Towards Provenance for Cybersecurity in Cloud-Native Production Infrastructure

1st Paul R. B. Houssel
Institut Polytechnique de Paris
SAMOVAR, Télécom SudParis
91120 Palaiseau, France
Orange Innovation, France
paul.houssel@orange.com

2nd Sylvie Laniepce
Orange Innovation, France
s.laniepce@orange.com

3rd Olivier Levillain
Institut Polytechnique de Paris
SAMOVAR, Télécom SudParis
91120 Palaiseau, France
olivier.levillain@telecom-sudparis.eu

***Abstract*—System provenance models the interactions between system subjects and objects, enabling post-mortem and root-cause analyses of cyberattacks. Despite numerous contributions to provenance systems, there remains little consensus on the reliability of existing telemetry collection methods. Linux Security Module (LSM) interfaces present a promising alternative thanks to their inherent stability and safety for production environments. However, since LSM do not capture the full granularity of system calls, it is unclear whether they can support the creation of sound provenance graphs. In this work, we study the evolution of these kernel interfaces and their coverage.**

***Index Terms*—Provenance, Linux Security Module, eBPF**

I. Context and Research Objectives

System provenance captures relationships among subjects, objects, and transformations, establishing causal links between system events [1], [2]. By modeling system behavior as a graph, provenance supports forensics and root cause analysis. To construct these graphs, telemetry may be collected from various levels in a Linux system. In user space, `ptrace` enables system call monitoring but suffers from high overhead, limited visibility, and has shown to be vulnerable to privilege escalation attacks in the past [3], [4]. In contrast, kernel-level monitoring tools such as *ftrace* [5] and *auditd* [6] offer lower overhead, improved security, and full visibility. However, these methods require kernel source modifications, resulting in lengthy peer review and community approval. Moreover, many production systems are hesitant to upgrade kernel versions due to compatibility constraints. Collection agents deployed as eBPF programs or kernel modules offer a more flexible alternative as they allow loading programs in the kernel, attaching to security and tracing interfaces.

Existing provenance systems leverage different kernel interfaces, for example *Clarion* [7] integrates *auditd* and Netfilter, while *ConProv* [8] combines kprobes, tracepoints, and Linux Security Module (LSM) hooks while *eAudit* [9] and *SysFlow* [10] rely solely on tracepoints.

Among available kernel instrumentation mechanisms, tracepoints and LSM hooks provide static interfaces designed for tracing and security policy enforcement at predefined locations, while kprobes offer a dynamic attachment to any kernel function. By probing telemetry in the kernel, all these solutions better align with modern production environments, where a common kernel executes various cloud-native microservices. These infrastructures, where each container runs within its own cgroup, demand fine-grained targeted collection which can be achieved by attaching LSM hooks directly to those cgroups [11]. Nevertheless, challenges remain in managing telemetry volume, ensuring utility, and mitigating overhead. Thus, there is a need to reconsider the telemetry collection approach. LSM hooks, traditionally employed by frameworks such as *SELinux* [12] and *AppArmor* [13] intercept Access Control (AC) operations. Each hook corresponds to a specific event (e.g., file access or socket creation), enabling attached LSMs, implemented as eBPF programs or kernel modules, to enforce security policies and trace telemetry. With AC-operation granularity, LSM hooks have shown to offer the stability essential for provenance systems. Indeed, kernel evolutions might introduce new attack paths that could be exploited in dynamic hooks if not correctly updated. Furthermore, because LSM hooks integrate deeply into the kernel's function call chain, they provide the most raw telemetry, rendering detection and audit processes agnostic to user-space application semantics. It also allows LSM hooks, unlike kprobes and tracepoints, to be resistant to Time-of-Check-to-Time-of-Use (TOCTOU) race condition attacks [14]–[16], which exploit delays between reading and verifying system call arguments. Additionally, by covering solely LSM operations rather than all system calls, this selective approach typically results in lower log volume while still in theory covering all AC-related events. Previous work, such as *CamFlow* [17] and *ProvBPF* [18] leverage LSM hooks for provenance collection, unfortunately without fully evaluating their coverage and stability.

II. Research Plan

To assess the suitability of LSM hooks for production provenance systems, this paper investigates two key research questions: 1) Do LSM hooks provide sufficient system coverage to construct sound provenance graphs?; 2) Compared to system calls, do LSM hooks offer greater stability across kernel versions? Preventing a provenance model built upon them to be constantly revised. This preliminary work, will

identify the need for additional LSM operations if gaps are identified, serving as a foundation to understand the validity of using LSM for our future provenance collection. To validate it, we will consider real-world attack scenarios, with a particular focus on ransomware. Ransomware presents a compelling case that production systems face and which generates high-intensity system activity, resulting in a high volume of kernel-level telemetry. Finally, we aim to generalize our findings to other attack types, refining our approach by adding benign noise and more diverse attacks.

III. Preliminary Results and Future Work

A. System Call and LSM Interface Relationships

We propose a static analysis method to map system calls to their related LSM hooks. Leveraging the flow graph generator *GNU cflow* [19] and the code-navigation tool *cscope* [20], our approach analyzes the Linux kernel source to construct function call graphs linking system call macros to the LSM hooks they trigger. Our method, without compilation preprocessing, is agnostic to the compilation configurations. In sum, we: 1) Parse `include/linux/syscalls.h` to identify all system calls of the current kernel version; 2) Locate across all C files the definition of these system calls defined as macros, using *cscope*. E.g., `open` is defined in `fs/open.c`; 3) For each identified source file, *GNU cflow* generates a function call graph, defining as a directed graph, the function flow from the system call definition macro; 4) Use Depth-First Search (DFS) traversal to find all paths connecting a system call to an LSM hook. If no relationship is found, we recursively analyze the files referencing the functions located as leaves in the generated graph, until reaching a recursion depth.

Our results map system calls to LSM hooks, correlating each security operation with the system calls that triggers it. In Linux 6.13, with a recursion depth of 5, we identify only 63.70% of the LSM hooks for which at least one related system call is identified. 69.15% of the system calls were found to lead to at least one LSM hook. Our manual review indicates that our approach does not capture all existing relationships. This limitation does not stem from the recursion depth, but from the static analysis tool's limitations in handling indirect function calls. Future work will explore alternative tools, such as the GCC plugin *Kayrebt* [21] to generate function activity diagrams and a dynamic analysis approach with the *Trinity* system call fuzzer [22].

B. System Call and LSM Interface stability

We assess the stability of LSM hooks and system calls by analyzing the changes made to their Application Binary Interface (ABI) across Linux kernel versions. We consider two minor versions per major release since LSM introduction in version 2.6, starting from 2.6.12. For each version, we extract system calls from `include/linux/syscalls.h` and LSM interfaces from `include/linux/lsm_hooks.h` and `include/linux/security.h` for pre 4.12 versions. We then track additions and removals of LSM hooks and system call functions and their arguments.

TABLE I
Evolution of LSM and system call ABI, showing the number of added ('+'), removed ('-'), and initial ('=') interface functions and argument modifications since the previous release.

Linux version	Release date	LSM hooks	Argument changes	System calls	Argument changes
2.6.12	2005-06-17	=131	–	=251	–
2.6.30	2009-06-09	+72/-22	19	+79/-0	8
3.1	2011-10-24	+17/-14	8	+17/-1	15
3.12	2013-11-03	+12/-4	12	+13/-0	16
4.1	2015-06-21	+7/-2	3	+9/-0	4
4.12	2017-07-02	+8/-3	19	+10/-0	5
5.1	2019-05-05	+23/-4	24	+37/-0	27
5.12	2021-04-25	+16/-2	26	+15/-1	13
6.1	2022-12-11	+10/-2	10	+8/-1	0
6.12	2024-11-17	+33/-8	20	+14/-1	7
6.13	2025-01-19	+6/-4	1	+4/-0	0

Table. I summarizes the evolution of LSM hooks and system call ABI's over time. Initially, there were 131 LSM interfaces, and although the number has grown to 270 by version 6.13, the evolution process involves frequent removals as well as additions. The changes to LSM hooks functions can be categorized as follows: 1) Introduction of new functions to support emerging security operations. E.g., between 2.6.12 and 2.6.30, `ptrace_traceme` and `ptrace_may_access` hooks were added following the introduction of the `ptrace` system call; 2) Renaming for enhanced context, reflecting changes in their visibility (the data they access to) even though the covered AC operations semantics remain. E.g., in version 6.13, the hook `inode_getsecid` was renamed to `inode_getlsmprop` to better reflect the broader context of data gathering across all registered LSMs; 3) Removal of obsolete hooks when their corresponding system calls are eliminated. Similarly, the arguments of these hooks are frequently adjusted by either converting an argument to a constant, renaming it, or adding additional arguments for clarity. We observe that new system calls integrate with existing LSM hooks (e.g., the `file_open` hook is now triggered by both `openat` and `openat2`). While argument adjustments are common, the removal of system calls is relatively rare, and limited to cases of renaming, type changes, or conversion to constants. LSM hooks are not as stable as initially thought, and their interface changes as much as those of system calls.

IV. Conclusion

Our study demonstrates that while LSM hooks capture only key system calls, they inherently cover all AC-related operations and adapt to evolving system-call interfaces. It still remains unclear if they provide a stabler basis for provenance models and cover all critical events. Future work shall focus on analyzing the mapping between system calls and LSMs to assess whether additional hooks are needed. A quantitative evaluation of ABI changes is necessary to fully understand their stability compared to those of system calls.

REFERENCES

[1] L. Carata, S. Akoush, N. Balakrishnan, T. Bytheway, R. Sohan, M. Seltzer, and A. Hopper, "A primer on provenance," *Communications of the ACM*, vol. 57, no. 5, pp. 52–60, May 2014.

[2] M. A. Inam, Y. Chen, A. Goyal, J. Liu, J. Mink, N. Michael, S. Gaur, A. Bates, and W. U. Hassan, "SoK: History is a Vast Early Warning System: Auditing the Provenance of System Intrusions," in *2023 IEEE Symposium on Security and Privacy (SP)*. San Francisco, CA, USA: IEEE, May 2023, pp. 2620–2638. [Online]. Available: https://ieeexplore.ieee.org/document/10179405/

[3] E. Connor, T. McDaniel, J. M. Smith, and M. Schuchard, "PKU Pitfalls: Attacks on PKU-based Memory Isolation Systems," in *Proceedings of the 29th USENIX Security Symposium (USENIX Security 20)*, Boston, MA, USA, 2020, pp. 1409–1426.

[4] "CVE-2019-13272," Jul. 2019. [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2019-13272

[5] "ftrace - Function Tracer — The Linux Kernel documentation." [Online]. Available: https://www.kernel.org/doc/html/v6.14-rc6/trace/ftrace.html

[6] "auditd(8) - Linux manual page." [Online]. Available: https://www.man7.org/linux/man-pages/man8/auditd.8.html

[7] X. Chen and Y. Chen, "CLARION: Sound and Clear Provenance Tracking for Microservice Deployments," in *Proceedings of the 30th USENIX Security Symposium (USENIX Security 21)*, Virtual Event, 2021.

[8] Q. Deng, Y. Zhang, Z. Xu, Q. Tan, and Y. Zhang, "ConProv: A Container-Aware Provenance System for Attack Investigation," in *Annual Computer Security Applications Conference (ACSAC) 2025 Proceedings*. Honolulu, Hawaii, USA: IEEE, 2024.

[9] R. Sekar, H. Kimm, and R. Aich, "eAudit: A Fast, Scalable and Deployable Audit Data Collection System," in *2024 IEEE Symposium on Security and Privacy (SP) Proceedings*, May 2024, pp. 3571–3589, iSSN: 2375-1207.

[10] T. Taylor, F. Araujo, and X. Shu, "Towards an Open Format for Scalable System Telemetry," in *2020 IEEE International Conference on Big Data (Big Data)*. Atlanta, GA, USA: IEEE, Dec. 2020, pp. 1031–1040. [Online]. Available: https://ieeexplore.ieee.org/document/9378294/

[11] S. Fomichev. (2022, Jun.) [patch bpf-next v11 03/11] bpf: per-cgroup lsm flavor. Linux Kernel Mailing List. [Online]. Available: https://lore.kernel.org/all/20220628174314.1216643-4-sdf@google.com/

[12] S. Smalley, C. Vance, and W. Salamon, *Implementing SELinux as a Linux security module.* NAI Labs Report, 2001.

[13] "AppArmor." [Online]. Available: https://apparmor.net/

[14] R. N. M. Watson, "Exploiting concurrency vulnerabilities in system call wrappers," in *Proceedings of the first USENIX workshop on Offensive Technologies*, ser. WOOT '07. USA: USENIX Association, Aug. 2007, pp. 1–8.

[15] R. Guo and J. Zeng, "Trace Me If You can: BypassingLinux Syscall Tracing," DEF CON 30, Las Vegas, US, 2022.

[16] ——, "Phantom Attack: Evading System Call Monitoring," DEF CON 29, Las Vegas, US, 2021.

[17] T. Pasquier, X. Han, M. Goldstein, T. Moyer, D. Eyers, M. Seltzer, and J. Bacon, "Practical whole-system provenance capture," in *Proceedings of the 2017 Symposium on Cloud Computing*. Santa Clara, California, US: ACM, Sep. 2017, pp. 405–418.

[18] S. Y. Lim, B. Stelea, X. Han, and T. Pasquier, "Secure Namespaced Kernel Audit for Containers," in *Proceedings of the ACM Symposium on Cloud Computing*. Seattle, WA, USA: ACM, Nov. 2021, pp. 518–532.

[19] "Cflow - GNU Project - Free Software Foundation." [Online]. Available: https://www.gnu.org/software/cflow/

[20] "Cscope Home Page." [Online]. Available: https://cscope.sourceforge.net/

[21] L. Georget, F. Tronel, and V. V. T. Tong, "Kayrebt: An activity diagram extraction and visualization toolset designed for the Linux codebase," in *2015 IEEE 3rd Working Conference on Software Visualization (VISSOFT)*. Bremen, Germany: IEEE, Sep. 2015, pp. 170–174.

[22] D. Jones, "Trinity: A Linux system call fuzzer." [Online]. Available: https://github.com/kernelslacker/trinity