sysfilter: System Call Filtering Beyond the Lab

Nicholas DeMarinis, Di Jin, and Vasileios P. Kemerlis
Brown University

***Abstract*—OS kernels expose their entire system call (syscall) API to every process, although most applications need only a fraction of it. `sysfilter` is an open-source binary analysis framework that automatically extracts the set of syscalls a commodity x86-64 Linux application can issue and confines the application to that set with `seccomp-BPF`. It needs no source code or symbols, and does not rely on dynamic tracing. Developed under ONR's Total Platform Cyber Protection (TPCP) program, `sysfilter` has moved beyond the lab since its release at RAID 2020: it was selected for transition to practice, Star Lab (then a Wind River subsidiary, now part of Mercury Systems) uses it in an embedded Linux-hardening product, and NEC cites it as prior art in a granted patent. Our RAID paper has about 160 citations from 154 distinct works. Of these, 92% come from outside our group, including 13 at IEEE S&P, USENIX Security, ACM CCS, and NDSS. Other groups use `sysfilter` as the binary-only baseline for new syscall filtering systems, reuse its analyses and data, or run it in student research. This report summarizes the artifact and documents its impact.**

1. Introduction

Applications reach the kernel almost exclusively through the syscall API. Linux v7.3 exposes 386 syscalls on x86-64, up from 347 in v5.5, yet in our 2020 study the median C/C++ binary in Debian `sid` could reach only 90 of them [1]. The kernel still grants unrestricted access to all syscalls: a compromised process therefore inherits OS services that its developers never intended it to use, and every unused syscall is extra kernel attack surface for privilege escalation. Linux has enforced per-process syscall allowlists through `seccomp-BPF` since v3.5, but writing such lists by hand is laborious and fragile, so only a few projects (e.g., Chrome, Firefox, OpenSSH) ship them. Policies derived from dynamic tracing are incomplete by construction and break applications on paths the traces did not cover.

`sysfilter` closes this gap. Given only an application's *binary*, it statically computes a safe but tight over-approximation of the developer-intended syscall set and installs a `seccomp-BPF` filter that enforces exactly that set. We published `sysfilter` at RAID 2020 [1] and released it as open source.

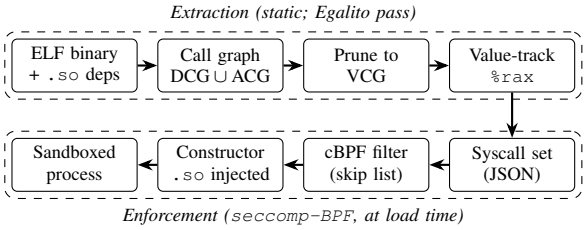


Figure 1. The `sysfilter` pipeline.

Its impact since then includes:

- **Industry and Government:** adoption by Star Lab, then a Wind River subsidiary and now part of Mercury Systems; transition to practice under ONR's TPCP program, funding its productization; prior art in NEC's US 11,733,989 B2 (§3.1).
- **Research:** about 160 citations from 154 distinct works, 141 of them by groups outside ours, spanning the top security venues as well as systems, software engineering, and usability venues (Table 1).
- **Binary-only Baseline:** independent groups run `sysfilter` as the point of comparison for new syscall-filtering systems and reuse its analyses, filters, and data (§3.2).
- **Education:** B.Sc. theses at TU Delft; a Ph.D. dissertation and student projects at Brown; coursework and Brown Systems Week (§3.3).

2. The Artifact

`sysfilter` (https://gitlab.com/brown-ssl/sysfilter) has two components (Fig. 1). The *extraction* tool, about 2.5 KLOC of C/C++, is a pass over the IR of the Egalito binary recompiler [2] (its original implementation was developed by K. Williams-King, a co-author of the RAID paper [1]). The *enforcement* tool, about 150 LOC of Python, compiles a syscall set into a `seccomp-BPF` filter and attaches it to the binary. Both work on stripped x86-64 ELF binaries produced by modern GCC and LLVM toolchains, including programs that mix languages and compilers.

Extraction. `sysfilter` resolves the program's shared-library dependencies recursively, along with any user-specified `dlopen`'ed objects and the `glibc` NSS modules. It recovers exact function boundaries from `.eh_frame`, so stripped binaries disassemble precisely.

It then builds a call graph across all objects. The *direct* call graph (DCG) follows direct calls, tail calls, and PLT bindings from the entry point and from every initializer and finalizer. To cover indirect calls soundly without points-to analysis, `sysfilter` relies on the fact that, in position-independent code (PIC), every absolute code pointer requires a relocation. Functions whose addresses appear in relocations are therefore *address-taken* (AT), and the functions reachable from them form the address-taken call graph (ACG). `sysfilter` then prunes DCG ∪ ACG into a *vacuumed call graph* (VCG) by iteratively removing AT functions whose addresses are taken only in unreachable code, or, when symbols are available, only in unreferenced function-pointer/dispatch tables. Finally, a use-define analysis resolves `%rax` at each reachable `syscall` instruction. If a value cannot be resolved, it aborts/warns instead of emitting an incomplete policy.

Enforcement. The enforcement tool compiles the set into a classic-BPF program. The program first checks the architecture, which blocks `int $0x80` and x32 confusion attacks, and then searches the allowlist linearly or via a skip list. It injects a small constructor library with `patchelf`, which sets `no_new_privs` and installs the filter on all threads before `main` runs. Union and hierarchical modes handle programs that `execve` one another.

Original Results. On 411 binaries (e.g., Coreutils, Nginx, Redis, SQLite, FFmpeg, MariaDB, Vim), sandboxed programs passed all of about 38.5K tests from their own suites. The run-time overhead was at most 1% on SPEC CINT2006 and at most 6% (Nginx) and 3% (Redis) with JIT-compiled filters. `sysfilter` analyzed 30,398 of the 33,829 C/C++ binaries in Debian `sid` (91.3%), with a median of about 30 s per binary, and showed which of 23 kernel CVEs remain reachable from each hardened binary [1].

3. Impact

3.1. Industry and Government

Star Lab / Wind River / Mercury Systems. Since November 2021, `sysfilter` has been part of Star Lab's Linux-hardening product, Kevlar Embedded Security, where it reduces the attack surface of embedded Linux systems. Star Lab was then a subsidiary of Wind River. (Star Lab's 2023 whitepaper describes Kevlar's syscall filtering in terms that match `sysfilter`'s design [3].) Filters are generated "entirely at build time" by static binary analysis that finds syscalls "hiding in rarely exercised code paths such as error handling logic." That analysis includes "a pruning phase to eliminate system calls detected in dependent libraries but not actually used by the program." The resulting allowlists are enforced via *seccomp* when each service starts. Kevlar ships pre-generated filters for system services on x86-64 and ARM64, and it provides build-time analysis of user applications within the root-filesystem build [3], [4]. The whitepaper also notes the `execve` inheritance issue that `sysfilter`'s union and hierarchical modes address.

In May 2025, Mercury Systems acquired Star Lab for its secure-processing portfolio for aerospace and defense [5].

ONR TPCP and Transition to Practice. `sysfilter` was developed under the Office of Naval Research (ONR) Total Platform Cyber Protection (TPCP) program, which targeted late-stage complexity reduction and hardening of legacy Navy software [6]. `sysfilter`, together with Egalito [2] and Nibbler [7], was then selected for transition to practice. ONR's METIS contract ("Matured and Enhanced TPCP Technologies for Improved Security") funded GrammaTech, as ONR's transition contractor, to make the selected university prototypes deployable [8]. For `sysfilter`, this transition effort spanned 2.5 years (October 2021 to June 2024) of engineering for deployment: continuous integration with regression, robustness, and coverage testing; security scanning of the codebase; binary packages for Ubuntu 20.04; integration with the contractor's binary-analysis toolchain; and versioned 1.0 (Sept 2023) and 1.1 (Dec 2023) releases.

Patents and Industrial Research. NEC Laboratories Europe's granted patent on automated syscall sealing for unikernel images (US 11,733,989 B2) lists `sysfilter` as prior art [9]. Researchers at Google, Amazon Web Services, IBM Research, and Red Hat cite `sysfilter`'s skip-list filter compilation as an established seccomp optimization [10]. Industrial researchers at Orange [11] and Bloomberg [12], [13] have also built on or cited it.

3.2. Research Impact

Citation Analysis. As of October 2026, Google Scholar lists 161 entries citing `sysfilter`. After merging duplicate entries for the same work (e.g., a preprint and its published version), 154 distinct works remain. Table 1 classifies them by venue and year. Only 13 of them (8%) share an author with `sysfilter` or are theses from our group—so 141 come from other groups. Citations have grown every year since release. They come from the top four security venues, with 13 independent works at IEEE S&P, USENIX Security, ACM CCS, and NDSS [14]–[20]. They also come from systems venues (ASPLOS, OSDI, USENIX ATC, DSN, Middleware), software-engineering and PL venues (ICSE, ASE, ISSTA, OOPSLA), usability (SOUPS), and mobile computing (MobiCom). The works that cite `sysfilter` have themselves received more than 2,000 citations.

Community Use. Citing works use `sysfilter` in three ways. *(i) A baseline* that new syscall-filtering systems must beat [11], [14], [21]–[27]. *(ii) A building block*, whose call-graph analysis, address-taken analysis, skip-list filters, CVE-to-syscall list, or extraction engine others reuse [10], [11], [14], [28], [29]. *(iii) Prior art* for syscall filtering and attack-surface reduction, cited in work on containers, microservices, and Kubernetes [19], [20], [30]–[43]; application and binary hardening [12], [13], [15], [16], [44]–[57]; language runtimes such as Node.js, Deno, PHP, and Go [22], [58]–[61]; embedded and IoT systems [62]–[64]; eBPF and kernel extensions [10], [65], [66]; compartmentalization and interposition [17], [18], [67]–[71]; confidential VMs [72],

TABLE 1. DISTINCT WORKS CITING SYSFILTER, BY VENUE TYPE AND YEAR. (DATA FROM GOOGLE SCHOLAR, OCTOBER 2026; DUPLICATE ENTRIES ARE MERGED.)

	'20	'21	'22	'23	'24	'25	'26*	n.d.	All
Top-4 security‡	1	1	3	2	6	2	1	–	16
Other security	1	1	2	3	2	5	1	–	15
Systems/SE/PL/other	–	2	2	7	8	6	5	–	30
Journals	–	–	4	4	6	11	7	–	32
Workshops/posters	–	3	1	1	2	3	–	1	11
Theses	–	2	5	7	10	8	2	3	37
Preprints/patents	–	1	3	1	1	2	5	–	13
Total	2	10	20	25	35	37	21	4	**154**

*Through October 2026. ‡IEEE S&P, USENIX Security, ACM CCS, NDSS. 13 of the 154 works (3 of the 16 top-4) are by sysfilter authors or our group.

[73]; OS compatibility layers [74]; web platforms [75], [76]; and, most recently, LLM-driven defenses [77], [78].

Binary-only Baseline. We read the full text of the works most likely to use sysfilter beyond related work. Table 2 lists the works we verified that run it, extend it, or reuse its components. *SysPart* (CCS 2023) uses sysfilter as the state-of-the-art binary-only baseline for temporal filtering of server binaries, and its call-graph construction reuses the Egalito VCG pass that sysfilter introduced and showed to be sound [14]. *B-Side* (Middleware 2024) compares against sysfilter and Chestnut on six servers and hundreds of Debian binaries. On average, sysfilter allowed 96 syscalls per dynamically linked binary, versus 274 for Chestnut, with an F1 score of 0.53 versus 0.31. B-Side also adopts sysfilter's address-taken analysis and evaluates against sysfilter's CVE-to-syscall list [11]. *Chestnut* (CCSW 2021) runs both sysfilter call-graph modes side by side with its own Binalyzer [21]. Lei and Shue (DIMVA 2025) compare their PHP-specific filters against sysfilter's 159-syscall policy for the PHP engine [22]. Our own later systems also build on sysfilter. *SysXCHG* (CCS 2023) uses it as its policy-extraction engine and reuses its skip-list filters [28]. *Timeloops* and *Santa* (DIMVA 2026) use it as the static baseline for learned microservice policies [26], [27]. *sysfilter+* extends it with type-based call-graph pruning, which removes 35% more indirect-call targets, and with temporal enforcement [29].

Shaping the Research Agenda. Independent evaluations have also mapped sysfilter's limitations, and those limitations shaped later work. sysfilter targets position-independent dynamically linked binaries, the default on modern Linux distributions. Run outside that assumption, on non-PIC static binaries, it either refuses to analyze them [11] or its pruned call graph under-approximates [21]. Its intra-procedural value tracking handles libc's syscall idiom and the syscall() wrapper but misses syscall numbers passed through other wrappers, which led B-Side to use symbolic execution [11]. Its address-taken over-approximation motivated type-based pruning [29].

TABLE 2. VERIFIED USES OF SYSFILTER. (†CO-AUTHORED BY A SYSFILTER AUTHOR OR ADVISED BY ONE.)

Work	Venue	Use of **sysfilter**
Chestnut [21]	CCSW 2021	Runs both modes vs. Binalyzer
SysPart [14]	CCS 2023	Baseline; reuses VCG pass
B-Side [11]	Middleware 2024	Baseline; AT analysis; CVE list
Lei & Shue [22]	DIMVA 2025	Policy-size comparison
TU Delft theses [23]–[25]	B.Sc. 2024	Run sysfilter vs. Chestnut, Confine, strace
Jia et al. [10]	arXiv 2023	Skip-list filter optimization
SysXCHG† [28]	CCS 2023	Extraction engine; skip-list filters
Timeloops† [26]	arXiv 2022	Static baseline
Santa† [27]	DIMVA 2026	Static baseline (vs. SysPart)
sysfilter+† [29]	Brown 2022	Type-based extension

A single whole-program policy motivated temporal and phase-based filtering (SysPart [14], Santa [27]). seccomp's inheritance across execve, which sysfilter's union and hierarchical modes work around, motivated kernel-level filter exchange in SysXCHG [28].

3.3. Education and Training

In 2024, at least three B.Sc. research projects at TU Delft ran sysfilter alongside Chestnut, Confine, and strace-based tracing on BusyBox configurations and real applications. They compared the tightness of the generated policies, analysis time, scalability, and setup effort [23]–[25]. One found that sysfilter "produced the smallest set in every experiment" on stripped binaries and "showed the best scaling" as features were added [23]. At Brown, sysfilter underpins a Ph.D. dissertation [79] and two Master's projects [29], [80]. The second of these, *sysfilter+*, ran a type-aware variant of sysfilter over 30K Debian sid binaries [29]. An undergraduate project ported sysfilter's extraction to FreeBSD and compared the syscall sets of 127 programs common to FreeBSD and Debian [81]. In the classroom, sysfilter has been part of the discussion of least-privilege enforcement in Brown's CSCI 1650 (Software Security and Exploitation) and CSCI 2951U (Topics in Software Security). It was also covered in the security module of Brown Systems Week 2025/2026, a five-day research seminar for 25–30 advanced (under)graduate and early Ph.D. students [82], [83].

4. Sustainability, Lessons, and Outlook

sysfilter has been developed in the open since February 2018 and is released under the BSD 3-Clause license. Its repository has over 1,000 commits, a 0.1.0 tag (Mar 2023), and the 1.0 and 1.1 release branches produced during the transition effort.

During its development we also upstreamed improvements to Egalito, including support for hand-written assembly, symbol-resolution fixes, and lower memory use, which benefit every Egalito user. Because `sysfilter`'s analyses are passes over Egalito's intermediate representation, and Egalito already supports AArch64 [2], porting `sysfilter` to AArch64 is straightforward. The architecture-specific parts are few: recognizing the syscall instruction (`svc` instead of `syscall`), tracking the syscall-number register (`x8` instead of `%rax`), and the architecture check in the filter preamble. Embedded deployments such as Star Lab's, which already target ARM64 [3], would benefit directly.

Four design choices drove `sysfilter`'s reuse. First, it operates on binaries alone, needing no source code, symbols, or recompilation. That is what made it fit an embedded vendor's root-filesystem build, and what makes it the natural baseline for later binary-level work. Second, it is fast: across Debian `sid`, the median extraction took about 30 s per binary, and 90% of binaries finished within 200 s [1]. Generating a policy therefore takes seconds, or minutes at most, without rebuilding the application and every library it links, as compiler- and source-based approaches require. That makes it practical to regenerate filters on every image build. Third, it puts soundness first: it aborts rather than emit a possibly incomplete policy, and it never relies on tracing. For adopters, filters that never break production code matter more than the last few syscalls of tightness. Fourth, it enforces with the stock kernel, using `seccomp-BPF` and a load-time constructor, so deployment needs no kernel changes. One lesson concerns outreach. A SOUPS 2025 study found that developers who write `seccomp` filters are often unaware that automated generators exist [84]. Packaging tools like sysfilter into distribution and container tooling, as Star Lab did for embedded builds, is one promising way to broaden their reach.

Availability

`sysfilter` is open source and available at https://gitlab.com/brown-ssl/sysfilter. Each component directory has a `README` with build and usage instructions, and the design is described in the open-access RAID 2020 paper [1].

References

[1] N. DeMarinis, K. Williams-King, D. Jin, R. Fonseca, and V. P. Kemerlis, "sysfilter: Automated System Call Filtering for Commodity Software," in *RAID*, 2020.

[2] D. Williams-King, H. Kobayashi, K. Williams-King, G. Patterson, F. Spano, Y. J. Wu, J. Yang, and V. P. Kemerlis, "Egalito: Layout-Agnostic Binary Recompilation," in *ASPLOS*, 2020.

[3] Star Lab Software, "System Call Filtering in Linux: An Unsung Hero," Whitepaper, 2023.

[4] ——, "How to Protect Embedded Linux with Star Lab's Kevlar Embedded Security," Blog post. [Online]. Available: https://www.starlab.io/blog/how-to-protect-your-embedded-devices-with-star-labs-kevlar-embedded-security

[5] Mercury Systems, Inc., "Mercury Acquires Star Lab to Advance Its Leadership Position in Secure Processing," Press release, May 1, 2025.

[6] GrammaTech, Inc., "Office of Naval Research Awards GrammaTech $9M for Cyber-Hardening Security Research," Press release (TPCP Innovative Naval Prototype), Oct. 2017.

[7] I. Agadakos, D. Jin, D. Williams-King, V. P. Kemerlis, and G. Portokalidis, "Nibbler: Debloating Binary Shared Libraries," in *ACSAC*, 2019.

[8] U.S. Department of Defense, "Matured and Enhanced Total Platform Cyber Protection (TPCP) Technologies for Improved Security (METIS): contract announcements," *Central New York Business Journal*.

[9] F. Huici and S. Santhanam, "Automated and Dynamic System Call Sealing," U.S. Patent 11,733,989 B2, NEC Corporation.

[10] J. Jia, Y. Zhu, D. Williams, A. Arcangeli, C. Canella, H. Franke, T. Feldman-Fitzthum, D. Skarlatos, D. Gruss, and T. Xu, "Programmable System Call Security with eBPF," arXiv:2302.10366, 2023.

[11] G. Thévenon, K. Nguetchouang, K. Lazri, A. Tchana, and P. Olivier, "B-Side: Binary-Level Static System Call Identification," in *ACM/IFIP Middleware*, 2024.

[12] T. Palit, S. Ghavamnia, and M. Polychronakis, "PhaseSeed: Precise Call Graph Construction for Split-Phase Applications using Dynamic Seeding," arXiv:2511.06661, 2025.

[13] S. Ghavamnia and J. Vanegue, "AMPLE: Fine-grained File Access Policies for Server Applications," in *IEEE/ACM ASE*, 2025.

[14] V. L. Rajagopalan, K. Kleftogiorgos, E. Göktaş, J. Xu, and G. Portokalidis, "SysPart: Automated Temporal System Call Filtering for Binaries," in *ACM CCS*, 2023.

[15] S. Ghavamnia, T. Palit, and M. Polychronakis, "C2C: Fine-grained Configuration-driven System Call Filtering," in *ACM CCS*, 2022.

[16] S. Ghavamnia, T. Palit, S. Mishra, and M. Polychronakis, "Temporal System Call Specialization for Attack Surface Reduction," in *USENIX Security*, 2020.

[17] D. Schrammel, S. Weiser, R. Sadek, and S. Mangard, "Jenny: Securing Syscalls for PKU-based Memory Isolation Systems," in *USENIX Security*, 2022.

[18] H. Lefeuvre, N. Dautenhahn, D. Chisnall, and P. Olivier, "SoK: Software Compartmentalization," in *IEEE S&P*, 2025.

[19] W. Blair, F. Araujo, T. Taylor, and J. Jang, "Automated Synthesis of Effect Graph Policies for Microservice-Aware Stateful System Call Specialization," in *IEEE S&P*, 2024.

[20] H. Kermabon-Bobinnec, Y. Jarraya, L. Wang, S. Majumdar, and M. Pourzandi, "Phoenix: Surviving Unpatched Vulnerabilities via Accurate and Efficient Filtering of Syscall Sequences," in *NDSS*, 2024.

[21] C. Canella, M. Werner, D. Gruss, and M. Schwarz, "Automating Seccomp Filter Generation for Linux Applications," in *ACM CCSW*, 2021.

[22] Y. Lei and C. A. Shue, "Making (Only) the Right Calls: Preventing Remote Code Execution Attacks in PHP Applications with Contextual, State-Sensitive System Call Filtering," in *DIMVA*, 2025.

[23] B. van der Mark, "An Analysis of System Call Set Extraction Tools on Configurable Linux Binaries," B.Sc. thesis, TU Delft, 2024.

[24] B. Selyem, "Exploring Automated System Call Sandbox Policy Generation," B.Sc. thesis, TU Delft, 2024.

[25] J. J. Patałuch, "System Call Sandboxing," B.Sc. thesis, TU Delft, 2024.

[26] M. Pancholi, A. D. Kellas, V. P. Kemerlis, and S. Sethumadhavan, "Timeloops: Automatic System Call Policy Learning for Containerized Microservices," arXiv:2204.06131, 2022.

[27] M. Pancholi, A. Kellas, K. Kaffes, S. M. Bellovin, S. Sethumadhavan, and V. P. Kemerlis, "Santa: Language Agnostic Automated System Call Policy Learning for Cloud Microservices," in *DIMVA*, 2026.

[28] A. J. Gaidis, V. Atlidakis, and V. P. Kemerlis, "SysXCHG: Refining Privilege with Adaptive System Call Filters," in *ACM CCS*, 2023.

[29] R. Abou Chaaya, "sysfilter+: Type-based System Call Filtering with Temporal Specialization," Master's project, Brown University, 2022.

[30] S. Ghavamnia, T. Palit, A. Benameur, and M. Polychronakis, "Confine: Automated System Call Policy Generation for Container Attack Surface Reduction," in *RAID*, 2020.

[31] M. Rostamipoor, S. Ghavamnia, and M. Polychronakis, "Confine: Fine-grained System Call Filtering for Container Attack Surface Reduction," *Comput. Secur.*, 2023.

[32] S. Kim, B. J. Kim, and D. H. Lee, "Prof-gen: Practical Study on System Call Whitelist Generation for Container Attack Surface Reduction," in *IEEE CLOUD*, 2021.

[33] Y. Xing, J. Cao, X. Wang, S. Torabi, K. Sun, F. Yan, and Q. Li, "SysCap: Profiling and Crosschecking Syscall and Capability Configurations for Docker Images," in *IEEE CNS*, 2022.

[34] Y. Xing, X. Wang, S. Torabi, Z. Zhang, L. Lei, and K. Sun, "A Hybrid System Call Profiling Approach for Container Protection," *IEEE Trans. Dependable Secure Comput.*, 2023.

[35] S. Yang, B. B. Kang, and J. Nam, "Optimus: Association-based Dynamic System Call Filtering for Container Attack Surface Reduction," *J. Cloud Comput.*, 2024.

[36] S. Li, J. Zhou, X. Li, W. Meng, B. Gong, and J. Hu, "Hela: A System Call Restriction Framework for Protecting the Entire Containers Lifecycle," *IEEE Trans. Cloud Comput.*, 2026.

[37] S. Song, S. Suneja, M. V. Le, and B. Tak, "On the Value of Sequence-Based System Call Filtering for Container Security," in *IEEE CLOUD*, 2023.

[38] M. V. Le, S. Ahmed, D. Williams, and H. Jamjoom, "Securing Container-based Clouds with Syscall-aware Scheduling," in *ACM AsiaCCS*, 2023.

[39] H. Kang, E. Marin, M. You, D. Perino, S. Shin, and J. Kim, "BeaCon: Automatic Container Policy Generation using Environment-aware Dynamic Analysis," *Comput. Secur.*, 2025.

[40] M. Rostamipoor, A. Sadeghi, and M. Polychronakis, "KubeKeeper: Protecting Kubernetes Secrets Against Excessive Permissions," in *IEEE EuroS&P*, 2025.

[41] N. Yang, W. Shen, J. Li, Y. Yang, K. Lu, J. Xiao, T. Zhou, C. Qin, W. Yu, J. Ma, and K. Ren, "Demons in the Shared Kernel: Abstract Resource Attacks Against OS-level Virtualization," in *ACM CCS*, 2021.

[42] K. Manakkal, N. Daughety, M. Pendleton, and H. Lu, "LITESHIELD: Secure Containers via Lightweight, Composable Userspace μKernel Services," in *USENIX ATC*, 2025.

[43] A. Moghaddas Borhan, H. Kermabon-Bobinnec, L. Wang, Y. Jarraya, and S. Majumdar, "CapMan: Detecting and Mitigating Linux Capability Abuses at Runtime to Secure Privileged Containers," in *RAID*, 2025.

[44] D. Zhan, Z. Yu, X. Yu, H. Zhang, and L. Ye, "Shrinking the Kernel Attack Surface Through Static and Dynamic Syscall Limitation," *IEEE Trans. Serv. Comput.*, 2022.

[45] C. Groh, S. Proskurin, and A. Zarras, "Free Willy: Prune System Calls to Enhance Software Security," in *ACM SAC*, 2023.

[46] Q. Zhang, C. Zhou, Y. Xu, Z. Yin, M. Wang, Z. Su, C. Sun, Y. Jiang, and J. Sun, "Building Dynamic System Call Sandbox with Partial Order Analysis," *Proc. ACM Program. Lang. (OOPSLA)*, 2023.

[47] C.-K. Hsu, Y.-C. Tseng, S.-W. Li, Y.-H. Lin, W.-C. Tian, L. C. Lam, and Y.-C. Liu, "Strengthening Application Security through Integrity Protection of System Call Usage," in *ACM SAC*, 2025.

[48] M. Rossi, M. Abbadini, M. Beretta, D. Facchinetti, and S. Paraboschi, "SysComb: Fine-Grained Transparent System Call Filtering for Attack Surface Reduction," arXiv:2608.26871, 2026.

[49] V. L. Rajagopalan and G. Portokalidis, "Evaluating the Effect of Improved Indirect Call Resolution on System Call Debloating," in *ACM FEAST*, 2024.

[50] S. Priyadarshan and S. Ghavamnia, "Applying System Call Filtering to Real-World Binaries (Experience Paper)," *Proc. ACM Softw. Eng. (ISSTA)*, 2026.

[51] M. Gelderie, V. Barth, M. Luff, and J. Birami, "Seccomp Filters from Fuzzing," in *SECRYPT*, 2022.

[52] C. Canella, S. Dorn, D. Gruss, and M. Schwarz, "SFIP: Coarse-Grained Syscall-Flow-Integrity Protection in Modern Systems," arXiv:2202.13716, 2022.

[53] C. Jelesnianski, M. Ismail, Y. Jang, D. Williams, and C. Min, "Protect the System Call, Protect (Most of) the World with BASTION," in *ACM ASPLOS*, 2023.

[54] M. M. Hasan, S. Ghavamnia, and M. Polychronakis, "Decap: Deprivileging Programs by Reducing Their Capabilities," in *RAID*, 2022.

[55] H. Patel, A. Snit, and M. Polychronakis, "Supply Chain Reaction: Enhancing the Precision of Vulnerability Triage using Code Reachability Information," in *ACSAC*, 2025.

[56] S. Joo, M. Park, H. Jang, and Y. Shin, "MimicCall: Bypassing System Call Filters via Kernel Function Redundancy," in *ACSAC*, 2025.

[57] H.-W. Hung, Y. Liu, and A. A. Sani, "Sifter: Protecting Security-Critical Kernel Modules in Android through Attack Surface Reduction," in *ACM MobiCom*, 2022.

[58] M. Abbadini, D. Facchinetti, G. Oldani, M. Rossi, and S. Paraboschi, "NatiSand: Native Code Sandboxing for JavaScript Runtimes," in *RAID*, 2023.

[59] ——, "Cage4Deno: A Fine-Grained Sandbox for Deno Subprocesses," in *ACM AsiaCCS*, 2023.

[60] W. Wang, X. Lin, J. Wang, W. Gao, D. Gu, W. Lv, and J. Wang, "Hodor: Shrinking Attack Surface on Node.js via System Call Limitation," in *ACM CCS*, 2023.

[61] C. Cesarano, M. Monperrus, and R. Natella, "GoLeash: Mitigating Golang Software Supply Chain Attacks with Runtime Policy Enforcement," *IEEE Trans. Softw. Eng.*, 2026.

[62] Y. He, Z. Zou, K. Sun, Z. Liu, K. Xu, Q. Wang, C. Shen, Z. Wang, and Q. Li, "RapidPatch: Firmware Hotpatching for Real-Time Embedded Devices," in *USENIX Security*, 2022.

[63] S. Lee and J. Kim, "PathGuard: Path-Aware Syscall Filtering for IoT Embedded Systems," in *HASP*, 2025.

[64] D. Zhan, Z. Yu, X. Yu, H. Zhang, L. Ye, and L. Liu, "Securing Operating Systems Through Fine-Grained Kernel Access Limitation for IoT Systems," *IEEE Internet Things J.*, 2022.

[65] H. Lu, S. Wang, Y. Wu, W. He, and F. Zhang, "MOAT: Towards Safe BPF Kernel Extension," in *USENIX Security*, 2024.

[66] W. He, H. Lu, F. Zhang, and S. Wang, "RingGuard: Guard io_uring with eBPF," in *ACM SIGCOMM Workshop on eBPF*, 2023.

[67] F. Yang, B. Im, W. Huang, K. Kaoudis, A. Vahldiek-Oberwagner, C.-C. Tsai, and N. Dautenhahn, "Endokernel: A Thread Safe Monitor for Lightweight Subprocess Isolation," in *USENIX Security*, 2024.

[68] M. Bauer and C. Rossow, "Cali: Compiler-Assisted Library Isolation," in *ACM AsiaCCS*, 2021.

[69] F. Yang, A. Vahldiek-Oberwagner, C.-C. Tsai, K. Kaoudis, and N. Dautenhahn, "Making 'syscall' a Privilege not a Right," arXiv:2406.07429, 2024.

[70] A. Jacobs, M. Gülmez, A. Andries, S. Volckaert, and A. Voulimeneas, "System Call Interposition Without Compromise," in *IEEE/IFIP DSN*, 2024.

[71] J. M. Gómez Moreno, V. Moutafis, A. Dionysiou, F. Kuipers, G. Smaragdakis, B. Coppens, and A. Voulimeneas, "Clair Obscur: The Light and Shadow of System Call Interposition – From Pitfalls to Solutions with K23," in *ACM/IFIP Middleware*, 2025.

[72] D. Kuvaiskii, D. Stavrakakis, K. Qin, C. Xing, P. Bhatotia, and M. Vij, "Gramine-TDX: A Lightweight OS Kernel for Confidential VMs," in *ACM CCS*, 2024.

[73] B. Mei, S. Xia, W. Wang, and D. Lin, "Cabin: Confining Untrusted Programs Within Confidential VMs," in *ICICS*, 2024.

[74] H. Lefeuvre, G. Gain, V.-A. Bădoiu, D. Dinca, V.-R. Schiller, C. Raiciu, F. Huici, and P. Olivier, "Loupe: Driving the Development of OS Compatibility Layers," in *ACM ASPLOS*, 2024.

[75] W. Zhang, D. Bali, J. Kerney, A. Panda, and S. Shenker, "Extracting Database Access-Control Policies from Web Applications," in *USENIX OSDI*, 2026.

[76] C. S. Oh, S. Lee, C. Qian, H. Koo, and W. Lee, "DeView: Confining Progressive Web Applications by Debloating Web APIs," in *ACSAC*, 2022.

[77] Y. Peng, Z. Zhu, J. Hu, J. Wang, H. Wan, and X. Zhao, "Kintsugi: Empowering LLMs to Mitigate Web Vulnerabilities via Runtime Policy Injection," in *USENIX Security*, 2026.

[78] D. Zhan, X. Yan, Z. Yu, Y. Guo, K. Tan, and L. Zhang, "AgentSys-Filter: A Multi-agent Analysis Approach for Scenario-Aware Syscall Limitation," in *IEEE/ACM ASE*, 2026.

[79] N. DeMarinis, "Improving Application Security at Scale by Reducing System Call and Library Overprivilege," Ph.D. dissertation, Brown University, 2022.

[80] K. Williams-King, "Hardening Commodity Software through Automated System Call Filtering and Binary Recompilation," Master's project, Brown University, 2020.

[81] S. Rowley, "Assessing the Effectiveness of System Call Filtering between Linux and FreeBSD," undergraduate research report, Brown University, 2022.

[82] Brown Systems Week, "Brown Systems Week 2025: Mini-Course on Supply-Chain Security," Brown University, Jun. 2025. [Online]. Available: https://brown-systems-week.github.io/2025/

[83] ——, "Brown Systems Week 2026: Mini-Course on Supply-Chain Security," Brown University, Jun. 2026. [Online]. Available: https://brown-systems-week.github.io/2026/

[84] M. Alhindi and J. Hallett, "Playing in the Sandbox: A Study on the Usability of Seccomp," in *SOUPS*, 2025.