A Study of Kernel Telemetry Options for Security-Oriented Provenance

COMPAS 2026 - Anglet

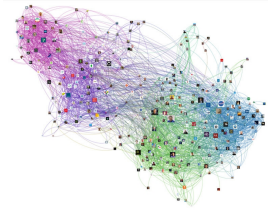
Paul Houssel[1,2] Nicolas Dejon[1] Olivier Levillain[2] Sylvie Laniepce[1]

[1]Orange Research, Châtillon, France
[2]Institut Polytechnique de Paris, SAMOVAR, Télécom SudParis, Palaiseau, France

1^{er} juillet 2026

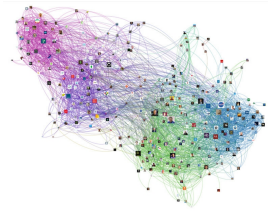
Provenance graphs [Carata et al., 2014]

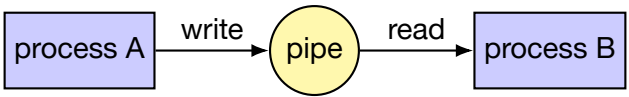
- model the whole-system behavior
- preserve the causality order across aggregated logs
- capture causal interactions between system *subjects* and *objects*:



Provenance graphs [Carata et al., 2014]

- model the whole-system behavior
- preserve the causality order across aggregated logs
- capture causal interactions between system *subjects* and *objects*:

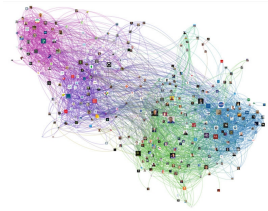


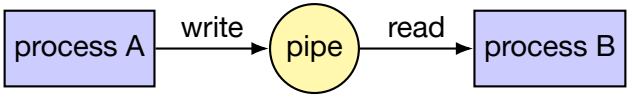


the `write`/`read` edges are causal events captured from kernel telemetry (e.g., system calls …)

Provenance graphs [Carata et al., 2014]

- model the whole-system behavior
- preserve the causality order across aggregated logs
- capture causal interactions between system *subjects* and *objects*:





the `write`/`read` edges are causal events captured from kernel telemetry (e.g., system calls …)

- Security use cases:
  - online threat detection on Linux servers [Song et al., 2025, Jiang et al., 2025, Blair et al., 2024]
  - forensics for security analysts [Pasquier et al., 2017, Li et al., 2022]

Challenges & the overlooked layer [Inam et al., 2023]

Provenance graphs are very large

- performance overhead
- availability
- storage complexity
- costly to analyze at scale
- portability
- integrity

Challenges & the overlooked layer [Inam et al., 2023]

Provenance graphs are very large

- performance overhead
- availability
- storage complexity
- costly to analyze at scale
- portability
- integrity

Research goal

Study the **capture layer** of these tools as it is the most crucial:

- *How* to capture? (capture approach)
- *Where* to capture? (capture location)
- Study which of these challenges these tools actually face

How to capture? – five kernel-telemetry approaches

Capture approaches for kernel telemetry (● good, ◗ moderate, ❍ poor).

location	approach	*efficiency*	*visibility*	*portability*	*safety*	*robustness*	example
user space	`ptrace`	❍	❍	◗	◗	❍	Alastor [Datta et al., 2022]
	fs snapshot	●	❍	◗	●	❍	Artisan [Yu et al., 2024]
kernel space	integrated systems	◗	◗	◗	●	◗	Clarion [Chen and Chen, 2021]
	out-of-tree kernel module	●	●	❍	◗	◗	LTTng [LTTng, 2005]
	eBPF	●	●	◗	●	◗	ProvBPF [Lim et al., 2021]

How to capture? – five kernel-telemetry approaches

Capture approaches for kernel telemetry (● good, ◗ moderate, ❍ poor).

location	approach	*efficiency*	*visibility*	*portability*	*safety*	*robustness*	example
user space	`ptrace`	❍	❍	◗	◗	❍	Alastor [Datta et al., 2022]
	fs snapshot	●	❍	◗	●	❍	Artisan [Yu et al., 2024]
kernel space	integrated systems	◗	◗	◗	●	◗	Clarion [Chen and Chen, 2021]
	out-of-tree kernel module	●	●	❍	◗	◗	LTTng [LTTng, 2005]
	eBPF	●	●	◗	●	◗	ProvBPF [Lim et al., 2021]

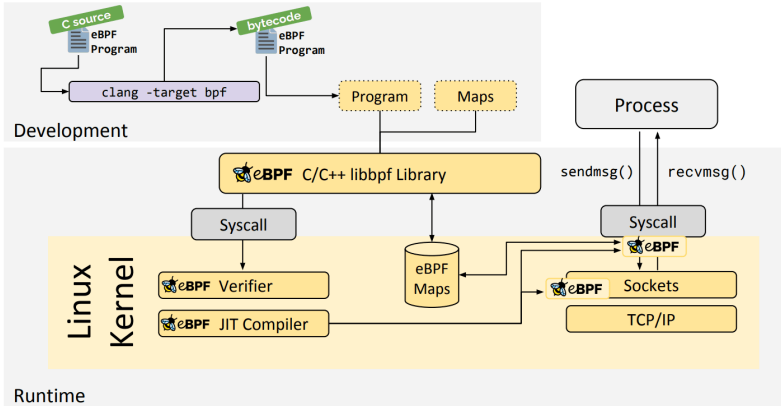
How to capture? – five kernel-telemetry approaches

Capture approaches for kernel telemetry (● good, ◗ moderate, ❍ poor).

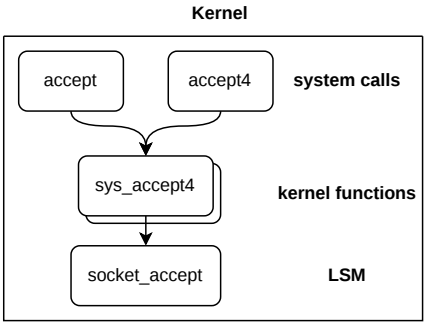
location	approach	*efficiency*	*visibility*	*portability*	*safety*	*robustness*	example
user space	`ptrace`	❍	❍	◗	◗	❍	Alastor [Datta et al., 2022]
	fs snapshot	●	❍	◗	●	❍	Artisan [Yu et al., 2024]
kernel space	integrated systems	◗	◗	◗	●	◗	Clarion [Chen and Chen, 2021]
	out-of-tree kernel module	●	●	❍	◗	◗	LTTng [LTTng, 2005]
	eBPF	●	●	◗	●	◗	ProvBPF [Lim et al., 2021]

What is eBPF?

eBPF runs sandboxed programs inside the kernel at runtime. A program is statically verified, JIT-compiled, attached to a kernel hook, and exchanges data with user space through maps:



Kernel telemetry capture locations



eBPF program types

Comparison of eBPF program types used for provenance capture layers.

Program type	Attach type	Capture location			Kernel version ≥
		system calls	kernel function	MAC operation (LSM)	
LSM	`LSM_MAC`	❍	❍	●	5.7
kprobe	`kprobe/kretprobe`	●	●	●	4.1
tracing	`fentry/fexit`	●	●	●	5.5

Kprobe eBPF attachment via `INT3` breakpoint

(1) Original function

```
some_kernel_func:
  PUSH RBP          ; 1st instr
  MOV  RBP, RSP
  ...
  RET
```

Kprobe eBPF attachment via INT3 breakpoint

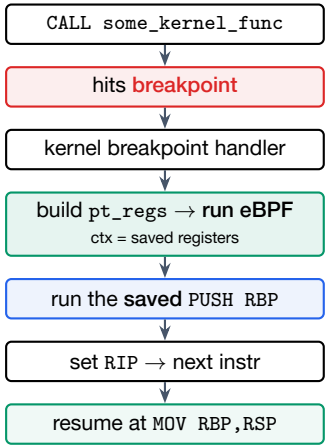
(1) Original function

```
some_kernel_func:
  PUSH RBP          ; 1st instr
  MOV  RBP, RSP
  ...
  RET
```

(2) After attach: 1st instr → breakpoint

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

The original `PUSH RBP` is saved aside so it can still be run later.



Key idea: the breakpoint detours into the handler (where eBPF runs); the displaced instruction is then run from its saved copy, and execution resumes

fentry eBPF attachment via `ftrace` trampoline

(1) Compiled with `-mfentry`

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

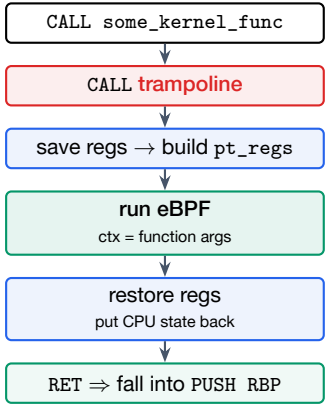
fentry eBPF attachment via `ftrace` trampoline

(1) Compiled with `-mfentry`

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

(2) After attach: NOPs → CALL

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



Key idea: the `fentry` NOP pad is replaced by a direct `CALL` to a trampoline: no breakpoint

Robustness: system-call tracing is exposed to TOCTOU [Guo and Zeng, 2021, Guo and Zeng, 2022]

Time-Of-Check to Time-Of-Use

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

- TOC: tracing program dereferences the user-space pointer
- TOU: kernel dereferences the same user-space pointer (attacker swaps it in between)

Robustness: system-call tracing is exposed to TOCTOU [Guo and Zeng, 2021, Guo and Zeng, 2022]

Time-Of-Check to Time-Of-Use

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

- TOC: tracing program dereferences the user-space pointer
- TOU: kernel dereferences the same user-space pointer (attacker swaps it in between)

→ *LSM* is deep in the execution flow ⇒ TOCTOU-resistant.

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

Portability: which capture interfaces are stable?

across LTS versions (vs. 5.15)

Interfaces	v5.15	v6.8	v6.17
# kernel function	54k	+12k/-4k	+21k/-6k
# ABI changes	–	3k	4k
# system call	345	+13/-1	+22/-1
# ABI changes	–	=	=
# LSM	203	+20/-4	+43/-8
# ABI changes	–	27	33

Portability: which capture interfaces are stable?

across LTS versions (vs. 5.15)

Interfaces	v5.15	v6.8	v6.17
# kernel function	54k	+12k/-4k	+21k/-6k
# ABI changes	–	3k	4k
# system call	345	+13/-1	+22/-1
# ABI changes	–	=	=
# LSM	203	+20/-4	+43/-8
# ABI changes	–	27	33

across architectures & flavors (vs. amd64 v5.15)

		architecture			configuration flavor			
Interfaces	*amd64*	*arm64*	*ppc64el*	*riscv64*	*aws*	*azure*	*gcp*	*lowlat.*
# kernel function	54k	+14k/-9k	+7k/-11k	+3k/-14k	+1k/-96	+3k/-3k	+2k/-0	+1k/-51
# ABI changes	–	119	134	116	=	8	=	=
# system call	345	+2/-43	+22/-23	+2/-49	=	=	=	=
# ABI changes	–	=	=	=	=	=	=	=
# LSM	203	=	=	=	=	=	=	=
# ABI changes	–	=	=	=	=	=	=	=

Capture location impacts the performance overhead

- LSM and fentry programs have the lowest performance overhead thanks to trampoline attachments over kprobes' CPU interrupts

Capture location impacts the performance overhead

- LSM and fentry programs have the lowest performance overhead thanks to trampoline attachments over kprobes' CPU interrupts
- LSM do not require double attachment, in contrast to system calls [Craun and Williams, 2025]:
  - **entry**: read arguments (gone once the call returns)
  - **exit**: read the verdict (drop denied/failed events)

  $\rightarrow \sim 50\%$ additional overhead

Capture location impacts the performance overhead

- LSM and fentry programs have the lowest performance overhead thanks to trampoline attachments over kprobes' CPU interrupts
- LSM do not require double attachment, in contrast to system calls [Craun and Williams, 2025]:
  - **entry**: read arguments (gone once the call returns)
  - **exit**: read the verdict (drop denied/failed events)

  $\rightarrow \sim 50\%$ additional overhead
- LSM is deeper in the execution flow: lower abstraction level which directly corresponds to the causality events. E.g., to cover the causal event of file opening:
  - capturing system calls requires instrumenting multiple system calls (`open`/`openat`/`creat`…) and additional filtering
  - LSM cover file opening with a single hook

LSM are even more efficient when requiring granular capture

Production needs granularity (e.g. trace one container on a multi-tenant host).

- *pre* – filter before capture (*cgroup* attach)
- *in* – filter inside the program
- *post* – filter in user space

LSM are even more efficient when requiring granular capture

Production needs granularity (e.g. trace one container on a multi-tenant host).

- *pre* – filter before capture (*cgroup* attach)
- *in* – filter inside the program
- *post* – filter in user space

pre filtering is cheapest, and only *LSM* programs can do it (*cgroup* attachment).

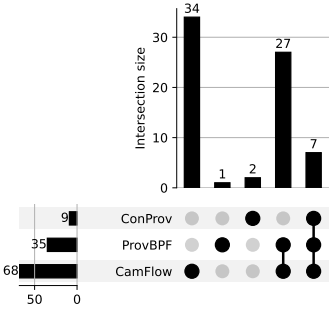
LSM: the most attractive, yet least-used basis

- **Capture approach**: eBPF now dominant, yet several systems still build on auditd despite its limits
- **Capture location**: *LSM* interfaces are the most attractive basis but are the least used:
  - Lower overhead
    - single attachment required, instead of two
    - more efficient trampoline attachment
    - *pre* filtering for cgroup-granularity
    - abstraction matches the causality events of interest for provenance capture
  - Robust: TOCTOU resistance of *LSM* vs. system-call tracing
  - Portable: acceptable ABI stability across kernel versions, architectures, and flavors

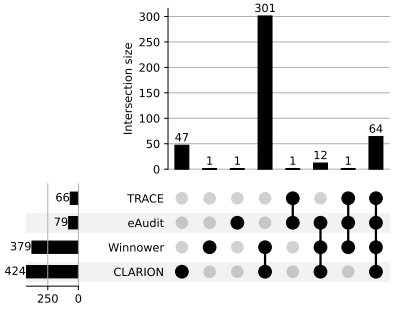
No consensus on the *capture model*

The **capture model** = the set of interfaces chosen to record causality events.

LSM capture models



system-call capture models



Each tool adopts a **vastly different** capture model, justified only implicitly, none verifies completeness.

Ongoing work

- An eBPF capture tool on *LSM* interfaces producing provenance graphs that are **low-overhead, lossless, robust, and portable**
  - a combination no existing tool achieves
- A minimal subset of *LSM* interfaces covering all causal events for security-oriented provenance
  - enough for PIDS and graph-query-based forensics

References I

Blair, W., Araujo, F., Taylor, T., and Jang, J. (2024).
Automated Synthesis of Effect Graph Policies for Microservice-Aware Stateful System Call Specialization.
In *2024 IEEE Symposium on Security and Privacy (SP)*, pages 4554–4572. IEEE.

Carata, L., Akoush, S., Balakrishnan, N., Bytheway, T., Sohan, R., Seltzer, M., and Hopper, A. (2014).
A primer on provenance.
Communications of the ACM.

References II

Chen, X. and Chen, Y. (2021).
CLARION: Sound and Clear Provenance Tracking for Microservice Deployments.
In *Proceedings of the 30th USENIX Security Symposium (SEC)*, pages 3989–4006. USENIX Association.

Craun, M. and Williams, D. (2025).
Pairwise bpf programs should be optimized together.
In *Proceedings of the 3rd Workshop on eBPF and Kernel Extensions*.

Datta, P., Polinsky, I., Inam, M. A., Bates, A., and Enck, W. (2022).
ALASTOR: Reconstructing the Provenance of Serverless Intrusions.
In *Proceedings of the 31st USENIX Security Symposium (SEC)*, pages 2443–2460. ACM.

References III


Falco (2014).
Falco: open source security tool for containers, kubernetes and cloud.


Guo, R. and Zeng, J. (2021).
Phantom Attack: Evading System Call Monitoring.
DEF CON 29.


Guo, R. and Zeng, J. (2022).
Trace Me If You can: BypassingLinux Syscall Tracing.
DEF CON 30.

References IV

Inam, M. A., Chen, Y., Goyal, A., Liu, J., Mink, J., Michael, N., Gaur, S., Bates, A., and Hassan, W. U. (2023).
SoK: History is a Vast Early Warning System: Auditing the Provenance of System Intrusions.
In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 2620–2638. IEEE.

Jiang, B., Bilot, T., Madhoun, N. E., Agha, K. A., Zouaoui, A., Iqbal, S., Han, X., and Pasquier, T. (2025).
ORTHRUS: Achieving High Quality of Attribution in Provenance-based Intrusion Detection Systems.
In *Proceedings of the 34th USENIX Conference on Security Symposium (SEC)*, pages 7173–7192. ACM.

References V


Li, J., Zhang, R., Liu, J., and Liu, G. (2022).
LogKernel: A Threat Hunting Approach Based on Behaviour Provenance Graph and Graph Kernel Clustering.
Security and Communication Networks, 2022(1):4577141.


Lim, S. Y., Stelea, B., Han, X., and Pasquier, T. (2021).
Secure Namespaced Kernel Audit for Containers.
In *Proceedings of the ACM Symposium on Cloud Computing*, pages 518–532. ACM.


LTTng (2005).
LTTng: an open source tracing framework for Linux.

References VI

Pasquier, T., Han, X., Goldstein, M., Moyer, T., Eyers, D., Seltzer, M., and Bacon, J. (2017).
Practical whole-system provenance capture.
In *Proceedings of the 2017 Symposium on Cloud Computing (SoCC)*. ACM.

Song, T., Organokov, M., Gulikers, L., Grassi, G., Carofiglio, G., and Meo, M. (2025).
Advancing Cloud-Native Cyber Threat Detection with Graph-Based Feature Engineering.
In *Proceedings of the 41st International Conference on Data Engineering (ICDE)*, pages 4291–4297. IEEE Computer Society.

Tracee (2020).
Tracee: Linux Runtime Security and Forensics using eBPF.

References VII


Yu, L., Ye, Y., Zhang, Z., and Zhang, X. (2024).
Cost-effective Attack Forensics by Recording and Correlating File System Changes.
In *Proceedings of the 33rd USENIX Security Symposium (SEC)*, pages 1705 – 1722. ACM.