

A Study of Kernel Telemetry Options for Security-Oriented Provenance

ISC 2026 – Rennes, France

Paul Housel^{1,2} **Olivier Levillain**² **Sylvie Laniepce**¹
Nicolas Dejon¹ **Hervé Debar**²

¹Orange Research, Châtillon, France

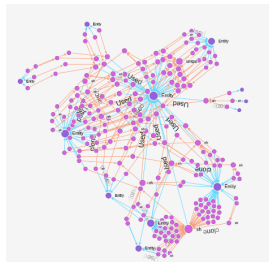
²Institut Polytechnique de Paris, SAMOVAR, Télécom SudParis, Palaiseau, France

28th of October, 2026



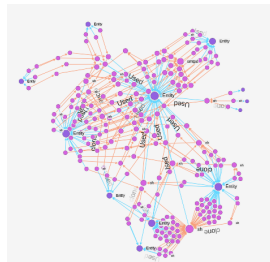
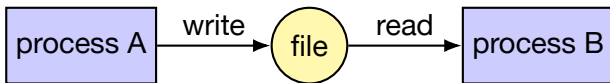
Whole-system provenance [Carata et al., 2014]

- observe the behavior of the whole system from kernel telemetry
- model how subjects allocate and use objects
- represented as a provenance graph



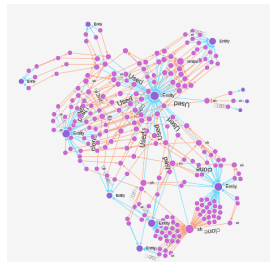
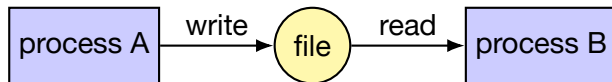
Whole-system provenance [Carata et al., 2014]

- observe the behavior of the whole system from kernel telemetry
- model how subjects allocate and use objects
- represented as a provenance graph



Whole-system provenance [Carata et al., 2014]

- observe the behavior of the whole system from kernel telemetry
- model how subjects allocate and use objects
- represented as a provenance graph

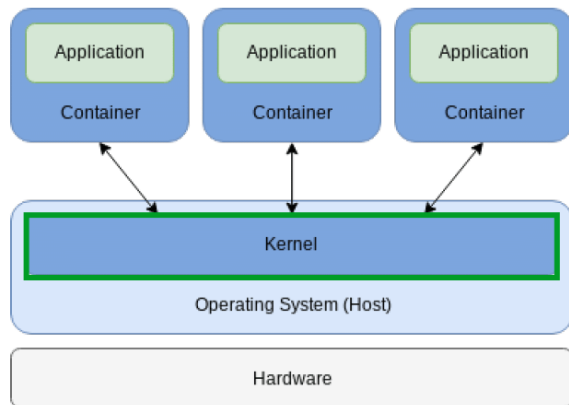


- security use cases:
 - ▶ Provenance-based Intrusion Detection Systems (PIDS) with graph queries or unsupervised learning [Jiang et al., 2025; Bilot et al., 2025]
 - ▶ forensics: post-mortem audits & threat hunting [Pasquier et al., 2018]

Whole-system provenance is captured from the kernel

Why?

- Robustness
- Efficiency
- Visibility



Challenges of provenance systems [Inam et al., 2023]

Graphs quickly grow very large

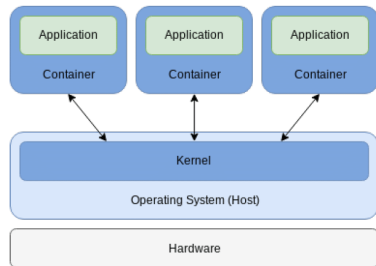
- performance overhead → capture availability
- storage overhead → shorter capture window
- important signals buried in noise → hard and costly analysis

Deployment in production environments

- large and heterogeneous fleet → portability issues
- attacker on the host → attacks on capture integrity and availability

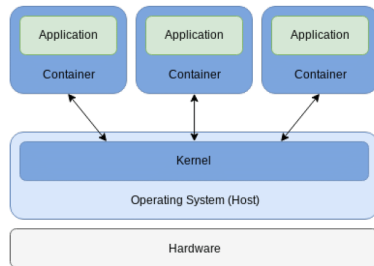
Challenges amplified in cloud-native environments

- fine-grained, per-container capture
- distinguish container and host activity
- higher volume of system activity
- distributed infrastructure
- container- and orchestration-specific semantics



Challenges amplified in cloud-native environments

- fine-grained, per-container capture
- distinguish container and host activity
- higher volume of system activity
- distributed infrastructure
- container- and orchestration-specific semantics



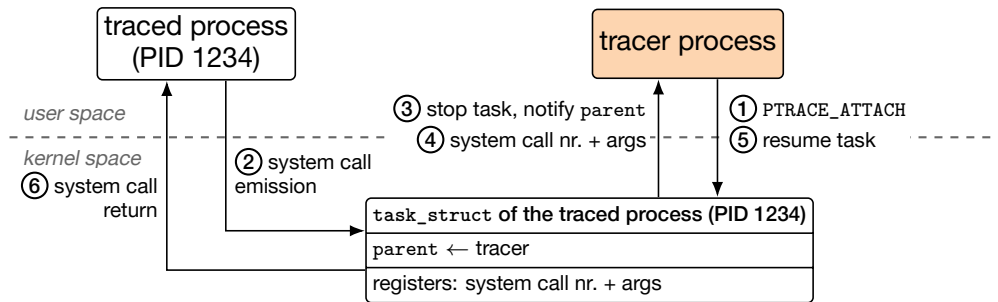
In our work, we show

- limitations present across the majority of provenance systems
- mostly caused by the capture layer of provenance systems

Capture approaches

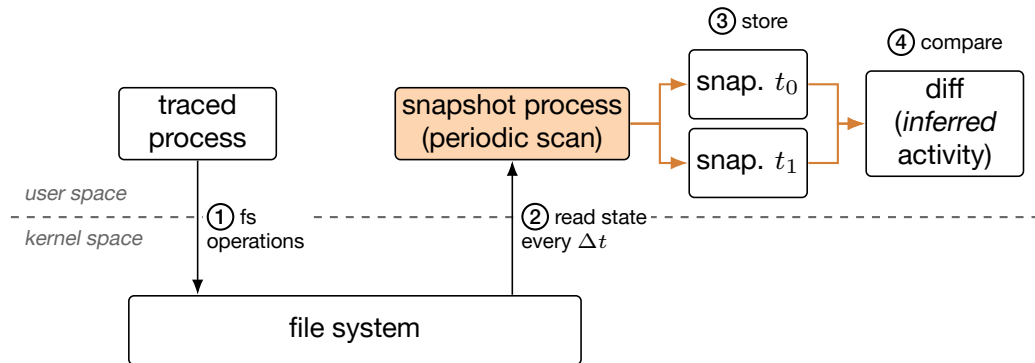
How to capture kernel telemetry? From user space

approach	robustness	visibility	efficiency	portability	safety	example provenance system using it
<code>ptrace</code>	○	○	○	►	►	ALASTOR [Datta et al., 2022]



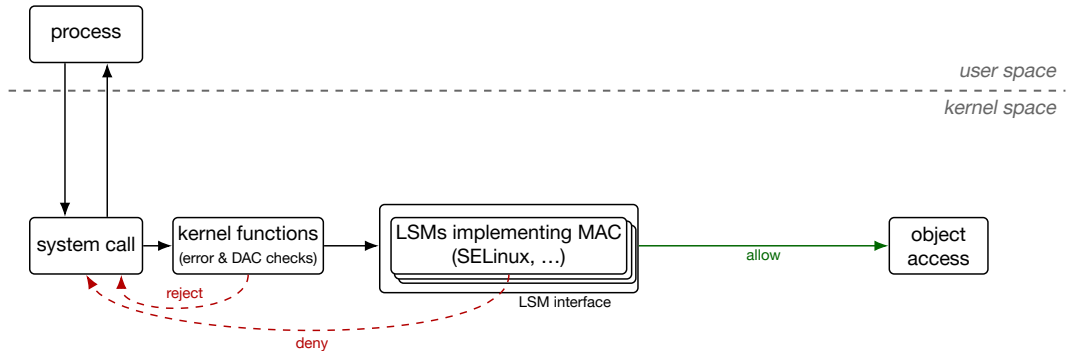
How to capture kernel telemetry? From user space

approach	robustness	visibility	efficiency	portability	safety	example provenance system using it
ptrace	○	○	○	►	►	ALASTOR [Datta et al., 2022]
fs snapshot	○	○	●	►	●	ARTISAN [Yu et al., 2024]



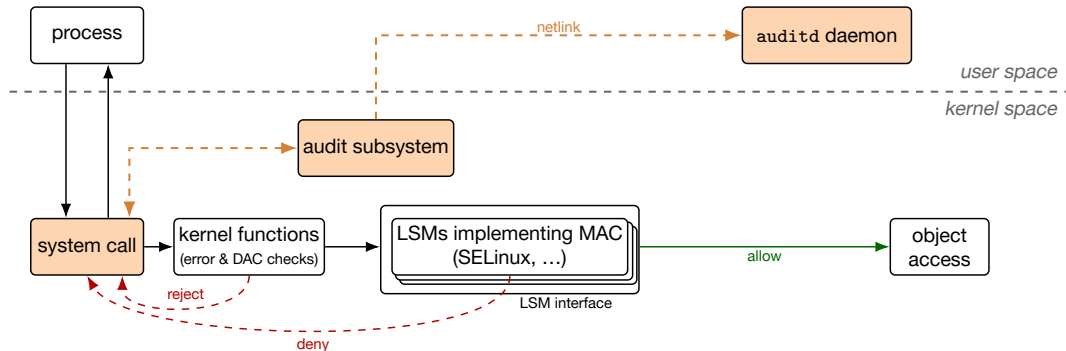
How to capture kernel telemetry? From the kernel

→ from the kernel: increased *robustness*, *efficiency* and *visibility*



How to capture kernel telemetry? From the kernel

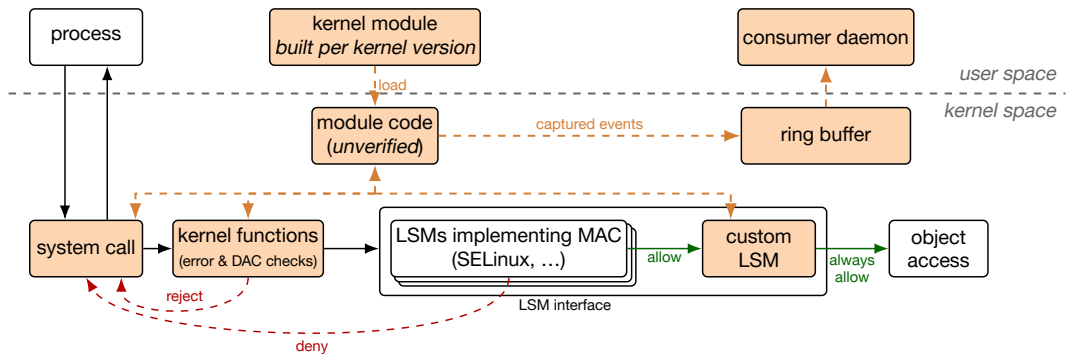
→ from the kernel: increased *robustness*, *efficiency* and *visibility*



Problem: integrated systems are too limited in *visibility* and *portability*

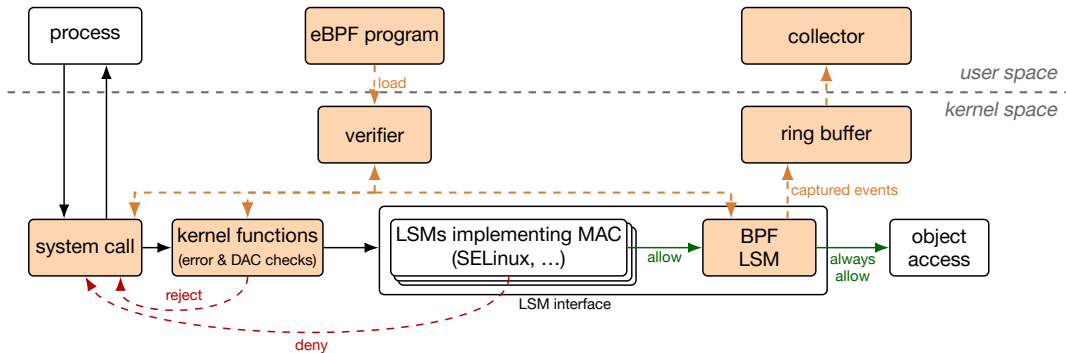
How to capture kernel telemetry? From the kernel

approach	portability	safety	example provenance system using it
out-of-tree kernel module	○	▶	CamFlow [Pasquier et al., 2017]



How to capture kernel telemetry? From the kernel

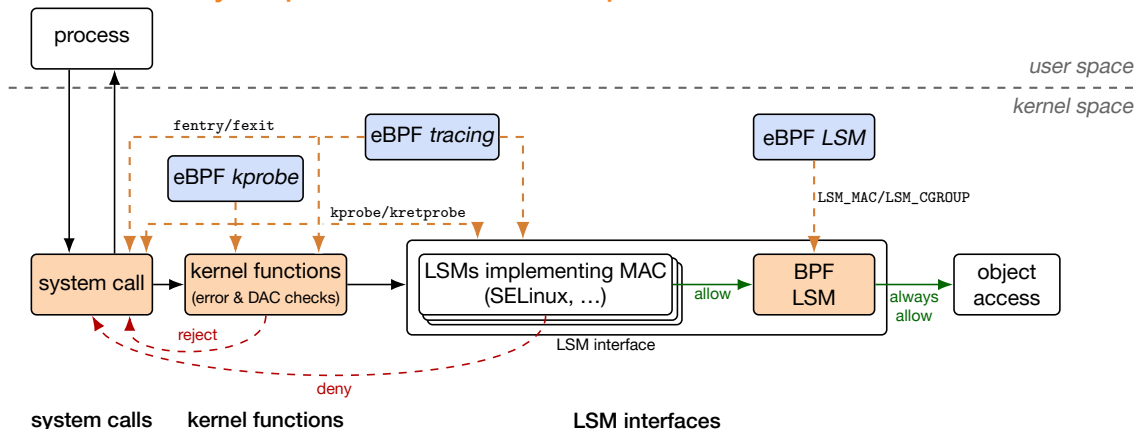
approach	portability	safety	example provenance system using it
out-of-tree kernel module	○	▶	CamFlow [Pasquier et al., 2017]
eBPF	▶	●	ProvBPF [Lim et al., 2021]



→ eBPF has become the most prominent approach: we focus on it

Capture location

Kernel telemetry capture locations for provenance



→ the capture location has an important impact on provenance

kprobe eBPF attachment via INT3 breakpoint

(1) Original function

```
some_kernel_func:
    PUSH RBP
    MOV  RBP, RSP
    ...
    RET
```

(2) After attach: 1st instr → breakpoint

```
some_kernel_func:
    INT3          ; was PUSH RBP
    MOV  RBP, RSP
    ...
    RET
```

PUSH RBP is put aside to run later

breakpoint

INT3 breakpoint handler

- ① save registers → pt_regs
- ② search kprobe by address
- ③ run its eBPF program
- ④ run saved PUSH RBP

resume

registered kprobes

address	program
vfs_read	...
tcp_connect	...
some_kernel_func	eBPF

costly breakpoint → less efficient

fentry eBPF attachment via BPF trampoline

(1) Compiled with `-mfentry`

```
some_kernel_func:
    NOP                ; fentry pad
    PUSH RBP
    MOV RBP, RSP
    ...
    RET
```

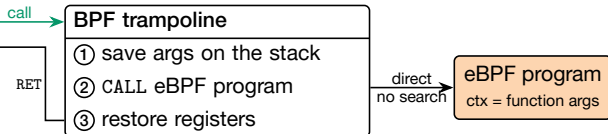
fentry eBPF attachment via BPF trampoline

(1) Compiled with `-mfentry`

```
some_kernel_func:
    NOP                ; fentry pad
    PUSH RBP
    MOV RBP, RSP
    ...
    RET
```

(2) After attach: `NOP` → `CALL`

```
some_kernel_func:
    CALL trampoline
    PUSH RBP            ; untouched
    MOV RBP, RSP
    ...
    RET
```

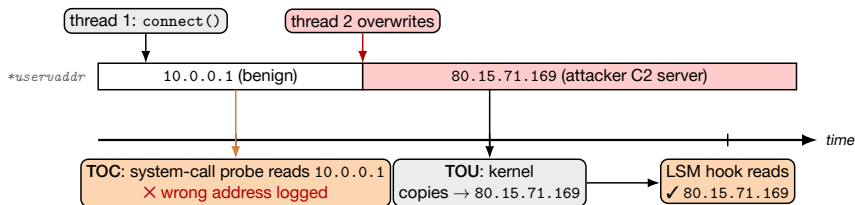


no need for a costly breakpoint → more efficient

LSM hooks are robust to TOCTOU attacks [Guo and Zeng, 2021]

1. Time of check (TOC): system-call probes dereference user-space pointers
2. Time of use (TOU): the kernel dereferences the same user-space pointers

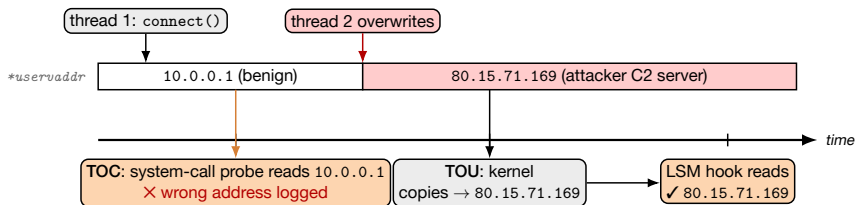
```
sys_connect(int fd, struct sockaddr __user *servaddr)
```



LSM hooks are robust to TOCTOU attacks [Guo and Zeng, 2021]

1. Time of check (TOC): system-call probes dereference user-space pointers
2. Time of use (TOU): the kernel dereferences the same user-space pointers

```
sys_connect(int fd, struct sockaddr __user *useraddr)
```

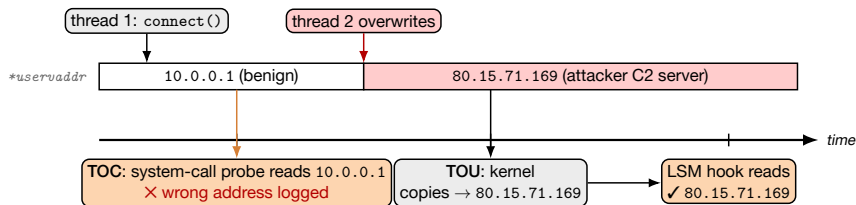


System calls	with ≥ 1 user-space pointer	file-related	socket address
all (Linux v6.14)	69% (282/407)	96	8

LSM hooks are robust to TOCTOU attacks [Guo and Zeng, 2021]

1. Time of check (TOC): system-call probes dereference user-space pointers
2. Time of use (TOU): the kernel dereferences the same user-space pointers

```
sys_connect(int fd, struct sockaddr __user *useraddr)
```



System calls	with ≥ 1 user-space pointer	file-related	socket address
all (Linux v6.14)	69% (282/407)	96	8

~73% of system calls that open-source security policies rely on are vulnerable

LSM hooks offer good portability

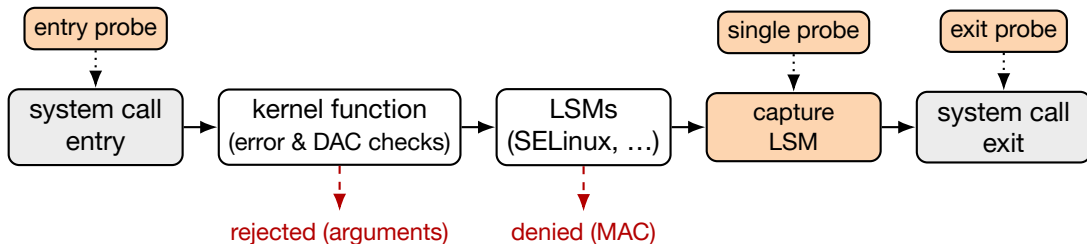
Portability of LSM hooks compared to system calls:

- unstable LSM hooks are not fundamental to provenance
- LSM hooks are more stable than system calls across different architectures

Capturing provenance with LSM hooks is more efficient

Provenance represents what actually happened in the system

- system calls: two probes, entry (arguments) + exit (return value)
- LSM hooks: after error & access control checks → a single probe



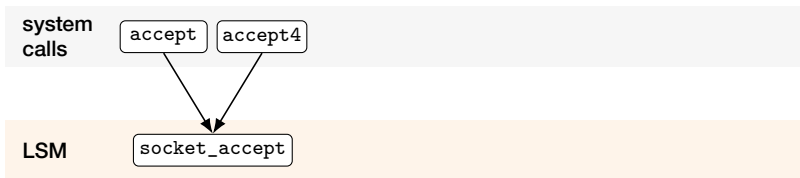
LSM hooks map directly to provenance events

- a system call is only one of several ways to cause a provenance event
- an LSM hook fires on the provenance event itself

LSM hooks map directly to provenance events

- a system call is only one of several ways to cause a provenance event
- an LSM hook fires on the provenance event itself

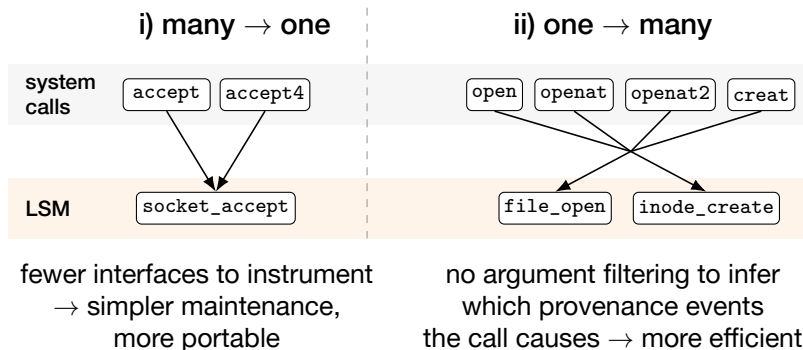
i) many → one



fewer interfaces to instrument
→ simpler maintenance,
more portable

LSM hooks map directly to provenance events

- a system call is only one of several ways to cause a provenance event
- an LSM hook fires on the provenance event itself



Performance overhead by eBPF program type

Kernel interfaces instrumented in our experimental setup

Tracing location	Traced interface per workload (captured event)		
	<i>network</i> (accept conn.)	<i>file</i> (file creation)	<i>process</i> (clone)
<i>system call</i>	accept, accept4	creat, open ^a , openat ^a , openat2 ^a	clone, clone3
<i>LSM</i>	socket_accept	inode_init_security	task_alloc

^a O_CREAT flag filtered to capture only file creations

Performance overhead by eBPF program type and capture location



■ *tracing* and *LSM* have the lowest per-hook cost (trampoline)

Performance overhead by eBPF program type and capture location



- *tracing* and *LSM* have the lowest per-hook cost (trampoline)
- *LSM* abstraction does not require filtering → more gain for *file* workload

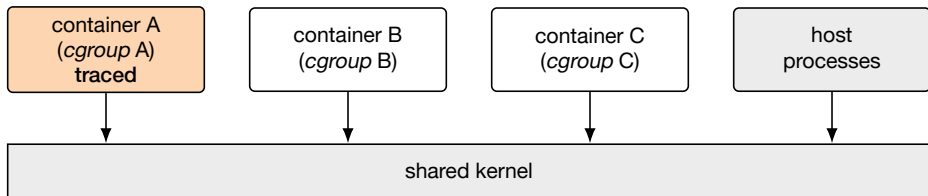
Performance overhead by eBPF program type and capture location



- *tracing* and *LSM* have the lowest per-hook cost (trampoline)
- *LSM* abstraction does not require filtering → more gain for *file* workload
- *LSM*'s single attachment offers substantial savings

Production needs granularity

- e.g., trace a single exposed service on a multi-tenant host



only *LSM* programs can be attached per *cgroup* → most efficient filtering

LSM: the most attractive, yet least-used basis

capture approach: eBPF has become dominant for capturing provenance

LSM hooks are the most attractive

- lower overhead
 - ▶ single attachment
 - ▶ more efficient trampoline attachment
 - ▶ *pre* filtering for *cgroup* granularity
 - ▶ abstraction matches the edges in our provenance graphs
 - ▶ confirmed in our macro benchmark
- robust: TOCTOU resistance
- portable

LSM: the most attractive, yet least-used basis

capture approach: eBPF has become dominant for capturing provenance

LSM hooks are the most attractive

- lower overhead
 - ▶ single attachment
 - ▶ more efficient trampoline attachment
 - ▶ *pre* filtering for *cgroup* granularity
 - ▶ abstraction matches the edges in our provenance graphs
 - ▶ confirmed in our macro benchmark
- robust: TOCTOU resistance
- portable

...yet system calls remain the norm

- even in open-source *capture agents*

because LSM hooks are:

- less known: kernel internals vs. documented interface
- harder to write security policies (e.g., hot patching)
- licensing: GPL-compatible only

Ongoing and next steps

- Which hooks are needed for Provenance exactly?
 - ▶ related works uses vastly different sets
 - ▶ none justifies its choice of captured events
 - identifying the set of LSM hooks to capture provenance
 - evaluating its coverage compared to system calls

Ongoing and next steps

- Which hooks are needed for Provenance exactly?
 - ▶ related works uses vastly different sets
 - ▶ none justifies its choice of captured events
 - identifying the set of LSM hooks to capture provenance
 - evaluating its coverage compared to system calls
- limitations of capturing from the kernel
 - ▶ blind to user-space memory (major attack vector)
 - is this really usefull for provenance?
 - ▶ we trust the kernel: not robust against kernel rootkits
 - can we capture provenance from the hypervisor without the VM knowing?

Ongoing and next steps

- Which hooks are needed for Provenance exactly?
 - ▶ related works uses vastly different sets
 - ▶ none justifies its choice of captured events
 - identifying the set of LSM hooks to capture provenance
 - evaluating its coverage compared to system calls
- limitations of capturing from the kernel
 - ▶ blind to user-space memory (major attack vector)
 - is this really usefull for provenance?
 - ▶ we trust the kernel: not robust against kernel rootkits
 - can we capture provenance from the hypervisor without the VM knowing?

Thank you! Questions?

Capture approaches for kernel telemetry

Capture approaches for kernel telemetry (● good, ◐ moderate, ○ poor)

location	approach	<i>portability</i>	<i>efficiency</i>	<i>visibility</i>	<i>safety</i>	<i>robustness</i>	example
user space	ptrace	◐	○	○	◐	○	ALASTOR [Datta et al., 2022]
	fs snapshot	◐	●	○	●	○	ARTISAN [Yu et al., 2024]
kernel space	integrated systems	○	◐	◐	●	◐	CLARION [Chen et al., 2021]
	out-of-tree kernel module	○	●	●	◐	◐	LTTng[LTTng 2005]
	eBPF	◐	●	●	●	◐	ProvBPF[Lim et al., 2021]

eBPF program types

Comparison of eBPF program types used for provenance capture layers

Program type	Attach type	TOCTOU resistance	Granularity					Stability	Kernel version $\geq$
			system calls	kernel function	user-space function	MAC operation	cgroup		
<i>LSM</i>	LSM_MAC	●	○	○	○	●	○	●	5.7
	LSM_CGROUP	●	○	○	○	●	●	●	6.0
<i>tracepoint</i>	tracepoint	◐	●	◐	○	○	○	●	4.7
<i>kprobe</i>	kprobe/kretprobe	◐	●	●	○	●	○	○	4.1
	uprobe/uretprobe	○	○	○	●	○	○	○	4.3
<i>tracing</i>	fentry/fexit	◐	●	●	○	●	○	○	5.5

Prevalence of system calls with user-space pointer arguments

Set of system calls	# system calls	# of those with a pointer that is	
	with ≥ 1 user-space pointer	file-related	socket-address
All system calls (Linux v6.14)	282/407 (69%)	96	8
$\in$ Falco attack signatures [Falco 2014]	30/41 (73%)	20	5
$\in$ Tracee attack signatures [Tracee 2020]	14/19 (74%)	9	1

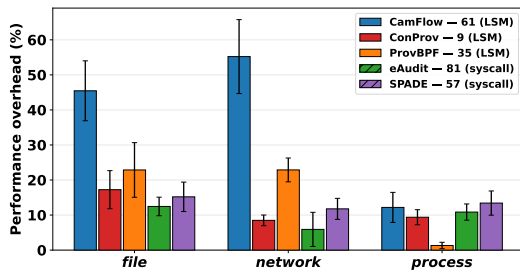
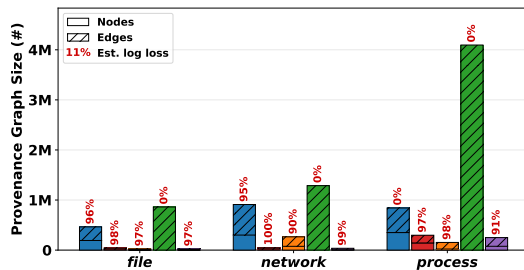
System calls referenced by Falco and Tracee attack signatures

Falco [Falco 2014]		Tracee [Tracee 2020]	
System calls	User-space pointer(s)	System calls	User-space pointer(s)
accept, accept4	upeer_sockaddr, upeer_addrln	connect	servaddr
bpf	attr	dup, dup2, dup3, mprotect, ptrace	()
chmod, fchmodat, open, openat	filename	execve, execveat	filename, argv, envp
connect	servaddr	finit_module	uargs
creat, mkdir, mkdirat, rmdir, unlink, unlinkat	pathname	init_module	umod, uargs
dup, dup2, dup3, fchmod, listen, ptrace, setns, setuid, socket, unshare, userfaultfd	()	mount	dev_name, dir_name, type, data
execve, execveat	filename, argv, envp	open, openat	filename
finit_module	uargs	open_by_handle_at	handle
init_module	umod, uargs	openat2	filename, how
link, linkat, rename, renameat, renameat2, symlinkat	oldname, newname	rename, renameat2	oldname, newname
openat2	filename, how	write	buf
recvfrom	ubuf, addr, addr_len	writev	vec
recvmsg, sendmsg	msg		
sendto	buff, addr		
symlink	old, new		

Provenance systems

Provenance system	Capture approach	Capture granularity		
		<i>pre</i>	<i>in</i>	<i>post</i>
SPADE [Gehani and Tariq, 2012] (2012)	AUDITD	system call	∅	event attribute
CamFlow [Pasquier et al., 2017] (2017)	LSM (patched LSM, Netfilter)	MAC operation network connection	kernel object kernel event	∅
WINNOWER [Hassan et al., 2018] (2018)	AUDITD	system call	∅	∅
TRACE [Irshad et al., 2021] (2021)	AUDITD	system call	∅	∅
ProvBPF [Lim et al., 2021] (2021)	eBPF (patched LSM)	MAC operation container	∅	∅
CLARION [Chen et al., 2021] (2021)	AUDITD	system call	container	∅
ConProv [Deng et al., 2024] (2024)	eBPF (<i>kprobe</i> , <i>tracepoint</i>)	system call capabilities socket connection file permission	container	∅
eAudit [Sekar et al., 2024] (2024)	eBPF (<i>tracepoint</i>)	system call	∅	∅

Macro benchmark of open-source *provenance systems*



Capture agents that could serve the capture layer of *provenance systems*

Capture agent	Capture approach	Capture granularity			Open source availability	Maintained
		<i>pre</i>	<i>in</i>	<i>post</i>		
AUDITD [auditd 2004] (2004)	kernel audit subsystem (<i>system call, LSM</i>)	system call MAC operation file watch	∅	event attribute	●	●
LTTng [LTTng 2005] (2005)	out-of-tree kernel module (<i>kprobe, tracepoint, programming libraries</i>)	system call kernel function kernel event C/C++, Java & Python program	∅	event attribute (incl. container)	●	●
Sysdig [sysdig 2014] (2014)	eBPF (<i>tracing, tracepoint</i>)	system call kernel event	∅	event attribute (incl. container)	●	●
Tracee [Tracee 2020] (2020)	eBPF (<i>LSM, tracing, kprobe, tracepoint</i>)	MAC operation system call kernel event network event	process, binary namespace container, pod <i>cgroup</i> , user	∅	●	●
Tetragon [tetragon 2022] (2022)	eBPF (<i>LSM, kprobe, tracepoint, uprobe</i>)	MAC operation system call kernel function user-space function	func. argument process, binary namespace container	event attribute	●	●

Macro benchmark of open-source *capture agents*

