CS 578: CYBER-SECURITY
PART II: OS SECURITY

Sanghyun Hong

sanghyun.hong@oregonstate.edu

Oregon State University

SAIL
Secure AI Systems Lab

COMPUTER SYSTEMS SECURITY

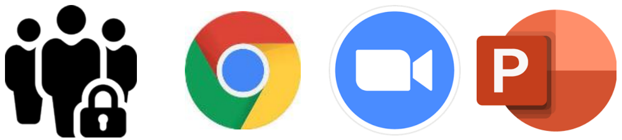
- What does an adversary want to do?
  - We learn
    - Buffer overflow
    - Heap overflow
    - Off-by-one
    - Use-after-free
  - Ok, after doing the buffer overflow, then what?
    - Subverting a system…
    - Get the root!

ATTACK SURFACE REDUCTION

PRELIMINARIES

- What is an operating systems?
  - Computer software that lies between hardware and applications

Humans Run Applications

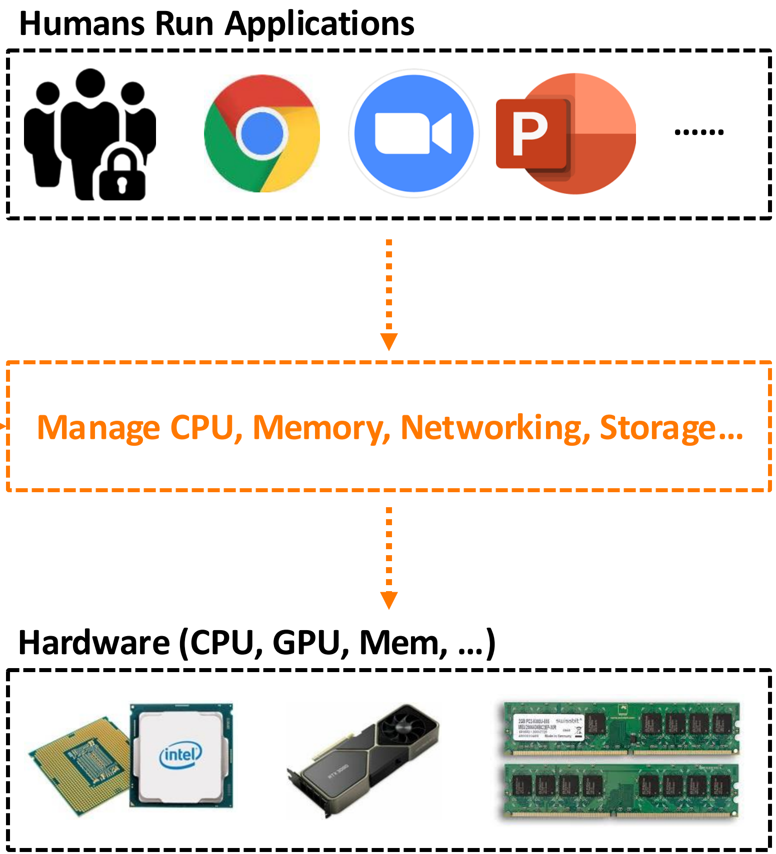


Operating System (OS)

Hardware (CPU, GPU, Mem, …)

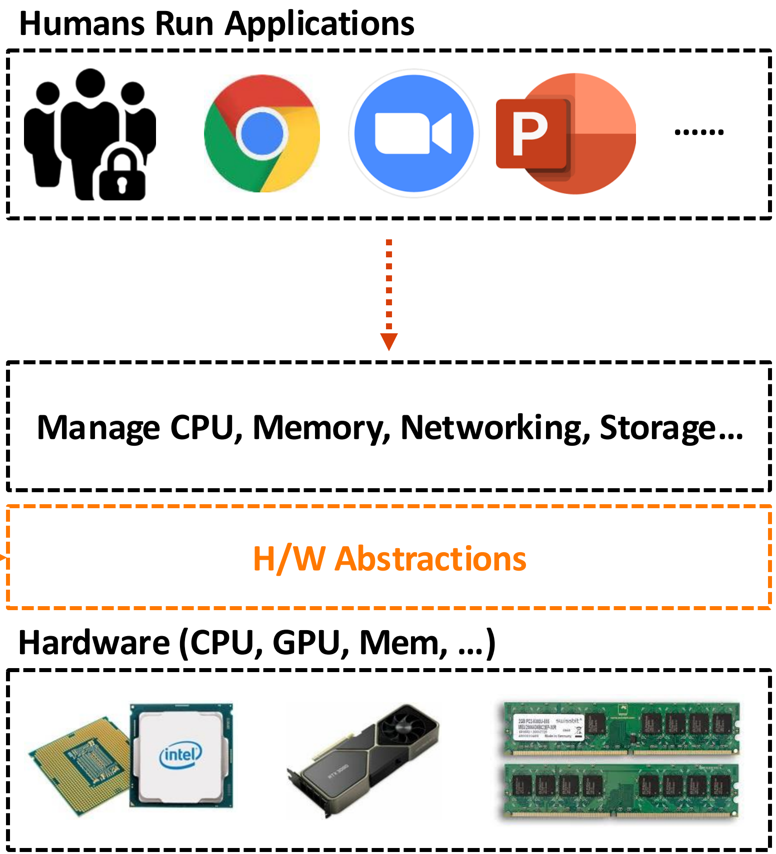
PRELIMINARIES – CONT'D

- What does it do?
  - **Manage resources**
  - Provide abstractions
  - Offer standard interfaces



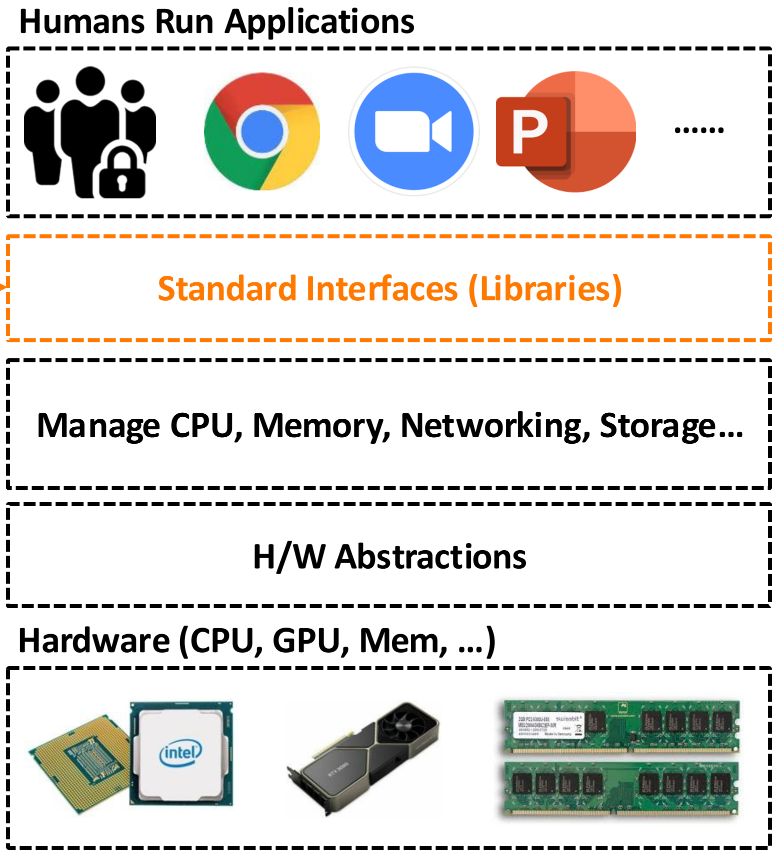
PRELIMINARIES – CONT'D

- What does it do?
  - Manage resources
  - **Provide abstractions**
  - Offer standard interfaces



PRELIMINARIES – CONT'D

- What does it do?
  - Manage resources
  - Provide abstractions
  - **Offer standard interfaces**



MODERN OPERATING SYSTEMS ARE COMPLEX AND LARGE

- Linux kernel supports
  - Managing many different hardware (e.g., memory, CPUs, GPUs, power system, …)
  - Many different interface to communicate and control hardware (e.g., device drivers, IOCTL)
  - Many different software libraries (e.g., OpenSSL, GlibC, …)

- The complexity may introduce potential vulnerabilities
  - Different developers write kernel device drivers, core functionalities, and so on

MODERN OPERATING SYSTEMS ARE COMPLEX AND LARGE – CONT'D

- How to reduce potential vulnerabilities?
  - Key intuition: attack surface reduction
  - CVE-2013-2094: *_perf_event_open* – not used by any common applications

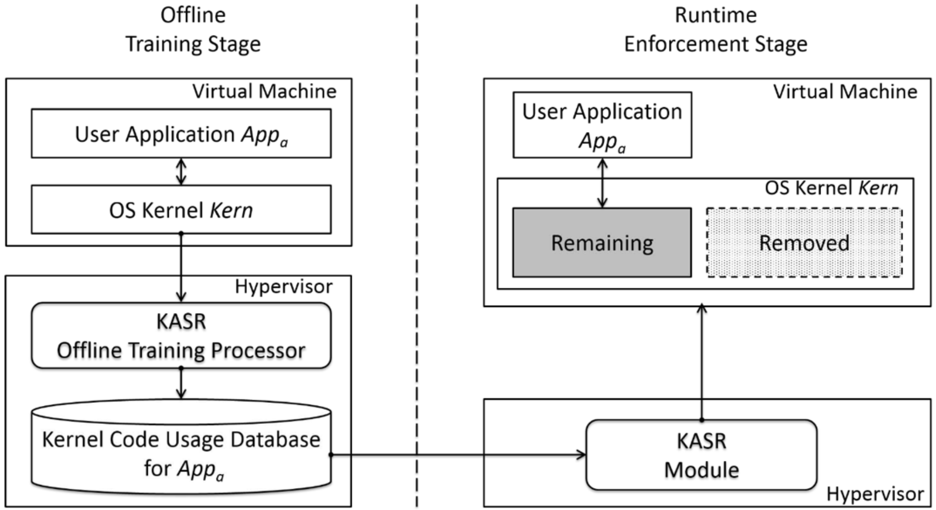
- Prior approaches and limitations
  - Build from scratch – build a new kernel with the reduced functionalities
    - Compatibility issues with the commodity hardware
    - Time consuming, more potential vulnerabilities, and so on
  - Re-construction – current monolithic kernel
    - Modifying existing kernel is not easy
  - Customization – tailor existing kernels without modifications
    - The lack of Linux distribution support and overhead
    - Compatibility issues with existing Kernel level protections (e.g., kASLR)

KASR: Kernel attack surface reduction

- Design goals
  - Reliable – reduce the attack surface
  - Transparent
    - Should work with the commodity kernels
    - Does not need the source code
    - Does not break the kernel code integrity
  - Efficient – minimal impact on the kernel performance (e.g., ~ 1% increase)

KASR: Kernel attack surface reduction

- Design choices
  - Use hypervisor
    - It runs the target system's kernel in a VM
    - It has a complete view of the VM's memory allocations (and de-allocations)
    - It supports libraries for dynamically altering the memory allocations (and de-allocations)
  - Do it at the page-level



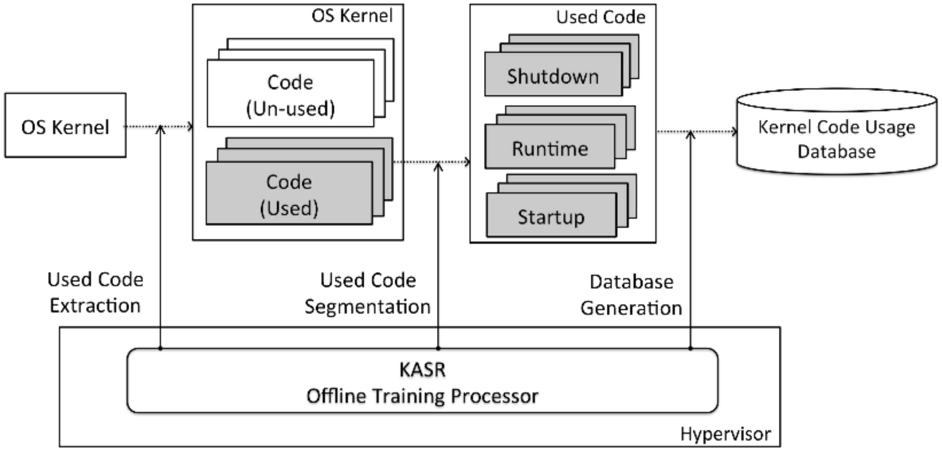
- Threat model
  - The hypervisor is clean
  - The VM is clean in the offline stage
  - It can be compromised in runtime

KASR: Kernel attack surface reduction

- Operation: profile-then-deploy
  - Offline profiling – identify the pages used by the VM
  - Online – selectively activate the used code (only) when requested

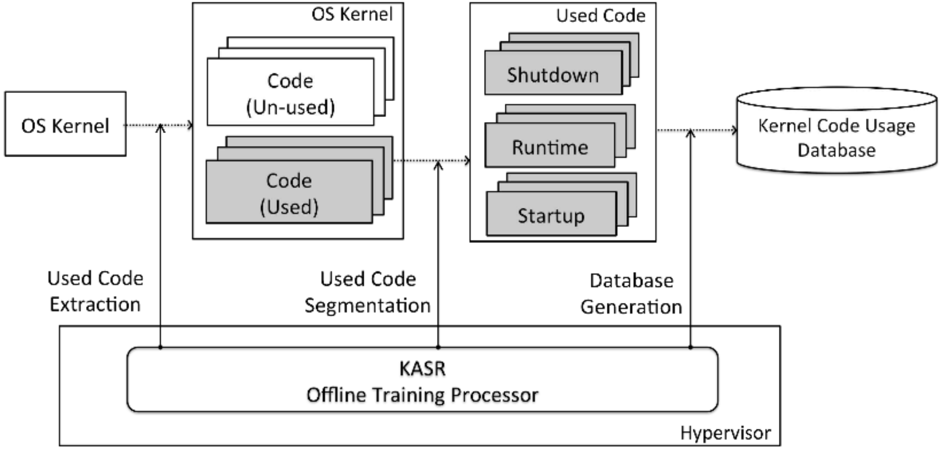
KASR: KERNEL ATTACK SURFACE REDUCTION

- Operation: profile-then-deploy
  - Offline profiling (training)
    - Kernel image: used code extraction via hypervisor
      - Used pages – *ftrace* is not appropriate, e.g., it misses pages at the start-up phase
      - Remove the executable permissions from all code pages of the kernel image
      - Get an exception, then set it to executable and record it



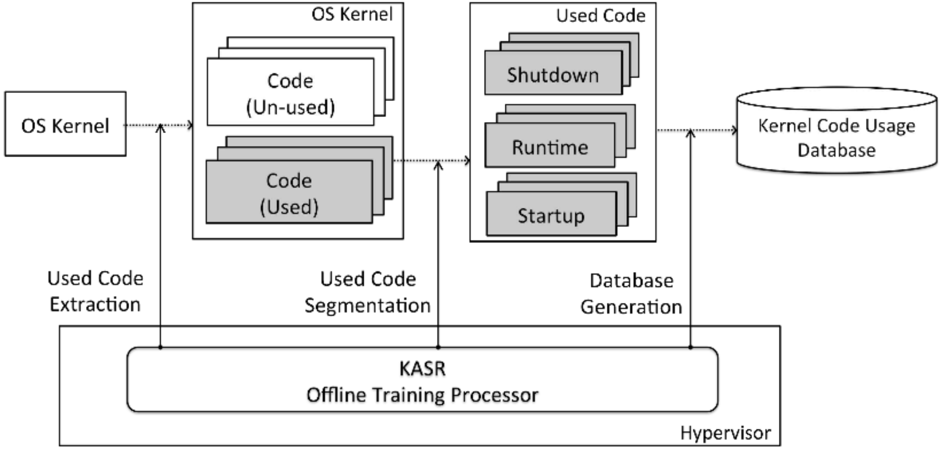
KASR: Kernel Attack Surface Reduction

- Operation: profile-then-deploy
  - Offline profiling (training)
    - Kernel modules
      - Linux kernel modules (LKMs) are dynamically loaded and unloaded at runtime
      - Pages allocated by LKMs are freed and re-allocated, e.g., think of a USB driver
      - Only the pages causing exceptions can gain the executable permission



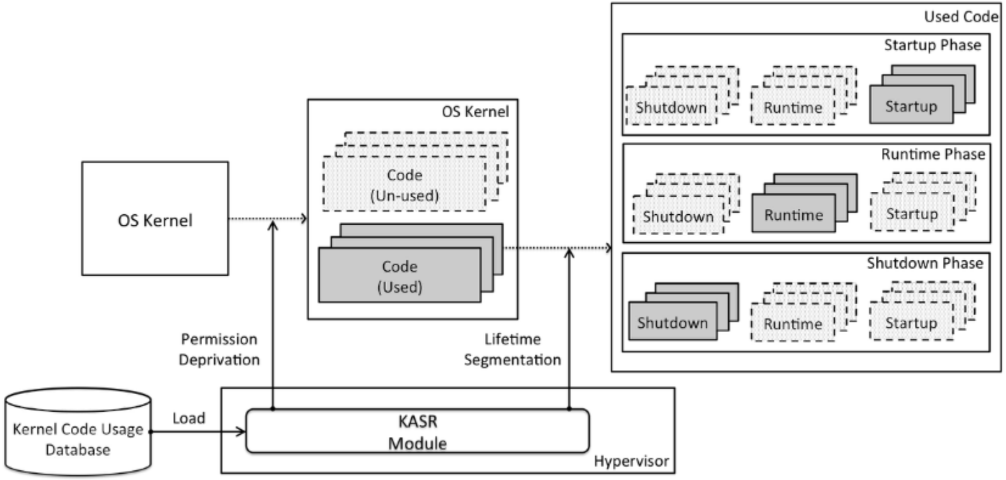
KASR: Kernel Attack Surface Reduction

- Operation: profile-then-deploy
  - Offline profiling (training)
    - Page identification
      - Use page frame number (PFN)
      - Address layout should be unique and consistent at a start-up, what? kASLR
      - Use multi-hash-value approach (why not one-hash, fuzzy hash?)



KASR: Kernel Attack Surface Reduction

- Operation: profile-then-deploy
  - Runtime enforcement
    - Permission deprivation: remove the execution permission from un-used pages
    - Lifetime segmentation: even for the used code, it deprives the execution permission



KASR: KERNEL ATTACK SURFACE REDUCTION

- KASR effectiveness (5 applications, e.g., httperf)
  - Substantial reduction of kernel pages
    - 53 – 54% reduction after the permission deprivation
    - 61 – 64% reduction after the lifetime segmentation

KASR: Kernel attack surface reduction

- KASR effectiveness (5 applications, e.g., httperf)
  - Substantial reduction of kernel pages
    - 53 – 54% reduction after the permission deprivation
    - 61 – 64% reduction after the lifetime segmentation
  - 40% CVE removals in the memory
    - Modules containing past CVE vulnerabilities are not loaded into the kernel
    - Among the total CVEs found in the past 2 years of these applications' GitHub repo
  - Rootkit prevention
    - LKM is the attack vector
      - Step 1: Inject malicious code into the kernel memory
      - Step 2: Hook the code on target kernel functions (e.g., syscalls)
      - Step 3: Transfer the kernel context flow to the code
    - These rootkits are not available to do the step 3

KASR: Kernel attack surface reduction

- KASR effectiveness (5 applications, e.g., httperf)
  - Substantial reduction of kernel pages
    - 53 – 54% reduction after the permission deprivation
    - 61 – 64% reduction after the lifetime segmentation
  - 40% CVE removals in the memory
    - Modules containing past CVE vulnerabilities are not loaded into the kernel
    - Among the total CVEs found in the past 2 years of these applications' GitHub repo
  - Rootkit prevention
    - LKM is the attack vector
      - Step 1: Inject malicious code into the kernel memory
      - Step 2: Hook the code on target kernel functions (e.g., syscalls)
      - Step 3: Transfer the kernel context flow to the code
    - These rootkits are not available to do the step 3
  - Marginal performance overhead (1.47% at max and 0.23% on average)

PROPER ACCESS CONTROL

Linux Access Control

- Everything is a file
  - **Definition:** a named collection of data (*e.g.*, movie.csv containing movie data)
  - **POSIX** : a sequence of data bytes
  - ***NIX OS** : everything
    - Files on secondary storages, *e.g.*, disks
    - Devices (mouse, keyboard, monitor, …)
    - Network devices (network card, sockets in OS, …)
    - Inter-process communications (pipes, sockets, …)

LINUX ACCESS CONTROL – CONT'D

- Directories
  - **Definition :** a folder containing files and directories
  - **Motivation:**
    - Scenario: one day you create 100k+ files and the next day, you want to use them
  - **Solution :**
    - **S0:** You are Von Neumann; remember all the files
    - **S1:** Your system creates a folder containing all the files for each user
    - **S2:** Your system creates multiple folders containing the same kinds

LINUX ACCESS CONTROL – CONT'D

~/lecture/CS578$ ls -alh

total 312K

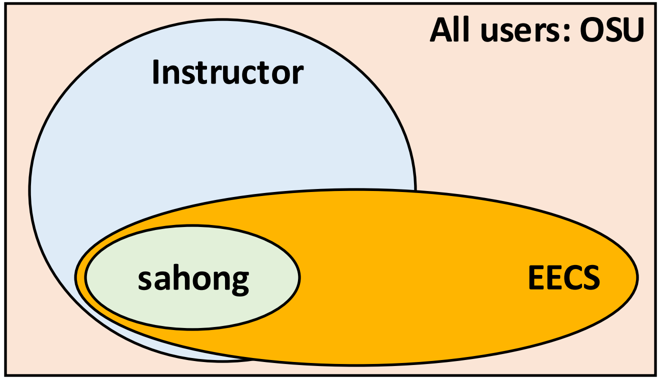
drwxrwx---.	6	sahong	upg1xxxx	186	Apr 10 22:14	.
drwxrwx---.	3	sahong	upg1xxxx	73	Apr 5 19:58	..
drwxrwx---.	2	sahong	upg1xxxx	95	Apr 5 19:58	bufferoverflow
drwxrwx---.	2	sahong	upg1xxxx	52	Apr 4 09:02	bufferoverrun
drwxrwx---.	8	sahong	upg1xxxx	299	Apr 10 21:56	.git
-rw-rw----.	1	sahong	upg1xxxx	430	Apr 5 19:56	.gitignore
lrwxrwxrwx.	1	sahong	upg1xxxx	22	Apr 10 22:14	home -> /nfs/stak/users/hongsa
-rw-rw----.	1	sahong	upg1xxxx	44	Apr 4 08:15	README.md
drwxrwx---.	2	sahong	upg1xxxx	79	Apr 5 20:07	thread
Permission	**# hard-link**	**owner**	**owner-group**	**size (b)**	**last modified**	**name**

LINUX ACCESS CONTROL – CONT’D

~/lecture/CS578$ ls -alh

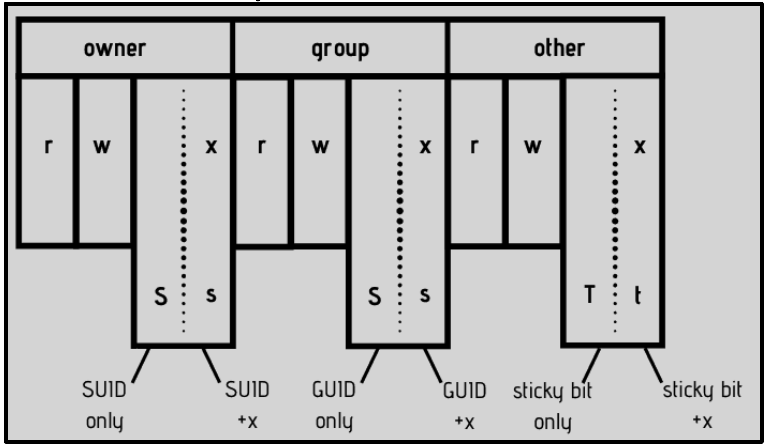
total 312K						
drwxrwx---.	6	sahong	upg1xxxx	186	Apr 10 22:14	.
drwxrwx---.	3	sahong	upg1xxxx	73	Apr 5 19:58	..
drwxrwx---.	2	sahong	upg1xxxx	95	Apr 5 19:58	bufferoverflow
… <omit the entries>						
Permission	**# hard-link**	**owner**	**owner-group**	**size (b)**	**last modified**	**name**

- Linux controls the access to files or directories based on three categories:
  - user : owner of a file or a directory
  - group : the group where users are
  - others: all the other users



LINUX ACCESS CONTROL – CONT'D

- Permission
  - **Read** : one can read files and directories with 'r' permission
  - **Write** : one can write files and dirs. with 'w' permission
  - **Execute:** one can execute files and dirs. with 'x' permission
  - **SetUID** : one can execute files and dirs. with the permissions of the owner/group of the command
  - **sticky** : except the creator and the root, no one can modify or delete the file



LINUX ACCESS CONTROL – CONT'D

~/lecture/CS578$ ls -alh

total 312K

… <omit the entries>

-rw-rw----.	1	sahong	upg1xxxx	44	Apr 4 08:15	README.md
drwxrwx---.	2	sahong	upg1xxxx	79	Apr 5 20:07	thread
Permission	**# hard-link**	**owner**	**owner-group**	**size (b)**	**last modified**	**name**

- Permission representation
  - drwxrwx---

[Type] d: directory, -: file

[User] the first three letters (rwx)
[Group] the second three letters (rwx)
[Others] the last three letters (---)

LINUX ACCESS CONTROL – CONT'D

~/lecture/CS578$ ls -alh

total 312K

… <omit the entries>

-rw-rw----.	1	sahong	upg1xxxx	44	Apr 4 08:15	README.md
drwxrwx---.	2	sahong	upg1xxxx	79	Apr 5 20:07	thread
Permission	**# hard-link**	**owner**	**owner-group**	**size (b)**	**last modified**	**name**

- Permission representation
  - drwxrwx---:
    - 770
    - 111111000

Interpretation

- Decimal #: 1st (user), 2nd (group), 3rd (others)
  + ex. 770 : 7 (user), 7 (group), 0 (others)
- Each #: Binary number
  1st (read), 2nd (write), 3rd (execute)
  + ex. 7 : 111 (rwx)
  + ex. 6 : 110 (rw)
  + ex. 600 : 110 000 000 (your ssh key)

LINUX ACCESS CONTROL – CONT'D

- Permission
  - **SetUID** : one can execute files and dirs. with the permissions of the owner/group of the command (e.g., /usr/bin/passwd)
  - **sticky** : except the creator and the root, no one can modify or delete the file (e.g., /tmp)

LINUX ACCESS CONTROL – CONT'D

- Linux supports uid-setting system calls
  - *setuid*, *seteuid*, *setreuid*, and *setresuid*
  - Three user IDs
    - Real UID: the user ID who launched the process
    - Effective UID: the user ID who will be effective while the process is running (e.g., setuid)
    - Saved UID: the user ID saved when there's a switch btw the real and effective UIDs
  - **Problem:** the setuid model is *not well-understood* and *poorly used*
    - What is the *appropriate privilege*?

DEMYSTIFY THE SETUID USAGE

- Goals
  - Understand the semantics of security operation APIs in OS
  - Check their documentations
  - Detect inconsistency between the documentations and implementations
  - Build security properties and check them in programs automatically

DEMYSTIFY THE SETUID USAGE

- Desiderata: principles of least privilege
- **Solution approach:** formal model
  - What's the formal model? Finite state automata (FSA)
  - How to define the state?
    - (r, e, s) – real, effective and set UIDs
    - In Linux, it becomes (r, e, s, b) – b stands for the *setuid* bit
  - How to extract the formal model?
    - Design a model extraction algorithm
    - Run the algorithm with simulations and build the model
  - What are the potential challenges?
    - The state space is too large
      - Use symmetry (isomorphism)
      - Non-root uID (100, 100, 100) is the same as (200, 200, 200)
    - No assumption about outside alterations of these user IDs

DEMYSTIFY THE SETUID USAGE

- Desiderata: principles of least privilege
- **Solution approach:** formal model
  - What's the formal model? Finite state automata (FSA)
  - How to define the state?
    - (r, e, s) – real, effective and set UIDs
    - In Linux, it becomes (r, e, s, b) – b stands for the *setuid*
  - How to extract the formal model?
    - Design a model extraction algorithm
    - Run the algorithm with simulations and build the mode
  - What are the potential challenges?
    - The state space is too large
      - Use symmetry (isomorphism)
      - Non-root uID (100, 100, 100) is the same as (200, 200,
    - No assumption about outside alterations of these user

GETSTATE():
1. Call `getresuid(&r,&e,&s)`.
2. Return (r, e, s).

SETSTATE(r, e, s):
1. Call `setresuid(`r, e, s`)`.
2. Check for error.

GETALLSTATES():
1. Pick n arbitrary uids $u_1, \dots, u_n$.
2. Let $U := \{u_1, \dots, u_n\}$.
3. Let $S := \{(r, e, s) : r, e, s \in U\}$.
4. Let $C := \{$`setuid(`x`)`, `setreuid(`x, y`)`, `setresuid(`x, y, z`)`, $\cdots$ $: x, y, z \in U \cup \{-1\}\}$.
5. Return (S, C).

BUILDMODEL():
1. Let $(S, C) :=$ GETALLSTATES().
2. Create an empty FSA with statespace S.
3. For each $s \in S$, do:
4. For each $c \in C$, do:
5. Fork a child process, and within the child, do:
6. Call SETSTATE(s), and then invoke c.
7. Finally, let $s' :=$ GETSTATE(), pass s' to the parent process, and exit.
8. Add the transition $s \xrightarrow{c} s'$ to the FSA.
9. Return the newly-constructed FSA as the model.

SECURITY BENEFITS

- Applications
  - Identify documentation errors, *setuid(2)* in Linux man page
  - Detect inconsistencies in the Linux implementation and documentation
    - *fsuid* in Linux is used for filesystem permission checking
    - *fsuid* becomes 0 only if at least one of r, e, s UIDs is 0
  - UID-setting system call's proper usage, e.g., *sendmail* 8.10.1

SECURITY RECOMMENDATIONS

- Guidelines
  - General
    - Selecting appropriate system calls
    - Obeying the proper order of these calls
    - Verifying proper execution of system calls
  - An improved API
    - Proposed new API
    - Implementation
    - Evaluation

(Kernel) fuzzing

PRELIMINARIES ON FUZZING

- An automated software testing technique
- Goal:
  - To find program inputs that expose a software bug (or vulnerability)
- Approach:
  - Construct inputs randomly (1990s)
  - Run a program on them until it crashes

PRELIMINARIES ON FUZZING

- History of fuzzing techniques
  - 2000s – fuzzing uses input mutations (e.g., bit-flips, bytes, insertion/deletion, …)
    - Black-box approach
  - 2010s – input mutations are based on the code coverage
    - White-box (scalability issues) / gray-box approach
    - Program instrumentation is needed to check the code coverage (e.g., gcov, LLVM-based, …)
    - It receives the feedback from the crash and create the next input for increasing coverage
  - 2014s – Coverage-guided fuzzing with AFL

(KERNEL) FUZZING

- Kernel fuzzing != software fuzzing
  - A crash will terminate the kernel, need to setup everything again
  - Kernel binary instrumentation is too complex and computationally demanding
  - Coverage-guided fuzzing: it is slow, feedback heavily relies on drivers and re-compilation
  - Many indeterminism (threads, stateful-ness, others…)
  - No generic ways to communicate with kernels and drivers, e.g., like stdin
  - …

(KERNEL) FUZZING

- Prior work

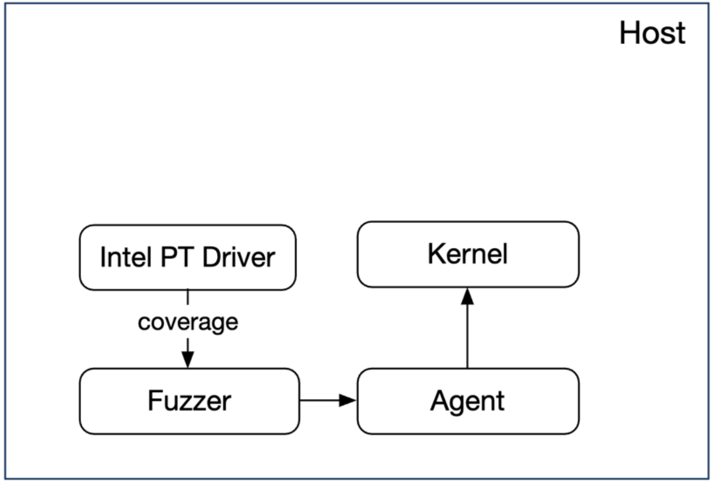
	Fast	Crash Tolerant	OS Independent	Binary Only
TriforceAFL (Jesse Hertz & Tim Newsham, NCC Group)	✗	✓	~	✓
Syzkaller (Dmitry Vyukov)	✓	✓	✗	✗
AFL Filesystem Fuzzer (Vegard Nossum & Quentin Casanovas, Oracle)	✓	~	✗	✗
PT Kernel Fuzzer (Richard Johnson, Talos)	✓	✗	✗	✓

KERNEL-AFL (KAFL)

- Intel Processor Trace
  - Intel's modern CPUs support tracing all the instructions executed by a process
  - Hardware-supported feature, so it is
    - Computationally efficient
    - Reliable
    - (Mostly) OS Independent
    - No source code access is required

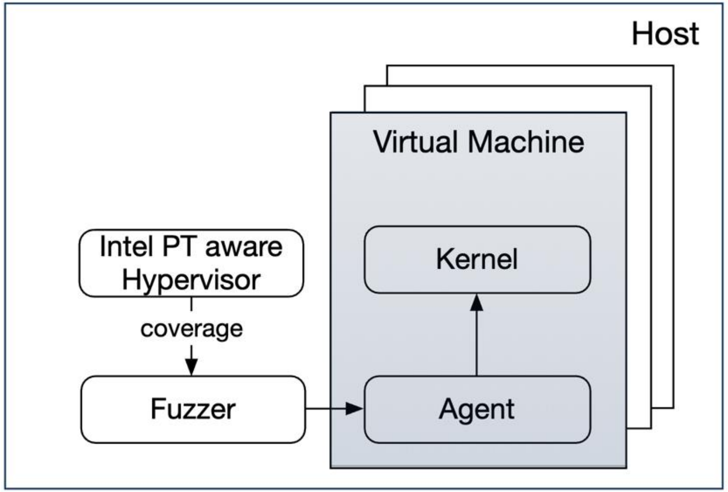
Kernel-AFL (kAFL)

- Intel Processor Trace
  - Intel's modern CPUs support tracing all the instructions executed by a process
  - Hardware-supported feature, so it is
    - Computationally efficient
    - Reliable
    - (Mostly) OS Independent
    - No source code access is required
  - Solution approach
    - Natively use Intel's PT will not work
    - It causes the kernel crash!



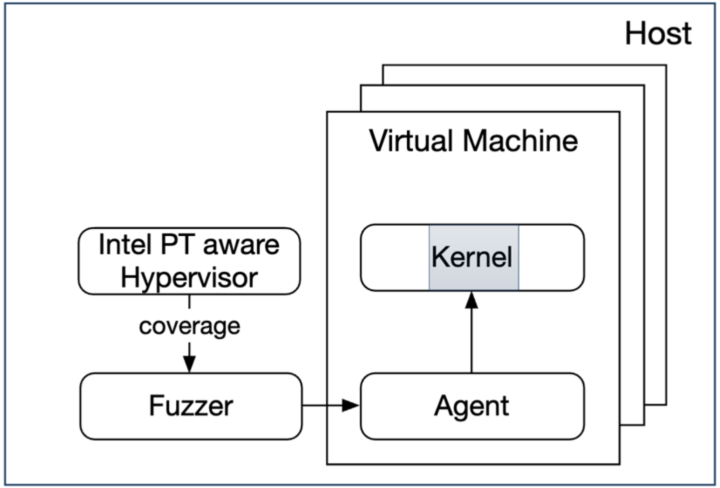
KERNEL-AFL (KAFL)

- Intel Processor Trace
  - Intel's modern CPUs support tracing all the instructions executed by a process
  - Hardware-supported feature, so it is
    - Computationally efficient
    - Reliable
    - (Mostly) OS Independent
    - No source code access is required
  - Solution approach
    - ~~Natively use Intel's PT will not work~~
    - Use a hypervisor
    - But it is still expensive to monitor the entire VM



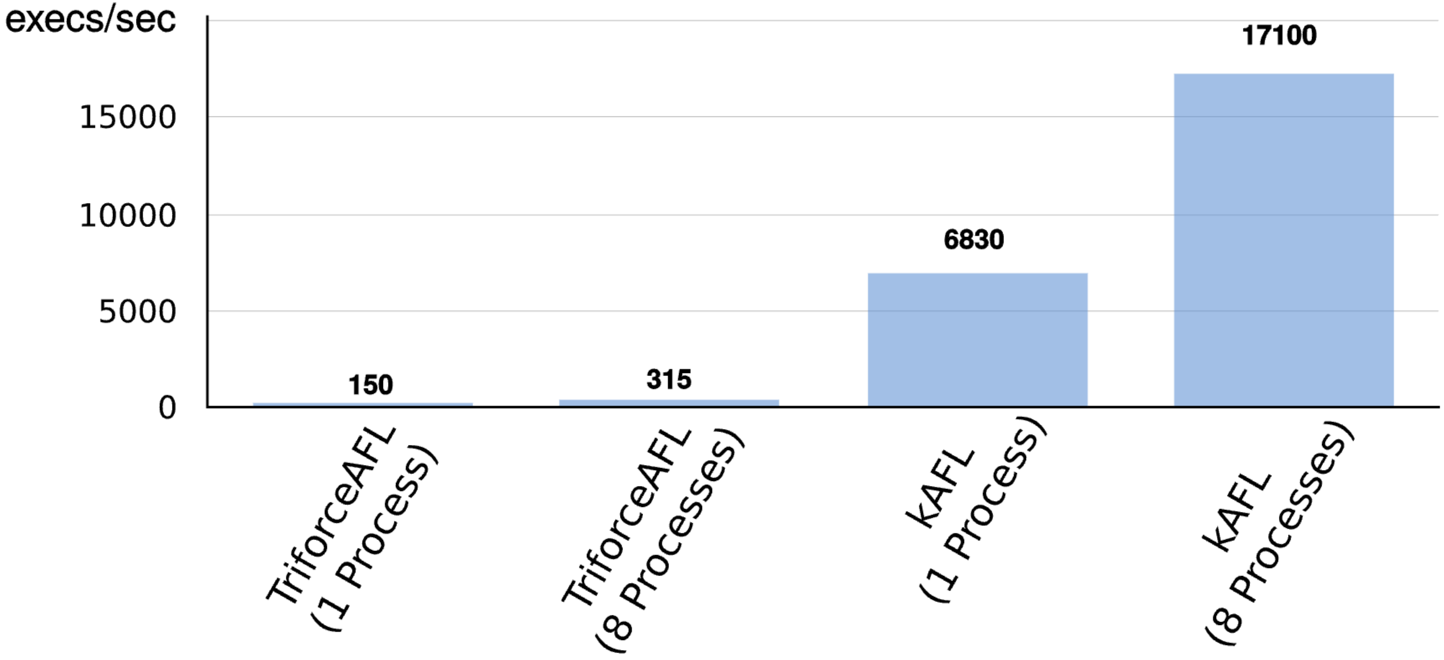
KERNEL-AFL (KAFL)

- Intel Processor Trace
  - Intel's modern CPUs support tracing all the instructions executed by a process
  - Hardware-supported feature, so it is
    - Computationally efficient
    - Reliable
    - (Mostly) OS Independent
    - No source code access is required
  - Solution approach
    - ~~Natively use Intel's PT will not work~~
    - Use a hypervisor
    - But it is still expensive to ~~monitor the entire VM~~
    - Filter out the trace based on:
      vCPUs, Supervisor, CR3 register, Instructions
    - Fuzzer communicates with the VM using the agent



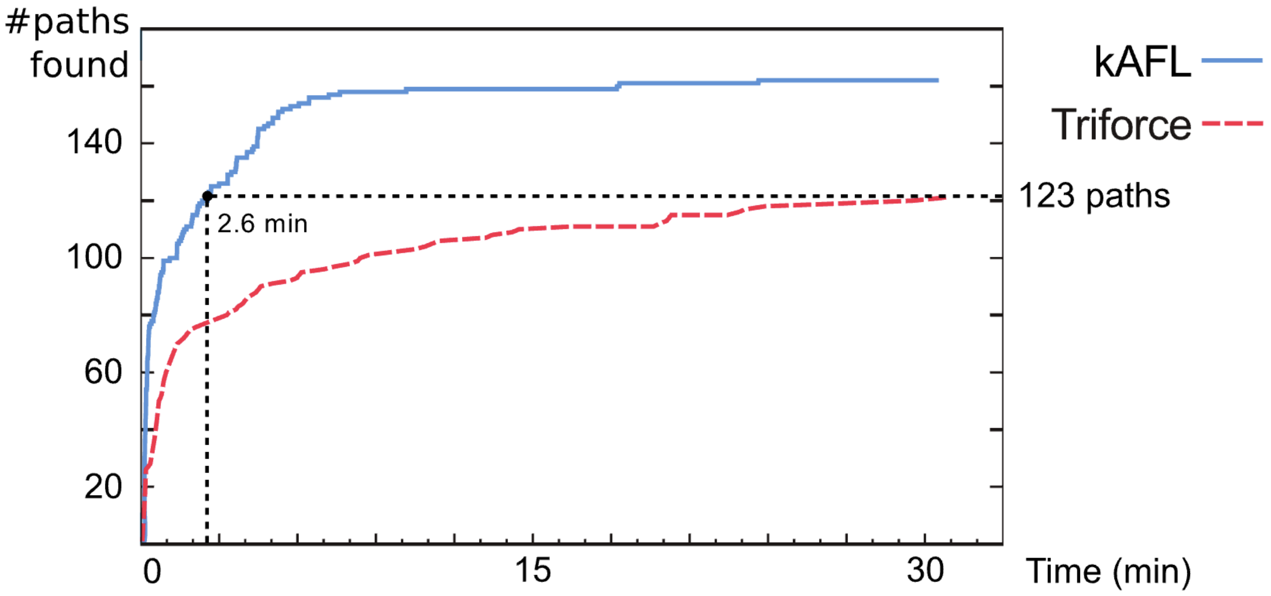
KERNEL-AFL (KAFL)

- Evaluation
  - Performance: 17,100 executions per second, on 8 processors



KERNEL-AFL (KAFL)

- Evaluation
  - Performance: 17,100 executions per second, on 8 processors
  - Code-coverage: to find the 123 distinct paths, kAFL takes 5-7 min, while prior approach takes ~2 hours



KERNEL-AFL (KAFL)

- Evaluation
  - Performance: 17,100 executions per second, on 8 processors
  - Code-coverage: to find the 123 distinct paths, kAFL takes 5-7 min, while prior approach takes ~2 hours
  - Discovered vulnerabilities
    - Linux: keyctl null pointer dereferences (CVE-1026-8650)
    - Linux: ext4 memory corruption
    - Linux: ext4 error handling
    - Windows: NTFS div-by-zero
    - MacOS: HFS div-by-zero
    - MacOS: HFS Assertion fail
    - MacOS: HFS Use-after-free
    - MacOS: APFS memory corruption

Thank You!

Sanghyun Hong

https://secure-ai.systems/courses/Sec-Grad/current

Oregon State University

SAIL
Secure AI Systems Lab