Poster: Hybrid Seccomp Profiling with LLM Assistance for Container Security

Jiwoo Ahn†, Youyang Kim†, Young-Kyoon Suh†, Ashish Kundu‡, Byungchul Tak†*
† Kyungpook National University, Daegu, Republic of Korea
‡ Cisco Research, San Jose, CA, USA

***Abstract*—The effectiveness of the Seccomp kernel feature depends on how tightly and accurately the necessary system calls are specified in the seccomp policy. Static code analysis may miss out or over-approximate required system calls. With dynamic analysis, it is difficult to cover all possible execution paths. In this work, we aim to advance the hybrid approach by enabling it to increase the coverage of the target application's functionalities via LLM test case generation.**

I. Introduction

Modern container platforms run many workloads on a shared Linux kernel, which makes the system-call interface a vulnerable attack surface. Seccomp-BPF provides a practical defense-in-depth layer by restricting each containerized process to only the system calls it needs. However, its effectiveness critically depends on accurate profiling. Overly permissive policies leave unnecessary kernel entry points open, while too restrictive policies can render containerized applications inexecutable. Accurate seccomp profiles are especially crucial in containerized service environments, given rapid deployments, frequent updates, third-party dependencies, and scaling.

Existing approaches to system call profiling are generally categorized into three types: static [1]–[3], dynamic [4], [5], and hybrid approaches [6]–[8]. *Static approaches* infer required system calls from program source code, binaries, or dependency graphs without executing the workload. They are known to provide complete coverage, but may suffer from over-approximation due to conservative analysis and limited visibility into runtime configuration and indirect behaviors. In *Dynamic approaches*, target applications are directly executed with representative workloads, and the observed system calls are recorded. They tend to produce tighter seccomp policies but can severely under-approximate legitimate code paths. *Hybrid approaches* combine both static and dynamic approaches in a complementary manner.

Although the *hybrid* approach is promising with combined advantages of both the static and dynamic approaches, further advancement is hindered by three major challenges: *i)* insufficient coverage of state-of-the-art dynamic techniques, *ii)* the lack of ground-truth system call sets for target applications to assess the quality of any hybrid techniques, and *iii)* over-approximation of static analysis. In this work, we aim to advance state-of-the-art hybrid approaches by addressing these challenges.

*Corresponding author.

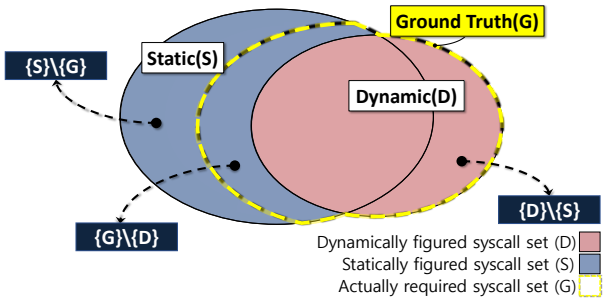


Fig. 1: Conceptual overview of syscall set boundaries. The diagram depicts the relationship between the statically predicted set (S), the dynamically observed set (D), and the ground truth (G).

II. System Call Classification in Hybrid Approach

Figure 1 depicts the logical view of the system call classification when static and dynamic approaches are applied. Our goal is to determine the ground truth system call set, G. It is believed that this holds:

$$|S \cup D| \geq |G| \geq |D| \tag{1}$$

However, we are unaware of the exact set G. Note that S alone is insufficient. We have found system calls that the static analysis technique missed (i.e., $\{D\} \setminus \{S\}$).

When building seccomp profiles, we believe it is more important to have a broad enough seccomp profile that guarantees uninterrupted execution, even at the cost of a slightly larger attack surface, than to have a tight seccomp profile that might crash applications. Thus, we combine the static and dynamic profiles to build $|S \cup D|$ first. Then, while monitoring the system calls during runtime, we treat $\{S\} \setminus \{D\}$ differently from legitimate system calls in set D. Since we are unaware of exactly which ones belong to G or not, all of $\{S\} \setminus \{D\}$ can potentially be $\{S\} \setminus \{G\}$. Thus, we count the invocations of these system calls and raise alerts when the count exceeds a threshold. This hybrid approach avoids the risk of service termination caused by overly strict filters while still detecting potential attacks. The feasibility of this hybrid approach lies in minimizing $\{G\} \setminus \{D\}$ to eliminate false alarms and maximizing $\{D\} \setminus \{S\}$ to identify system calls missed by static analysis. Achieving this requires test case generation with significantly higher coverage than state-of-the-art approaches, which is the challenge we address by leveraging LLMs to generate comprehensive and valid test suites.

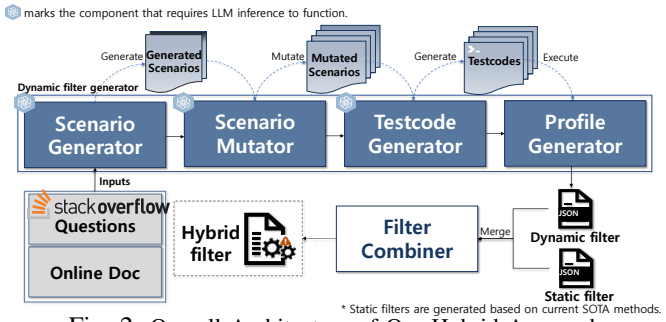


Fig. 2: Overall Architecture of Our Hybrid Approach

III. Methodology

The overall architecture of our system is shown in Fig. 2. It consists of two steps. First, we leverage LLMs to generate a high-coverage test suite. A dynamic seccomp filter is then built from the observed system calls during the execution of the generated test suite. Second, we produce a hybrid seccomp profile by integrating the dynamic filter with statically extracted syscalls using a known static analysis profiling tool.

For input data, we use the official documentation of the target application as the primary source for our dynamic filter generator. To further capture real-world use cases beyond formal documentation, we collect application-specific questions from *Stack Overflow* [9], which often reflect common usage scenarios, edge cases, and even misconfigurations encountered by users. We adopt *Retrieval-Augmented Generation* (RAG) and *few-shot prompting* to improve the correctness and coverage of generated scenarios and test suites.

The generated test cases (Table I) are executed in a controlled environment to trace system calls and construct a dynamic filter, which is then combined with a static analysis–based filter to form a hybrid filter.

IV. Ground Truth Identification

To enable a robust evaluation of our generated seccomp profiles, we aim to establish a reliable ground truth that captures the relationship between libc functions and the system calls they may invoke. Static call graph construction becomes fundamentally challenging in the presence of indirect calls, such as those implemented via function pointers or callbacks, where call targets are resolved only at runtime. Prior work has shown that Andersen-style points-to analyses and their variants inevitably over-approximate possible targets; thus, accurately resolving indirect calls in large codebases remains an open problem. To address this limitation, we adopt a semi-automated approach.

We analyzed 4,052 glibc source files for our build target and identified 608 unique indirect call sites. To establish a reliable ground truth, we manually resolve these sites and reconnect missing call-graph edges, bridging the gap between statically identified syscalls and actual requirements.

V. Preliminary Evaluation

Table II compares the set of system calls in the seccomp profiles produced by the dynamic component of ours and two other state-of-the-art techniques — Chestnut [6] and Confine [1]. Even though the static analysis tools conservatively constructed seccomp profiles, they still missed certain system calls that were triggered only by our generated workloads, which resulted in service failures or termination.

TABLE I: Number of generated test codes by LLM per application

Apps	Testcodes		Total	LoC	Size(MB)
	Document	Stackoverflow			
Redis	2,383	12,162	14,545	364K	16
Nginx	2,167	18,183	20,350	436K	23
Memcached	3,691	1,488	5,179	169K	8
Apache Httpd	1,910	28,814	30,724	743K	31
Postgresql	2,903	111,815	114,718	3,058K	151

TABLE II: Seccomp profile generation result comparison. The numbers in parentheses indicate the number of missing system calls for each work.

Container Images	# of System calls		
	Ours	Chestnut (Sourcealyzer)	Confine
Redis	70	87 (14)	148 (2)
Nginx	76	123 (7)	170 (1)
Memcached	48	98 (6)	150 (2)
Apache Httpd	65	94 (10)	151 (2)
Postgresql	93	113 (15)	249 (2)

VI. Future Plan

In our future work, we plan to implement a call counter and alarm system specifically for the set of system calls identified solely through static analysis. This mechanism will monitor and notify of any execution within this subset during runtime. Furthermore, we aim to establish a ground truth set through manual verification to rigorously evaluate the performance and accuracy of our proposed profiling approach.

References

[1] S. Ghavamnia, T. Palit, A. Benameur, and M. Polychronakis, "Confine: Automated system call policy generation for container attack surface reduction," in *23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020)*, 2020, pp. 443–458.

[2] N. DeMarinis, K. Williams-King, D. Jin, R. Fonseca, and V. P. Kemerlis, "Sysfilter: Automated system call filtering for commodity software," in *23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020)*, 2020, pp. 459–474.

[3] S. Ghavamnia, T. Palit, S. Mishra, and M. Polychronakis, "Temporal system call specialization for attack surface reduction," in *29th USENIX Security Symposium (USENIX Security 20)*, 2020, pp. 1749–1766.

[4] L. Lei, J. Sun, K. Sun, C. Shenefiel, R. Ma, Y. Wang, and Q. Li, "Speaker: Split-phase execution of application containers," in *Detection of Intrusions and Malware, and Vulnerability Assessment*, M. Polychronakis and M. Meier, Eds. Cham: Springer International Publishing, 2017, pp. 230–251.

[5] Z. Wan, D. Lo, X. Xia, L. Cai, and S. Li, "Mining sandboxes for linux containers," in *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, 2017, pp. 92–102.

[6] C. Canella, M. Werner, D. Gruss, and M. Schwarz, "Automating seccomp filter generation for linux applications," in *Proceedings of the 2021 on Cloud Computing Security Workshop*, ser. CCSW '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 139–151.

[7] D. Zhan, Z. Yu, X. Yu, H. Zhang, and L. Ye, "Shrinking the kernel attack surface through static and dynamic syscall limitation," *IEEE Transactions on Services Computing*, vol. 16, no. 2, pp. 1431–1443, 2022.

[8] V. L. Rajagopalan, K. Kleftogiorgos, E. Göktas, J. Xu, and G. Portokalidis, "Syspart: Automated temporal system call filtering for binaries," in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, 2023, p. 1979–1993. [Online]. Available: https://doi.org/10.1145/3576915.3623207

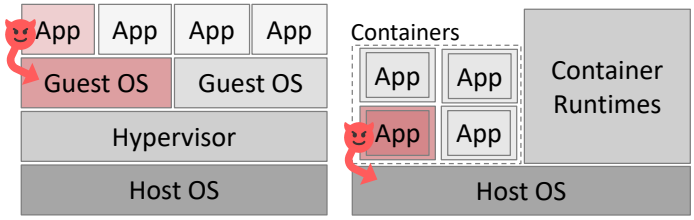
[9] Stack Overflow, "Stack overflow," https://stackoverflow.com, 2026, online; accessed 12-Jan-2026.

Hybrid Seccomp Profiling with LLM Assistance for Container Security

Jiwoo Ahn[1], Youyang Kim[1], Young-Kyoon Suh[1], Ashish Kundu[2], Byungchul Tak[1]
Kyungpook **N**ational **U**niversity(**KNU**)[1], Cisco Research[2]

Background

- **Containers can be an entry-point to the kernel for attackers**
  Containers share the host kernel, making them more vulnerable than VMs.
- **Architecture: VMs vs Containers**

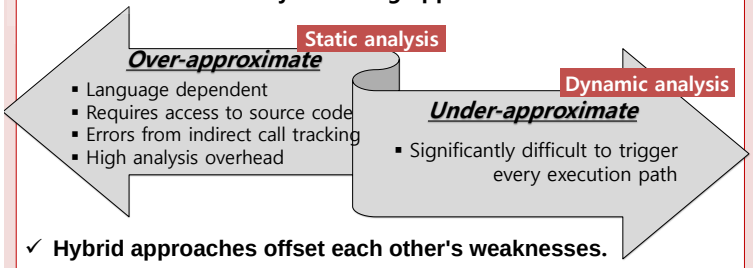


- **Attack Surface Reduction**
  - Modern Linux supports over 450 system calls, many of which are unnecessary for typical application behavior.
  - Linux exposes all system calls to user-space processes.
  - Each call may present a potential attack surface for adversaries.

- **SECCOMP restricts available system calls to a certain process.**
  - The effectiveness of a Seccomp policy depends on how precisely it captures the system call set required by target application.
  - Accurately identifying the system call set is time-consuming, requires domain expertise and labor-intensive.

Motivation and Research Goal

- **Limitations of two major existing approaches**



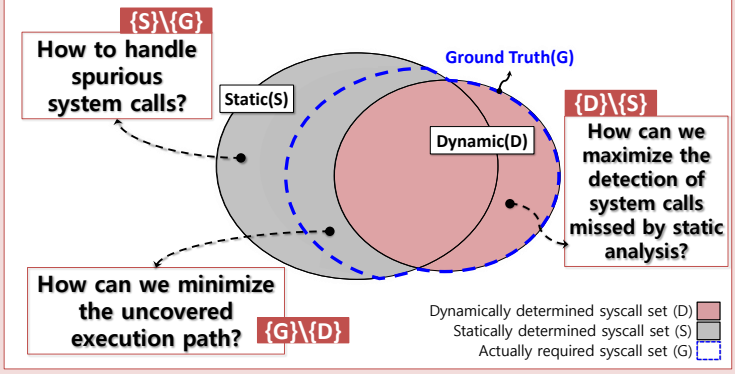
- ✓ **Hybrid approaches offset each other's weaknesses.**
  However, in existing approaches, the dynamic phase relies solely on human-crafted test suites and is treated merely as an auxiliary step for verification.

- **We want to:**

"Drastically enhance the dynamic component of hybrid approaches through automated generation of test codes via LLMs"

Goal: By strengthening dynamic analysis,
- Minimize under-approximation.
- Identify candidates that may be over-approximated.

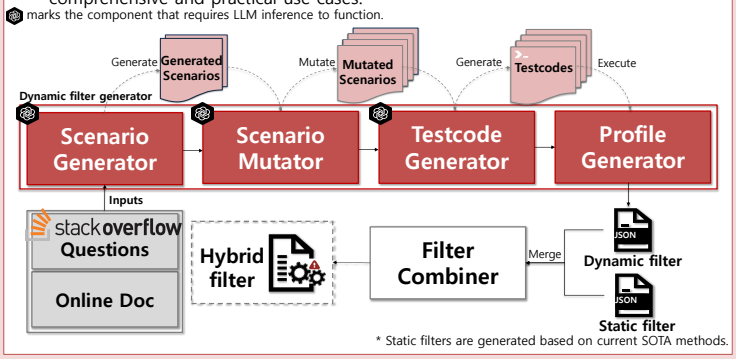
Research Questions



[Acknowledgement] This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government(MSIT) (RS-2021-NR060080) and by Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (RS-2025-25453740, Development of a large-scale mixed device control and management platform for edge AI server systems).

Overall Architecture

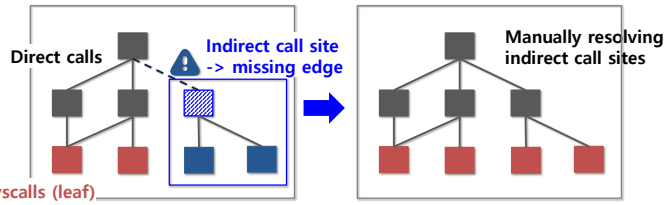
- ✓ **Our objective is to achieve coverage over all feasible execution paths:**
  Utilize *Online documentation* and stackoverflowquestions to cover comprehensive and practical use cases.



Ground Truth(GT) Identification

- **Construct a ground-truth baseline for evaluation**
  - Most C/C+ applications invoke system call through **libc library**.
  - Build a callgraph mapping from **libc library** to **system calls.**

- **Indirect calls lead to incomplete call graphs.**
  - Indirect call when target address is determined at runtime.
  - e.g.) function pointers, jump tables, dynamic dispatch
  - Indirect call resolution remains an unresolved challenge.



- **Our approach with manual inspection for libc function to syscall map:**
  - Step 1. Generate AST using clang frontend for each C file in libc codebase.
  - Step 2. Extract direct call relations and indirect call sites.
  - Step 3. Augment call graph with manually configured indirect edges.

- **Total indirect call sites vary by library:**
  - **Musl libc v1.1.18: 76** sites
  - **Musl libc v1.2.5: 103** sites
  - **Glibc v2.36: 608** sites
  - Scope: We will resolve only the indirect calls involved in syscall invocation by tracing the call graph back from syscall sites.

Preliminary Evaluation

- **[Table1] Number of allowed system calls: prior studies vs. ours**
  - Numbers in parens: # of missed syscalls per work, found by ours.
  - Confine/our tool: containeraized; Chestnut/Musl GT: standalone bin.

Container Images (Musl based)	# of system calls			
	Musl GT	Chestnut[1] - Sourcealyzer	Ours Dynamic	Confine[2]
Redis	80	87 ({D}/{S}=12)	67	206 ({D}/{S}=1)
Nginx	99	123 ({D}/{S}=4)	71	256 ({D}/{S}=0)
Memcached	81	98 ({D}/{S}=3)	44	246 ({D}/{S}=0)
Apache Httpd	56	94 ({D}/{S}=3)	56	236 ({D}/{S}=0)
Postgresql	100	113 ({D}/{S}=11)	92	252 ({D}/{S}=1)

Container Images (Glibc based)	# of system calls		
	Chestnut – Sourcealyzer	Ours Dynamic	Confine
Redis	N/A	70	148 ({D}/{S}=2)
Nginx	N/A	76	170 ({D}/{S}=1)
Memcached	N/A	48	150 ({D}/{S}=2)
Apache Httpd	N/A	65	151 ({D}/{S}=2)
Postgresql	N/A	93	249 ({D}/{S}=2)

[1] Canella, Claudio, et al. "Automating seccomp filter generation for linux applications." CCSW 2021.
[2] Ghavamnia, Seyedhamed, et al. "Confine: Automated system call policy generation for container attack surface reduction." RAID 2020.