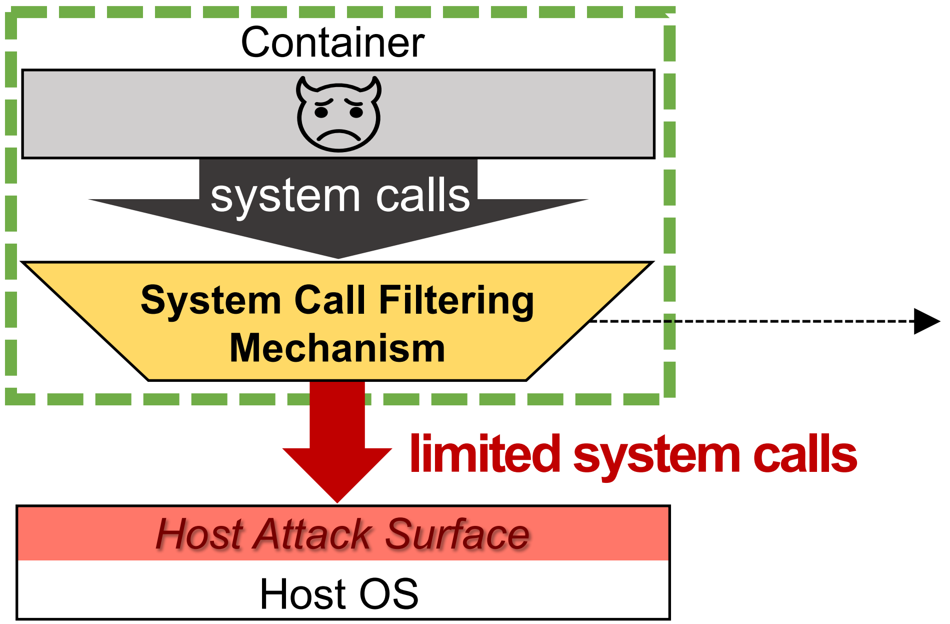
- Container Escape
  - Exploitation of Host kernel vulnerabilities through carefully **crafted system calls**



- 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)

System Call Filtering Protection Mechanism

The fewer system calls, the more secure!

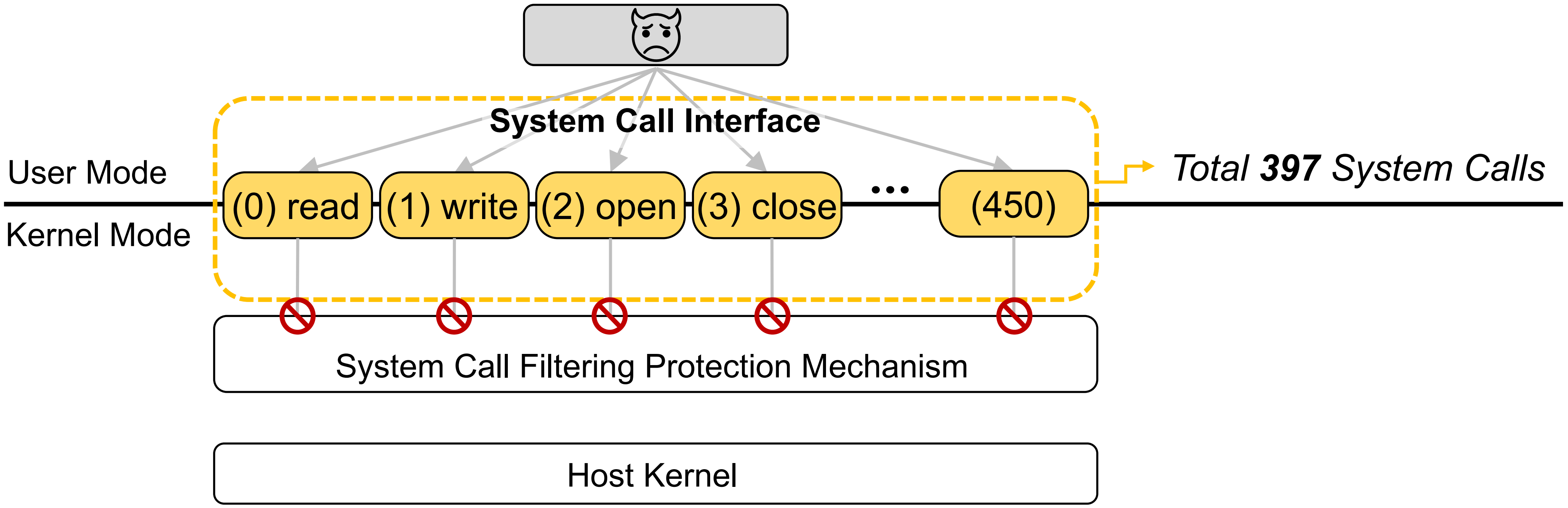


Seccomp (Secure Computing mode)

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

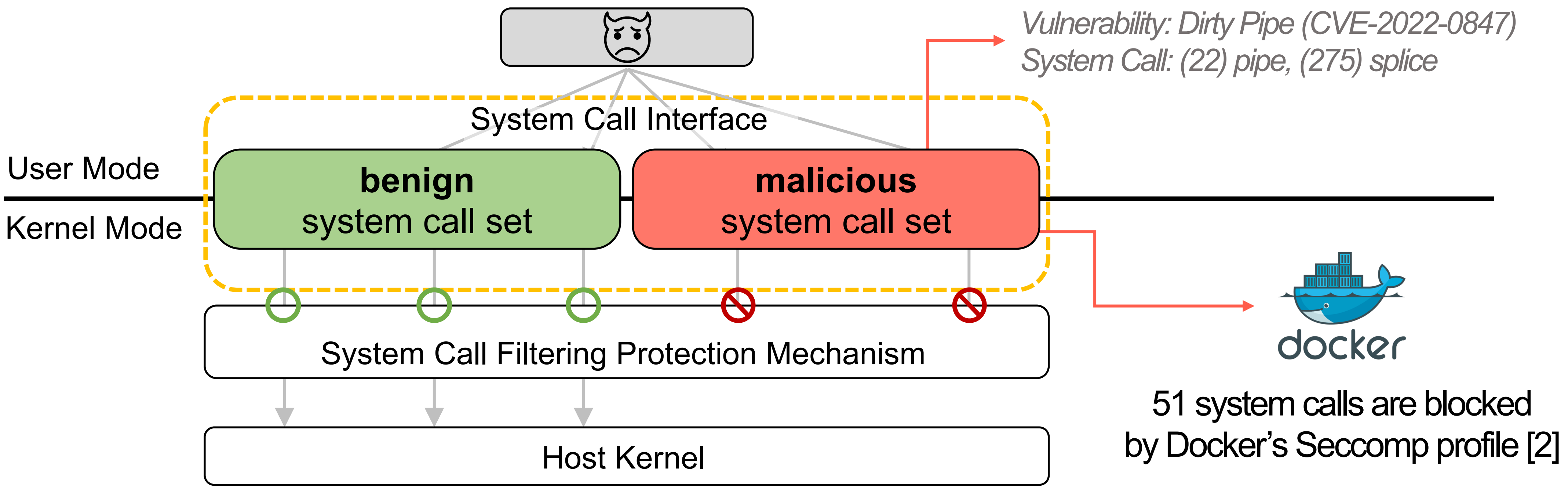
System Call Filtering Protection Mechanism

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



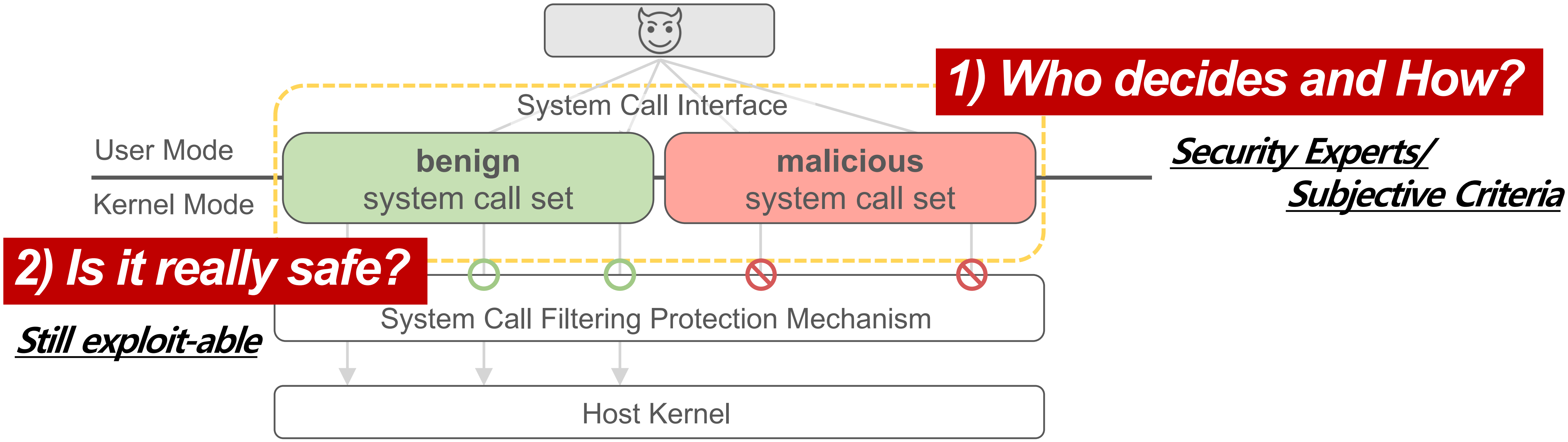
System Call Filtering Protection Mechanism

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



Problems

- "Which System Call should be allowed/blocked?"
  - Which System Call is benign/malicious?



Two Novel Approaches for System Call Filtering

- *"Research on System Call Filtering Technique for Defense against Host Kernel Exploits in Container Environment"*
  - Goal: To improve the security capability of the system call filtering

Problem 1) Who decides and How?

- Limitation of Problem 1
  - Manual analysis by experts through subjective criteria
- Approach 1
  - Risk auto-assessment of system calls using objective criteria

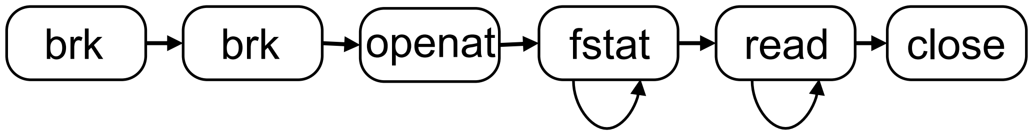
Top Rank capset : 0.49551

Last Rank close : 0.026151

Problem 2) is it really safe?

- Limitation of Problem 2
  - Exploitable by combining allowed system calls
- Approach 2
  - Exploitable system call combination investigation

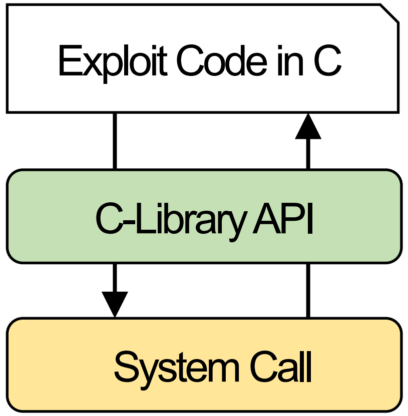
System Call Sequence



Used in 22 Exploit Codes

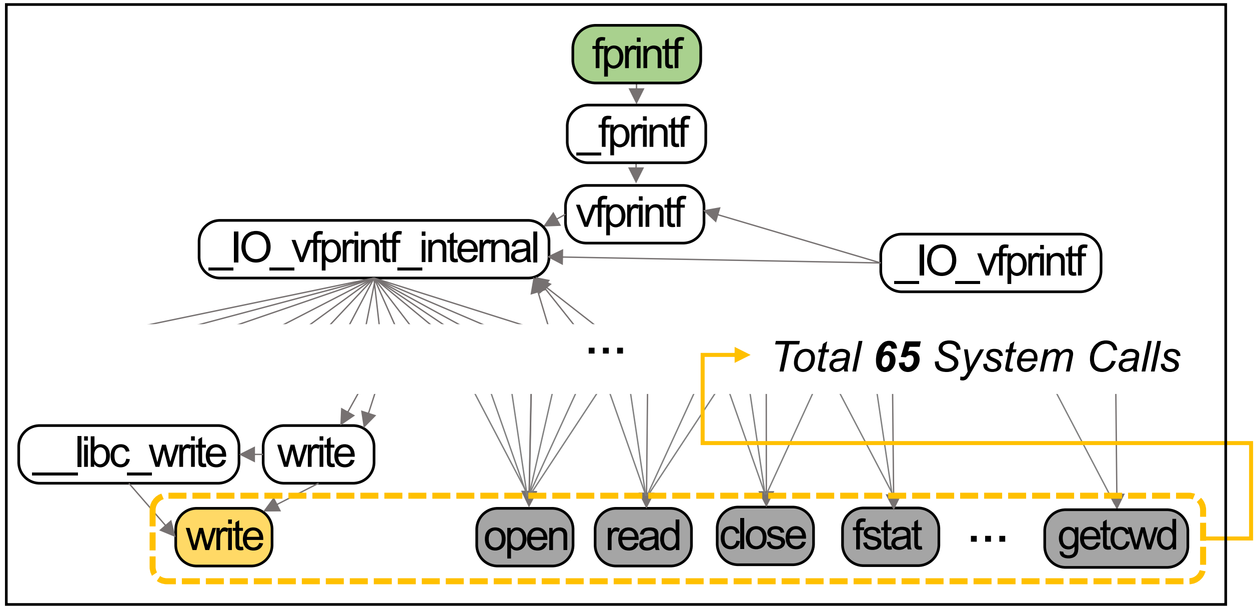
Key Exploit Code Analysis Methodologies

- Goal: To investigate the system calls that the exploit codes invoke
  - System Call <u>Set</u> / System Call <u>Sequence</u>
- Only Dynamic Analysis
  - Method of tracing system calls while directly executing the exploit codes
  - \+ Upon successful execution, complete information can be obtained
  - – Exploit code is system-dependent
    - ► It is very difficult to set up the environment to run the exploit code
    - ► The Linux kernel has thousands of version histories
- Only Static Analysis
  - Method of analyzing the source code without executing the exploit codes
  - \+ Automated Large-scale analysis is possible
    - ► No configuration is required for the code execution
  - – C language makes static analysis difficult
    - ► Dynamic linking/allocation and pointers



Problems in Precedent Researches

- Challenge in **Static Analysis**
  - Confine [3] (RAID '20) : built libc-to-syscall call graph



► ***missing/boating system calls***

  - Nicholas et al. [4], Abubakar et al. [5], and Olufogorehan et al. [6] : generated syscall lists

► ***Do not attempt to mine for system call sequences***

- Challenge in **Dynamic Analysis**
  - Lopes et al. [7] : used unit testing combined with fuzzing by running the target application
  - Wan et al. [8] : created custom profiles with pre-defined test suites
  - Speaker [9] : traced system call during a container execution

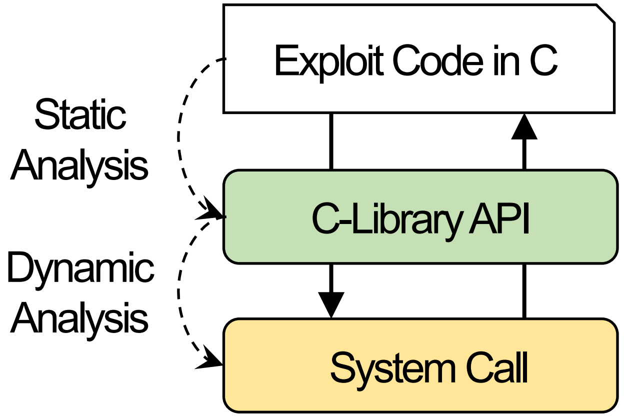
► ***users must have a comprehensive understanding of the applications' behavior***

► ***Not sound to find all corner cases***

► ***Do not attempt to mine for system call sequences***

Key Exploit Code Analysis Methodologies

- Hybrid Analysis
  - Combine static analysis and dynamic analysis to generate a system call set/sequence corresponding to each exploit code
  - **Static Analysis**: to extract library function set/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 set/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

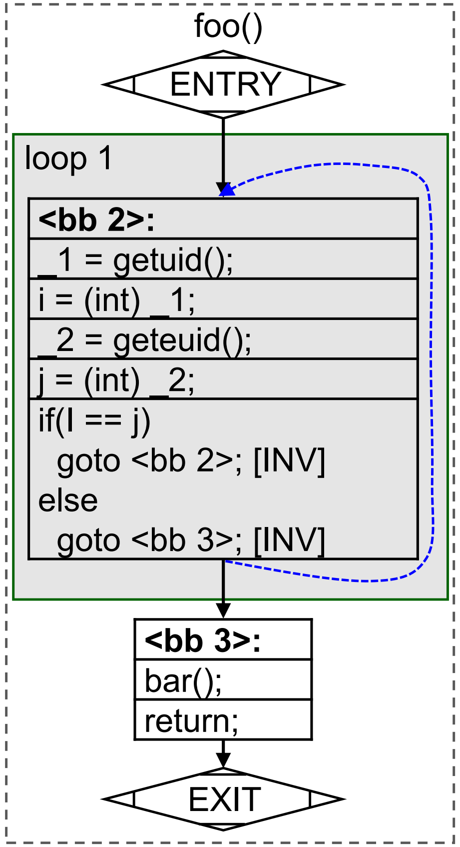
foo: getuid-geteuid-getuid-geteuid-getuid-geteuid-bar

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 [10] + ftrace mechanism)

(a) API Sanity Checker Dataset

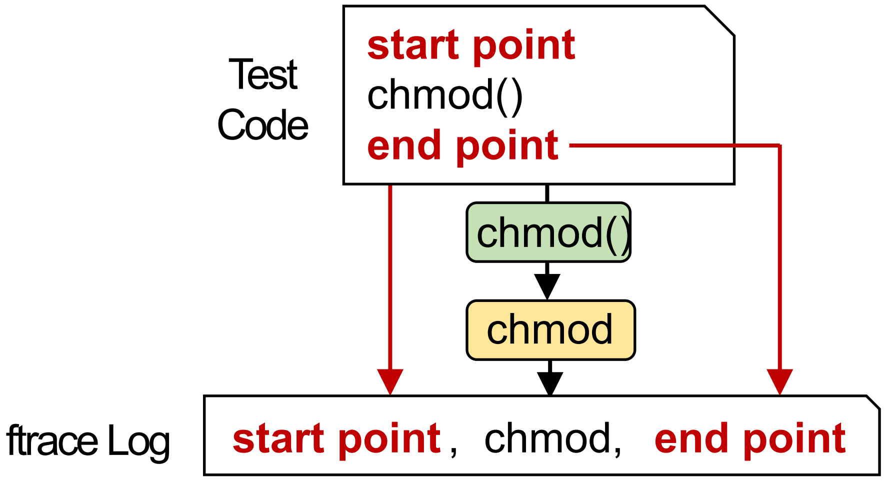
- An automatic generator of basic unit tests for a shared C/C++ library

(b) ftrace mechanism

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

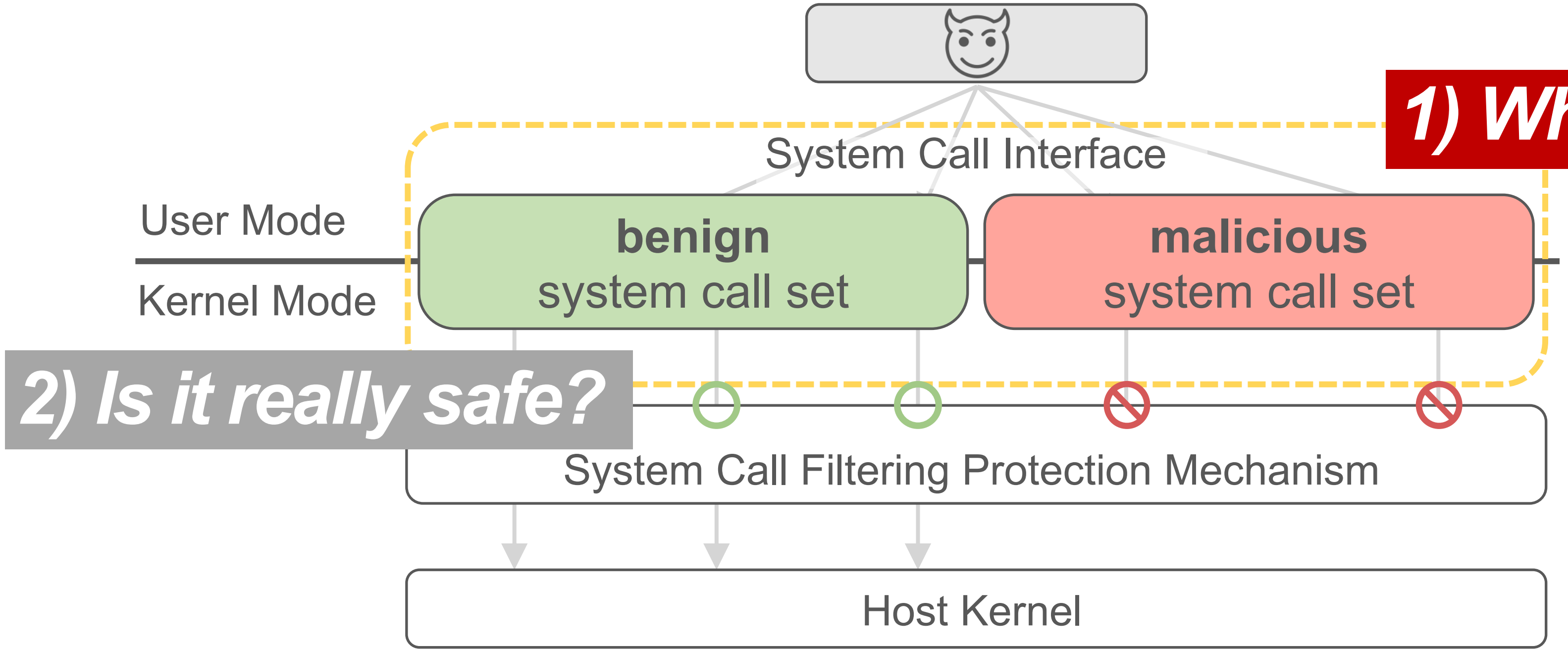
```
#include <rpc/types.h>
#include <sys/stat.h>>

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



Research on Problem 1

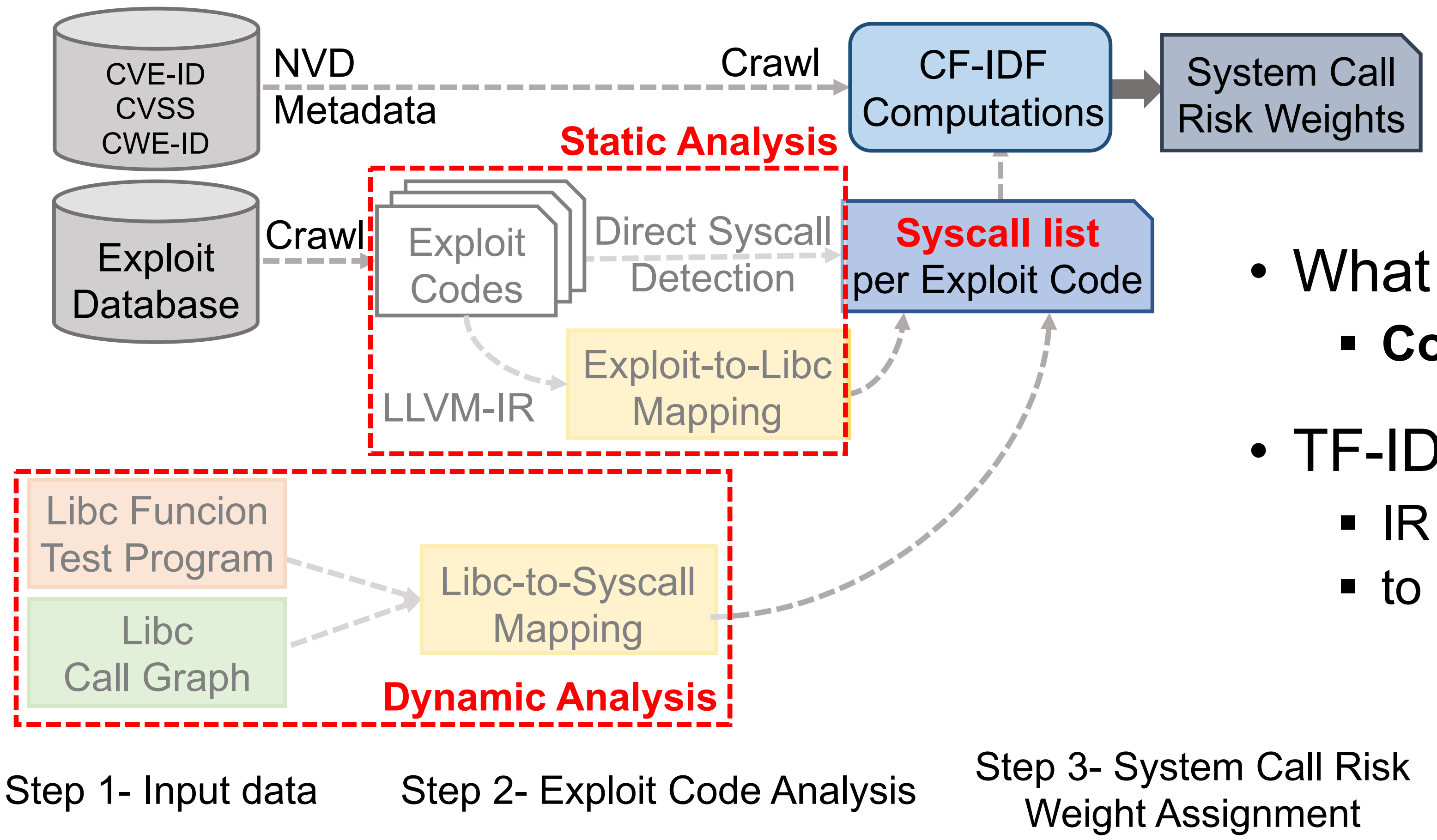
- "Which System Call should be allowed/blocked?"
  - Which System Call is benign/malicious?



Risk auto-assessment of syscalls using objective criteria

Overview of Research on System Call Auto-Assessment of Risk

- Goal: System call risk quantification



- What is a "***Risky System Call***" ?
  - **Core role** in attack logic
- TF-IDF variant = "**CF-IDF**"
  - IR technique
  - to extract *keywords* from *documents*

|| ||
---|---
key syscalls | ***exploit codes***

in SecQunat: Quantifying Container System Call Exposure, ESORICS '22

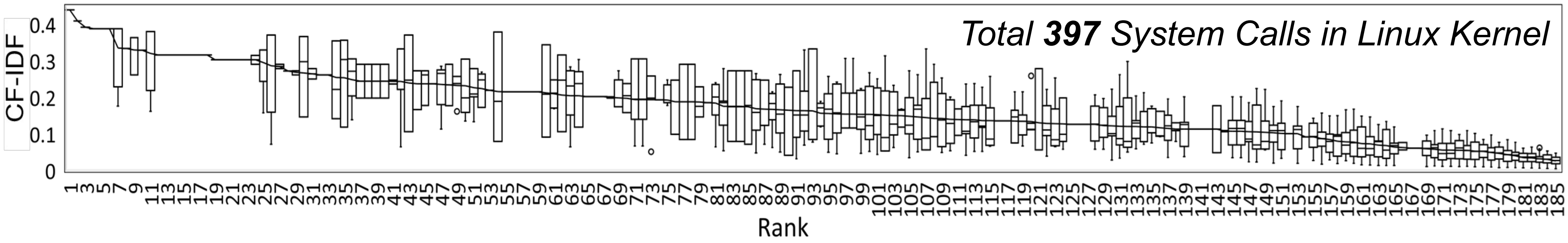
System Call Risk Assessment Result

- The risk of 185 system calls was automatically quantified through objective metrics from 298 exploits
- Possible to filter reflecting the latest vulnerability status

CVSS of Vulnerability is **10.0** *(highest vulnerability risk score)*

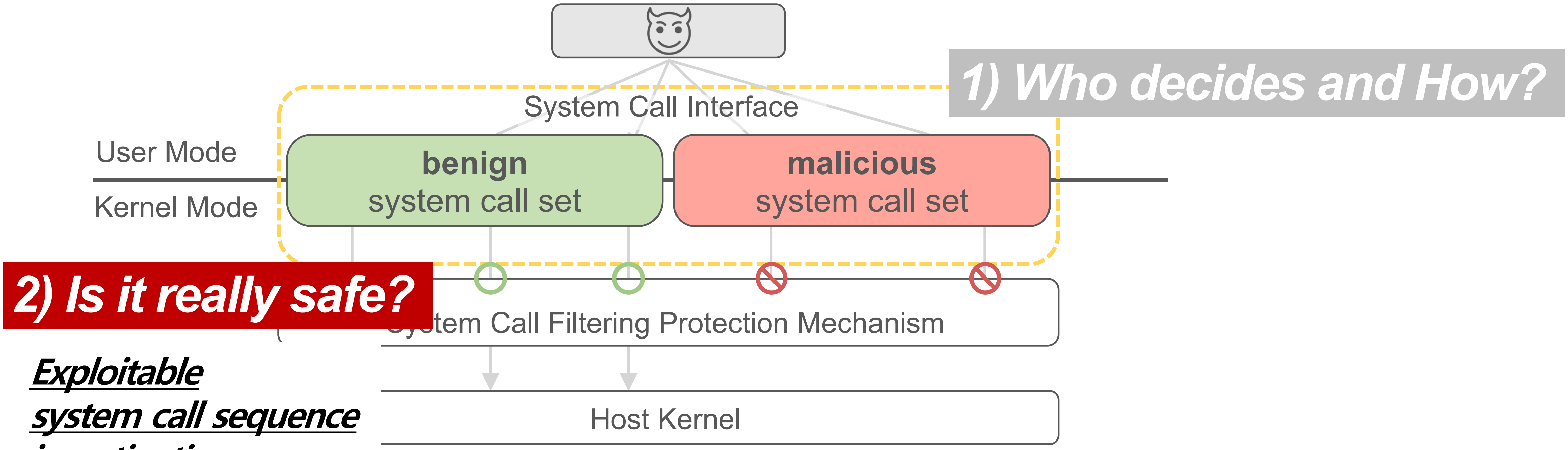
Rank	Syscall	Weight	Rank	Syscall	Weight	Rank	Syscall	Weight	Rank	Syscall	Weight
1	**capset**	**0.439551**	4	io_uring_reg	0.388023	11	keyctl	0.32028	12	io_setup	0.316477
2	add_key	0.4094431	7	shutdown	0.335366	12	fchown	0.316477	12	io_submit	0.316477
3	recvmmsg	0.392371	8	settimeofday	0.334059	12	flock	0.316477	12	kcmp	0.316477
4	getresuid	0.388023	9	rename	0.329819	12	mknod	0.316477	…	…	…
4	sendfile	0.388023	10	creat	0.329663	12	mq_notify	0.316477	**185**	**close**	**0.026151**

Appears in **192** out of 298 exploits (about **64%**)



Research on Problem 2

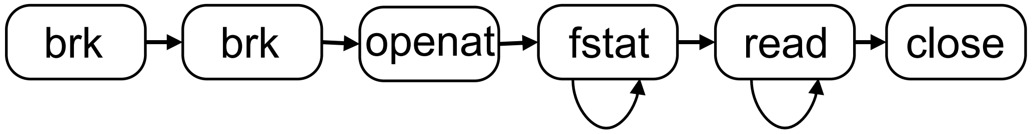
- "Which System Call should be allowed/blocked?"
  - Which System Call is benign/malicious?



<u>Exploitable system call sequence investigation</u>

Motivation of Malicious System Call Sequence Research

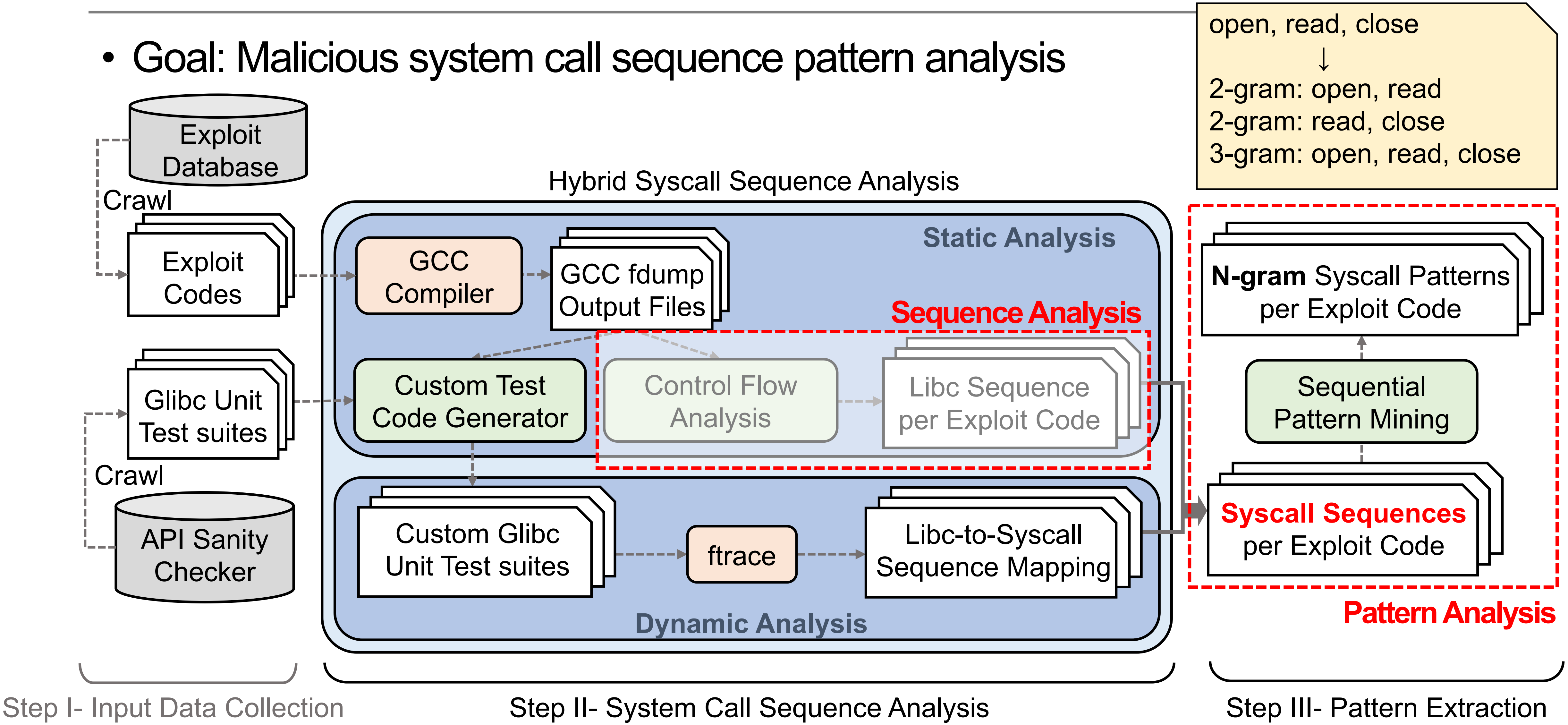
- A malicious sequence consisting of only allowed system calls



- Motivation: Limitation of Docker's Seccomp Profile
- Docker CAN NOT block **55%** of 106 exploits
  - The 47 host kernel vulnerabilities CAN be exploited using only system calls allowed by Docker's Seccomp profile
- The proposed mechanism CAN block **70%** of exploits that Docker cannot block
- Towards system call sequence filtering for enhanced container security
  - *"Can we find patterns in the malicious system call sequences?"*

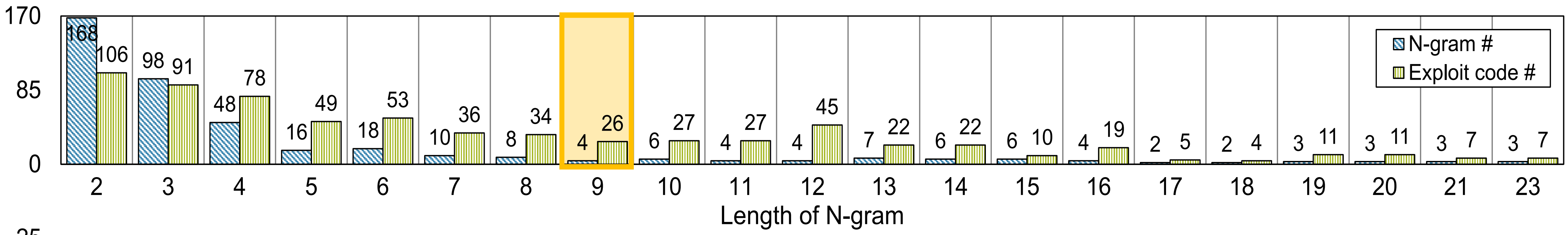
Overview of Malicious System Call Sequence Pattern Analysis

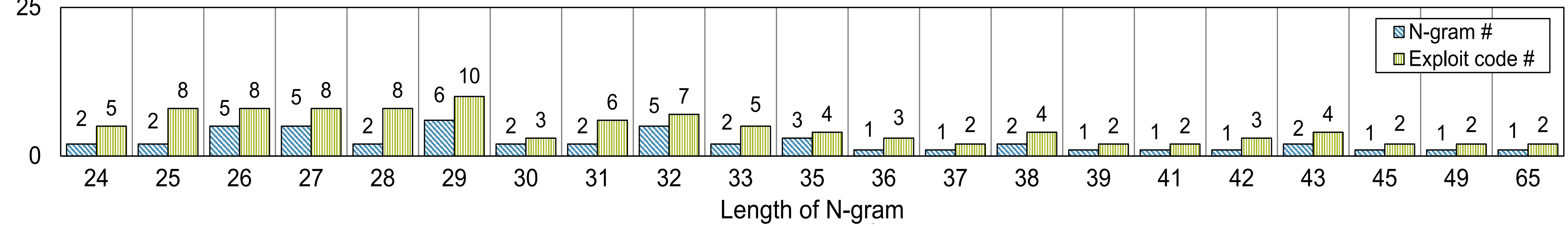
- Goal: Malicious system call sequence pattern analysis



Malicious System Call Sequence Pattern Result

- The 471 system call sequence patterns are found from 106 exploits





9-gram #	9-gram pattern	Exploit #	Total Exploits
4	$(\text{openat,stat,clock\_nanosleep})^3$	3	26
	$(\text{brk,brk,openat})^2$,fstat,read,fstat	7	
	$(\text{fstat,read})^4$,close	18	
	brk,brk,openat,$(\text{fstat,read})^3$	22	

Effectiveness of Sequence-based Filtering Mechanism

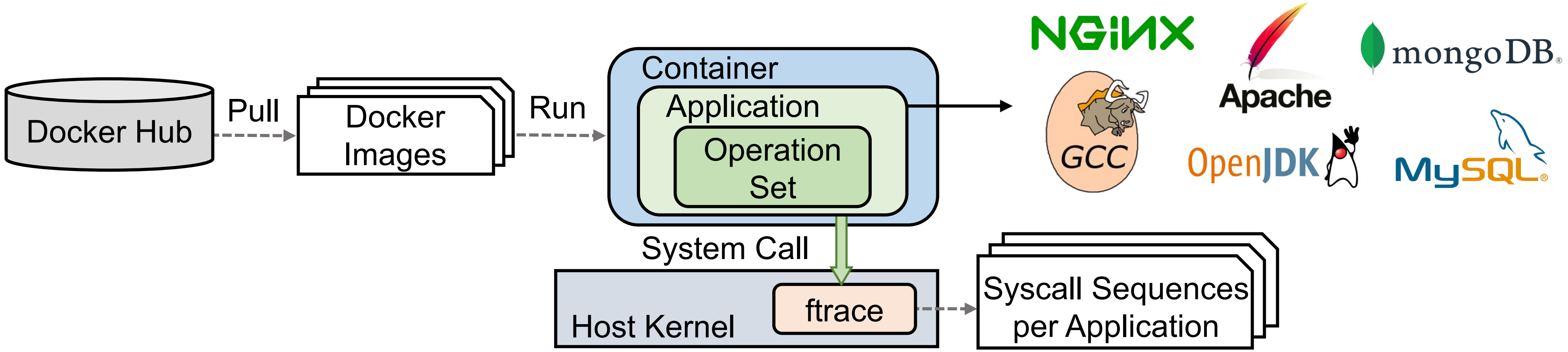
- *What's the effect of pattern lengths on identifying malicious system call sequences?*

Rank	Length	N-gram pattern	Exploit percentage
1	2	brk, brk	52%
2	2	openat, fstat	38%
3	2	read, close	36%
4	2	rt_sigaction, rt_sigprocmask	35%
4	2	read, fstat	33%

- Top patterns that appear in the most number of exploits
  - provide high coverage when employed as filtering candidates against exploits
  - likely interrupt benign application operations
- Condition for better defense ability of system call sequence pattern
  - **Appearing in multiple exploits**
  - **Longer**

Investigation of Patterns Falsely Disrupting Benign Application

- *To what degree do patterns falsely disrupt benign application executions?*



- We have collected per-thread system call traces from 15 popular applications
  - ► Performing diverse application-specific operation
    - → e.g., {GET, PUT, POST} for web servers
    - → e.g., {INSERT, DELETE, SELECT} for NoSQL databases, amongst others

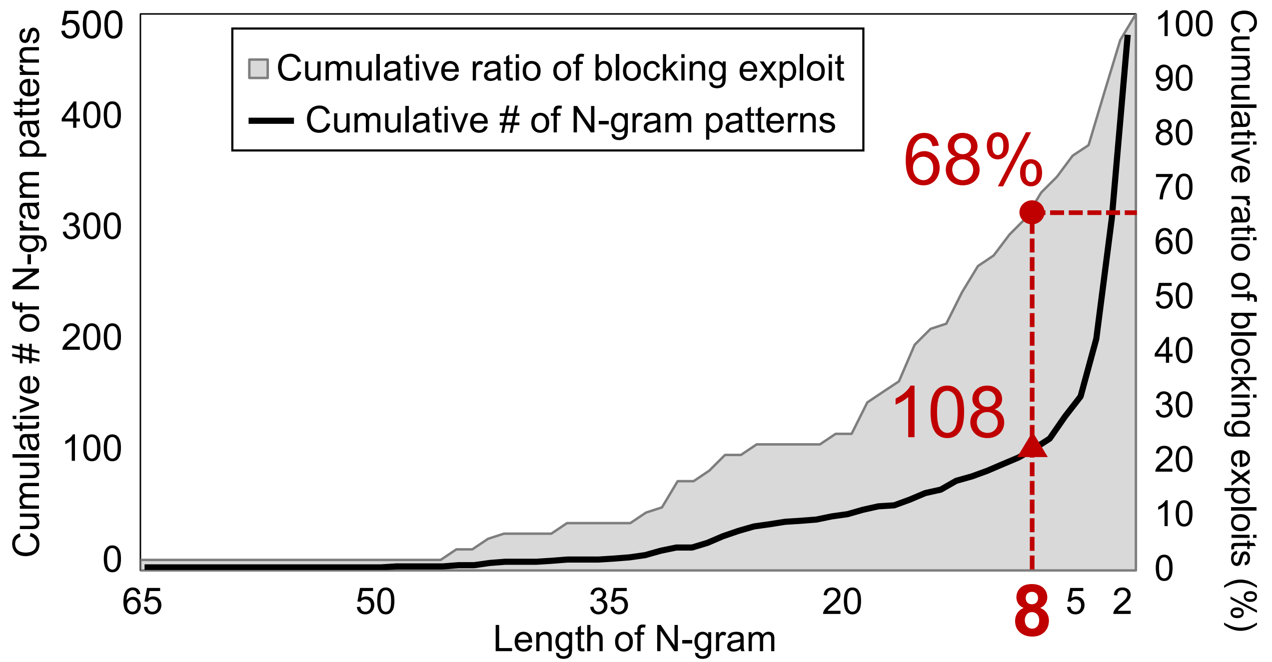
Comparison of Malicious Patterns and Normal Applications

- There exist certain length of N-grams on the low side above which do not disrupt normal applications

Application Name	# of N-grams from exploit codes found in application trace (%)									
	2-gram	3-gram	4-gram	5-gram	6-gram	7-gram	8-gram	…	65-gram	**Total**
nginx	3 (1.8%)	1 (1%)	–	–	–	–	–	…	–	4 (0.8%)
httpd	12 (7.1%)	2 (2%)	–	–	–	–	–	…	–	14 (3%)
tomcat	7 (4.2%)	3 (3.1%)	–	–	–	–	–	…	–	10 (2.1%)
node	4 (2.4%)	2 (2%)	–	–	–	–	–	…	–	6 (1.3%)
mongodb	19 (11.3%)	6 (6.1%)	1 (2.1%)	–	–	–	–	…	–	26 (5.5%)
mysql	22 (13.1%)	11 (11.2%)	5 (10.4%)	2 (12.5%)	–	–	–	…	–	40 (8.5%)
mariadb	11 (6.5%)	2 (2%)	–	–	–	–	–	…	–	13 (2.8%)
redis	9 (5.4%)	3 (3.1%)	–	–	1 (5.6%)	1 (10%)	–	…	–	14 (3%)
gcc	34 (20.2%)	16 (16.3%)	5 (10.4%)	3 (18.8%)	1 (5.6%)	–	–	…	–	59 (12.5%)
openjdk	50 (29.8%)	18 (18.4%)	5 (10.4%)	3 (18.8%)	2 (11.1%)	–	–	…	–	78 (16.6%)
gzip	9 (5.4%)	13 (13.3%)	4 (8.3%)	–	1 (5.6%)	–	–	…	–	27 (5.7%)
bzip2	9 (5.4%)	8 (8.2%)	3 (6.3%)	1 (6.3%)	1 (5.6%)	–	–	…	–	22 (4.7%)
qalc	48 (28.6%)	14 (14.3%)	5 (10.4%)	3 (18.8%)	2 (11.1%)	–	–	…	–	72 (15.3%)
ghostscript	36 (21.4%)	21 (21.4%)	6 (12.5%)	2 (12.5%)	1 (5.6%)	–	–	…	–	66 (14%)
lowriter	71 (42.3%)	28 (28.6%)	7 (14.6%)	4 (25%)	2 (11.1%)	–	–	…	–	112 (23.8%)
Total	82 (48.8%)	45 (45.9%)	9 (18.8%)	4 (25%)	3 (16.7%)	1 (10%)	–	…	–	144 (30.6%)
# of N-grams from exploits	168	98	48	16	18	10	8	…	1	471
Proportion of blocked apps	100%	100%	60%	46.6%	53.3%	6%	–	–	–	–

Short Sequence is Good Enough!

- Blocking capability and the ratio that blocks benign applications are in trade-off relationship
  - From the application set we experimented with, the system call sequence length of 8 and above has been shown to be free from disruptive side-effects
  - The use of all the N-grams starting from length 8 and above up to 65-gram block **68%** of 108 exploit codes we tested



All of attacks that Docker Seccomp CANNOT prevent can block!

- Docker Seccomp CAN NOT protect against **58** *(55%)* out of the 106 exploits
- Patterns with a length of 8 or more can mitigate **40** out of the 58 exploits (about **70%** Coverage)
- For the remaining 18 (30%) exploits that share only short patterns (lengths 1 through 7), it is still possible to mitigate them by using exploit-specific system call sequences that are not caught as patterns
  - Candidate with length of 13 for CVE-2022-0847 exploit:

    < openat, fstat, pipe, fcntl, $write^3$, $read^3$, splice, write, close >

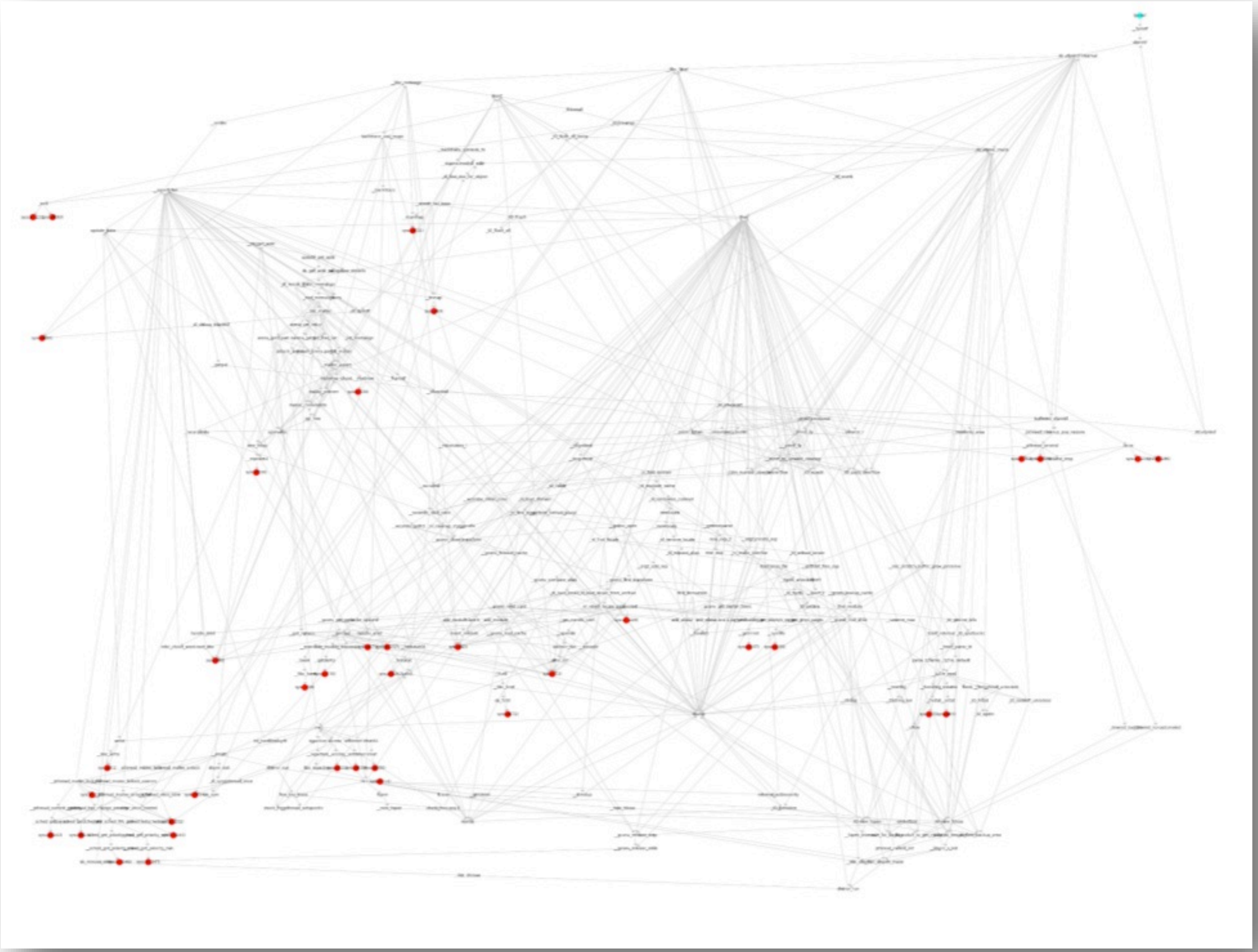
Conclusion

- System call filtering protection 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
- The challenge to select an appropriate system calls to block
- System call risk reflecting the latest vulnerability status can be a guide for system call policies
- System call sequence blocking can compensate for the loopholes in individual system call blocking
- *Prevention of software disasters is essential as well as recovery*

Reference

1. https://financesonline.com/cloud-computing-statistics/.
2. “Seccomp profiles for Docker,” https://docs.docker.com/engine/security/ seccomp/.
3. 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).
4. 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).
5. Muhammad Abubakar, Adil Ahmad, Pedro Fonseca, and Dongyan Xu. 2021. SHARD: Fine-Grained Kernel Specialization with Context-Aware Hardening. In USENIX Security Symposium.
6. Olufogorehan Tunde-Onadele, Yuhang Lin, Jingzhu He, and Xiaohui Gu. 2020. Self-Patch: Beyond Patch Tuesday for Containerized Applications. In IEEE ACSOS.
7. 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.
8. Wan, Zhiyuan, et al. "Mining sandboxes for linux containers." 2017 IEEE International Conference on Software Testing, Verification and Validation (ICST).
9. L. Lei et al., “SPEAKER: Split-Phase Execution of Application Containers,” in Detection of Intrusions and Malware, and Vulnerability Assessm ent.
10. API Sanity Checker. An automatic generator ofbasic unit tests for a shared C/C++ library. https://lvc.github.io/api-sanity-checker/.

Appendix. fprintf glibc library function call graph



in Confine [3]

Static Analysis to Map Exploit Code-Glibc Function

- Using Clang + LLVM-IR
  - Getting info on invocation of glibc function and direct system call

```
inline _syscall1(int, close, int, fd); ① direct syscall(inline)

void alloc_victim(void) ② user-defined function
{
    fd = open(FILE_PATH, O_WRONLY, mode); ③ library function
  ④ close(fd);   invocation of ①
    __asm__(
            "pusha\n"
            "movl %1, %%eax\n"
            "movl $("xstr(CLONEFL)"), %%ebx\n"
            "movl %%esp, %%ecx\n"
            "movl $120, %%eax\n"
            "int  $0x80\n"        ⑤ direct syscall (asm)
            "movl %%eax, %0\n"
            "popa\n"
            : : "m" (pid), "m" (dummy)
            );
    syscall(SYS_exit, 0); ⑥ direct syscall (with syscall())
}
```

Example source code

```
define dso_local void @alloc_victim() ② {
  %1 = load i32, i32* @mode, align 4
  %2 = call i32 (i8*, i32, ...) @open(i8* ③
   getelementptr inbounds ([14 x i8], [14 x i8]* @.str, i64
   0, i64 0), i32 1, i32 %1) #6
  ...
  %8 = call i32 @close(i32 %7) ④
  call void asm sideeffect "pusha\0Amovl $1, %eax\0Amovl
⑤ $$(CLONEFL), %ebx\0Amovl %esp, %ecx\0A
   movl $$120, %eax\0Aint $$0x80\0Amovl %eax, $0\0Apopa\0A",
   "*m,*m,~{dirflag},~{fpsr},~{flags}"(i32* @pid, i32*
   @dummy) #4, !srcloc !3
  %9 = call i64 (i64, ...) @syscall(i64 60, i32 0) #7 ⑥
  br label %10
10:                                           ; preds = %6, %5
  ret void
}
define dso_local i32 @close(i32 %0) #0 {
  ...
  %6 = call i64 asm sideeffect "syscall", "={ax},0,{di},
   ~{r11},~{rcx},~{memory},~{dirflag},~{fpsr},~{flags}“    ①
   (i64 3, i64 %5) #4, !srcloc !2
  ...
}
```

LLVM-IR format

Appendix. CF-IDF Dtails

- TF-IDF variant
  - Document = Exploit Code
  - Term = System Call
  - **IDF(Inverse Document Frequency)**

    $$= \frac{|Doc\ Set|}{Term\ f\ in\ Doc\ Set}$$

    ► TRUE: close, exit, nanosleep in most codes
  - TF(Term Frequency)

    $$= Term\ f\ in\ a\ Doc$$

    ► FALSE
  - **CF(Class Frequency)**

    $$= Doc\ f\ with\ Term\ in\ Class$$

    ► Class = Vulnerability

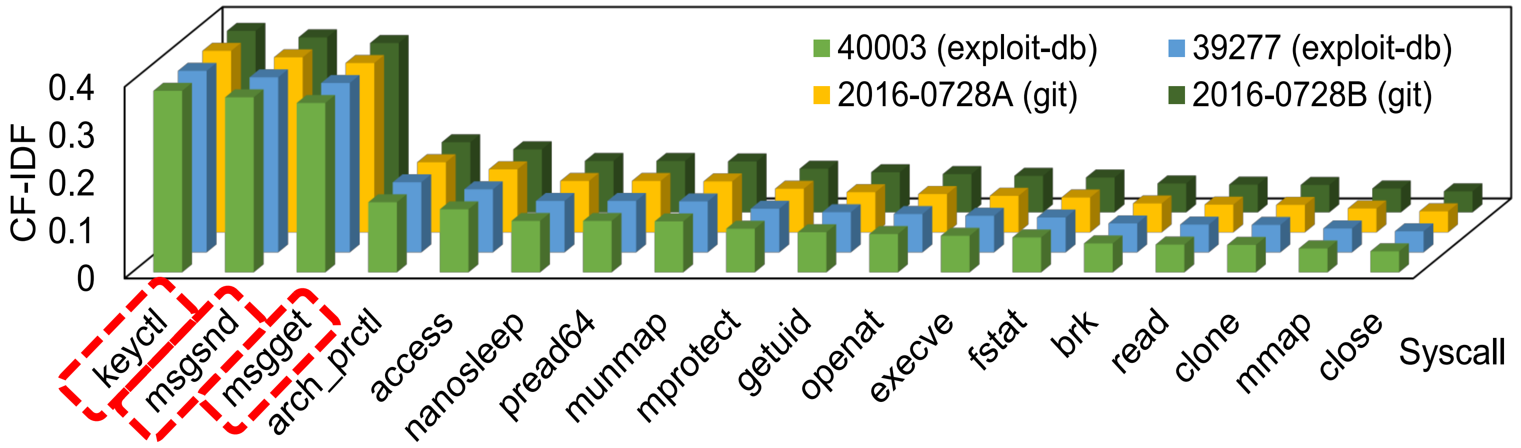
    ► pipe & splice always in Dirty Pipe
- *CF-IDF = CF X IDF*

Appendix. Full Ranked List of System Calls

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

Appendix. Verification of Our Risk Metric

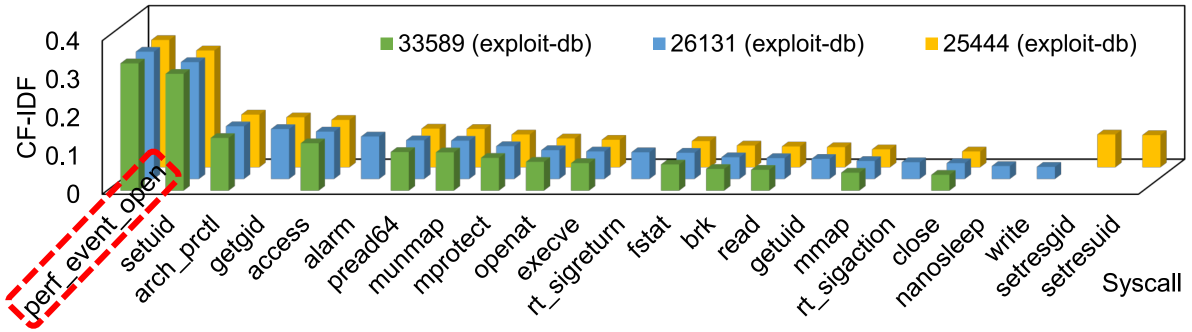
- Empirical justification supporting the soundness of the CF-IDF metric
  - No previous work that suggested the quantified risk of system calls
- CF-IDF risk vectors are signatures that represent the characteristic of exploit codes
  - It would generate similar signatures for the exploit codes that are truly similar in their nature

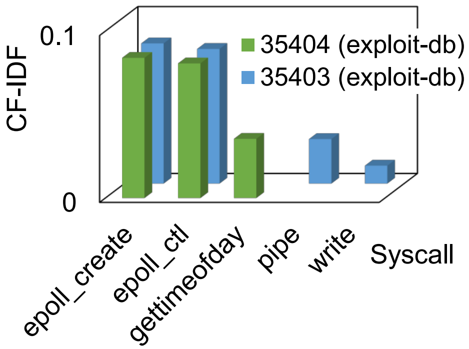


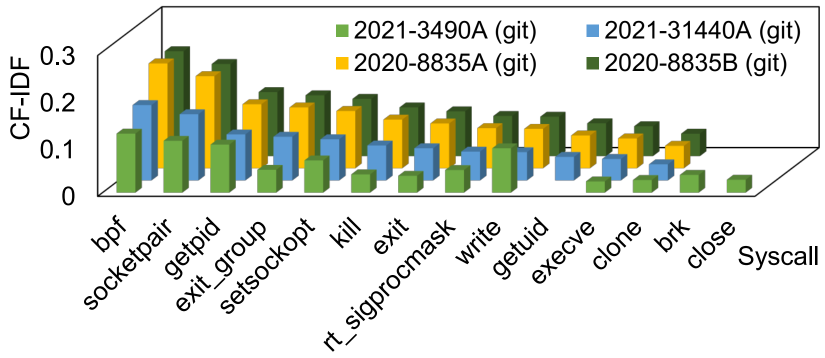
Exploit-ID (source)	CVE-ID	CVE Desctiption	Cosine Similarity
40003 (exploit-db)	CVE-2016-0728	Keyring object reference mishandling with crafted **keyctl** syscall	-
39277 (exploit-db)			1.0
2016-0728A (git)			1.0
2016-0728B (git)			1.0

Appendix. Additional Sample Groups Similarity Comparison

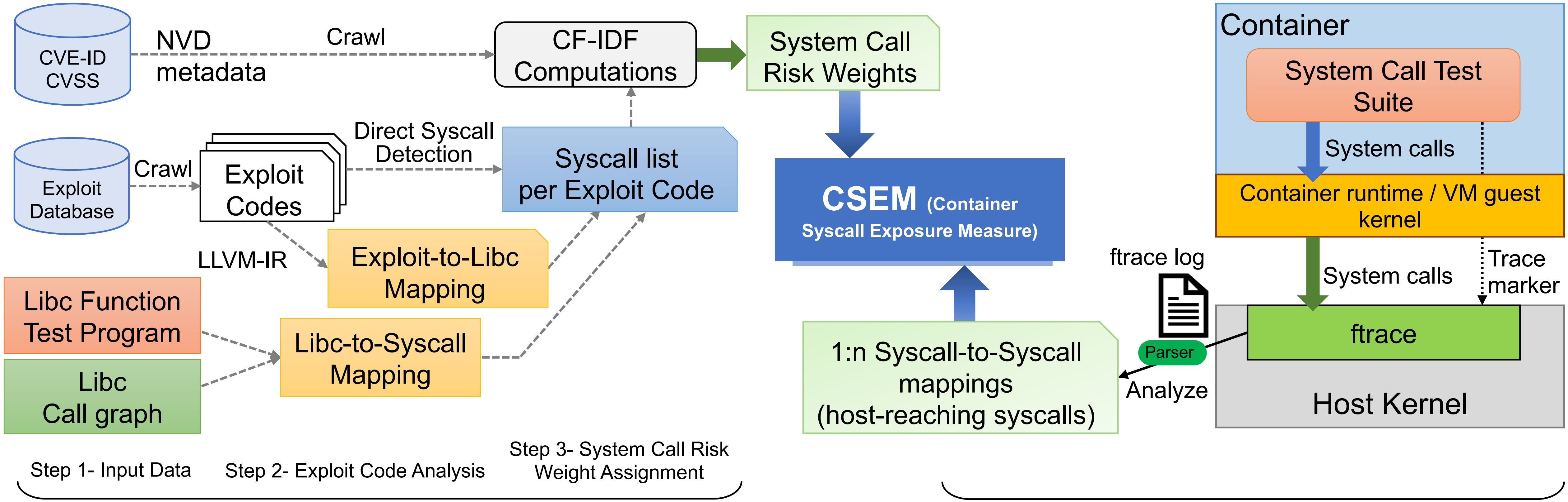
Group	Grouping Criteria	Exploit-ID (source)	CVE-ID	Similarity
G-I	Same CVE. Incorrect integer data type via crafted `perf_event_open`. PrivEsc.	33589 (exploit-db)	CVE-2013-2094	-
		26131 (exploit-db)		1.0
		25444 (exploit-db)		1.0
G-II	Different CVE but same vulnerability type. Improper traversal via crafted `epoll_create` and `epoll_ctl`. DoS.	35403 (exploit-db)	CVE-2011-1083	-
		35404 (exploit-db)	CVE-2011-1082	0.764533
G-III	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







Appendix. SecQuant Overview



(a) SCAR (System Call Assessment of Risk)

(b) SCED (System Call Exposure Discovery)

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	