System Call Sandboxing

Analysis of *PWD* and *NGINX* system call policy generation using dynamic and static techniques

Jakub Jarosław Patałuch

Supervisor: Alexios Voulimeneas

EEMCS, Delft University of Technology, The Netherlands

A Thesis Submitted to EEMCS Faculty Delft University of Technology,
In Partial Fulfilment of the Requirements
For the Bachelor of Computer Science and Engineering
June 23, 2024

Name of the student: Jakub Jarosław Patałuch

Final project course: CSE3000 Research Project
Thesis committee: Alexios Voulimeneas, Przemysław Pawełczak

Abstract

System call sandboxing represents a pivotal security measure in the contemporary digital landscape, where reducing the attack surface of applications is crucial to mitigate potential cyber threats. This paper investigates the efficacy of static versus dynamic system call filtering techniques across different execution phases of selected applications, namely *PWD* and *NGINX*. Employing automated tools such as *sysfilter*, and *chestnut*, we collected comprehensive data through *strace* to delineate essential system calls required for each application phase. Our analysis compares these results with the policies generated by the automated tools, providing insights into the strengths and limitations of static and dynamic sandboxing methodologies. This study ultimately seeks to refine system call policies and balance robust security with necessary application functionality.

1 Introduction

Context and Importance of System Call Sandboxing

System calls are the essential interface between an application and the OS kernel, allowing for file handling, process control, and network communications [4]. Through these calls, applications request the kernel to perform restricted tasks. This is crucial for the efficiency and functionality of the software. However, the capabilities of system calls also make them prone to exploitation. Malicious actors can utilize syscalls to execute arbitrary code, escalate privileges, or perform unauthorized actions, often resulting in significant security threats [1].

System call sandboxing is a security technique to mitigate these risks by limiting applications to a minimal set of system calls [6]. This restricted environment aims to prevent malicious or buggy code from executing harmful operations by reducing an application's attack surface.

Motivation for Research

Currently, the complexity and connectivity of applications are rapidly increasing. Modern applications, often web-based and distributed, depend heavily on external communications and data exchanges [7]. This complexity not only enhances the capabilities of software but also increases its vulnerability to cyberattacks. The reliance on multiple system calls makes applications particularly prone to such.

Significant incidents like the Heartbleed bug and the EternalBlue exploit have demonstrated how vulnerabilities in system operations can lead to security disasters [3; 11]. These incidents highlight the need for effective system call management and monitoring techniques.

Research Objectives and Questions

This research aims to evaluate and compare static and dynamic system call sandboxing techniques. Static sandboxing employs predefined policies that restrict application behavior without adaptation during runtime, while dynamic sandboxing adjusts permissible system calls in response to the application's context [10]. The primary objective is to determine which methodology more effectively reduces the attack surface without impacting the functionality or significantly impacting the performance of the application.

The specific research questions this study addresses are:

- What are the essential system calls required for the correct operation of selected applications (PWD and NGINX) across various execution phases?
- How do static system call filtering techniques compare to dynamic techniques in terms of accuracy, security, and performance?
- Can dynamic system call sandboxing adjust more effectively to the operational context of an application, thereby providing enhanced security without impacting system performance?

By addressing these questions, this research aims to provide an understanding of system call sandboxing's impact on application security and provide actionable insights for optimizing these techniques in real-world applications.

The remainder of this paper is organized as follows: Section 2 provides an overview of related work and existing literature. Section 3 outlines the methodological approach, while Section 4 presents experimental results. Section 5 discusses responsible research, and Section 6 offers a discussion that concludes with future directions. An appendix is included for key results and generative AI usage.

2 Background and Literature review

2.1 Overview of System Call Sandboxing Techniques

Static Sandboxing Methods:

Static system call sandboxing involves defining a fixed set of rules or policies that specify which system calls an application can execute. These policies are typically established based on the analysis of the application's code or its binary and remain unchanged during runtime. The primary advantage of static sandboxing lies in its simplicity and predictability. It provides a stable security environment that can be thoroughly tested and verified. However, one significant drawback of static methods is their overapproximation issue. Static methods cannot adapt to application behavior changes or respond to emerging threats. Since these methods rely just on static information, they often permit more system calls than the application requires during execution. This can lead to policies that are too permissive, potentially leaving exploitable gaps. Tools like *sysfilter* and *Binalyzer* from *chestnut* utilize such static techniques.

Dynamic Sandboxing Methods:

Contrasting with static methods, dynamic sandboxing adapts to the application's behavior in real-time. This approach involves monitoring the application's execution and dynamically adjusting the set of permissible system calls based on the current context and state of the application. Dynamic methods, such as those implemented by *confine* and *Dynalyzer* part of *chestnut*, offer the advantage of being more responsive to the application's needs, reducing unnecessary restrictions and better mitigating unforeseen threats. The com-

plexity of implementing dynamic sandboxing, however, introduces challenges in terms of performance overhead and the potential for false positives, where legitimate actions might be mistakenly blocked as the dynamic method may not cover all possible execution paths.

2.2 Previous Research and Gap Analysis

Key Studies and Findings

System call sandboxing has been the focus of extensive research. Canella et al.'s [2] work on *chestnut* emphasized automating seccomp filter generation by dynamically analyzing Linux applications to detect necessary system calls. Ghavamnia et al. [5] introduced *confine*, which leverages static and runtime analysis to refine system call policies based on the containerized application's behavior. Both studies illustrate the growing precision in sandboxing technologies on a year-by-year basis.

Research Gaps

Despite advancements, significant gaps remain in the adaptation of sandboxing policies to specific application phases and in managing the performance impact. Most existing research tends to generalize the application's behavior across all phases, which can lead to inefficiencies or security risks. For instance, an application may only need certain system calls during initialization, which is unnecessary and potentially dangerous once it transitions to a serving phase (*execve* syscall which is usually needed as the first instruction and carries a risk of RCE - Remote Code Execution). Additionally, the performance degradation associated with monitoring and adjusting system call permissions in real-time is often inadequately addressed. A notable contribution in this area is the study by Porter et al., presented in their paper *Temporal System Call Specialization for Attack Surface Reduction* [9], which is one of the first to propose separating system call filtering into distinct phases based on application needs. This approach significantly enhances the precision of sandboxing policies, enabling more tailored security measures that adapt to the varying requirements of each phase. By focusing on phase-specific system call needs, this methodology not only reduces the attack surface related to the use of so-called "dangerous syscalls" (See Figure 1) but also addresses some of the performance concerns highlighted in other studies.

2.3 Relevance of Automated Tools

Development and Contributions

Automated tools like *sysfilter*, *confine*, and *chestnut* represent significant steps forward in system call policy generation. *Sysfilter* uses binary analysis to determine which system calls are likely to be used by an application, effectively reducing the manual effort required in defining policies. Confine extends this by incorporating runtime behavior to dynamically adjust system call lists, thereby offering more granular control tailored to the application's operational context under containerized environments. *Chestnut* provides two options: *Binalyzer* and *Dynalyzer* which can do both.

Effectiveness and Accuracy in Real-World Scenarios

Evaluating these tools in real-world scenarios has shown varied results. *Chestnut*'s approach to dynamically generate seccomp filters based on observed behavior presents a promising method to reduce the attack surface while maintaining application functionality. However, the real-world applicability of these tools often confronts challenges such as the overhead of dynamic analysis and the complexity of accurately predicting application needs without prior extensive profiling.

To conclude, the literature on system call sandboxing highlights a critical evolution from static to dynamic methodologies, driven by the need for adaptive security measures in complex application environments. Automated tools have played a key role in advancing this field, though their deployment in real-world scenarios reveals both a lot of positives and even more limitations. Continued research is necessary to close the gap between theoretical effectiveness and practical usability, particularly in optimizing performance and enhancing the accuracy of dynamic sandboxing techniques.

3 Methodology

3.1 Experimental Setup

Docker Environment Configuration

To ensure reproducibility and control over the testing environment, each application, PWD and NGINX, was housed in its own Docker container. The docker host was run on a Ubuntu 24.04 LTS machine equipped with Intel I5-8250U, 16GB DDR3 RAM, and SSD drive with 60GB of storage and 8GB of RAM dedicated for the docker. Ubuntu 18.04 LTS was selected as the base operating system for these containers. The choice of Ubuntu 18.04 LTS, the latest Long-Term Support version available at the time when both the *chestnut* and *sysfilter* papers were published, was strategic. This version provides a stable and widely supported platform, ensuring that the OS would not introduce variables into the sandboxing experiments due to its inconsistencies. Additionally, this version of OS is recommended by the papers' authors.

Application Selection and Configuration

The selection of applications was aimed at examining a system-call sandboxing across a wide spectrum of complexity and operational behavior. PWD, a simple utility, provides a clear base case of predictable system call usage. NGINX, in contrast, is a robust, high-performance web server and reverse proxy, offering a complex case study with multiple execution phases and a rich set of system call interactions. For this study, NGINX was configured to run with a single worker to simplify the analysis and focus more on its system call behavior. It was set up to serve a static index page under "/index.html", but was accessed by the client browser not through a direct path like "/index.html," but via "/" using match rules, further adding complexity to its behavior.

3.2 Data Collection Techniques

System Call Monitoring with *strace*

The *strace* tool was indispensable for monitoring and logging the system calls invoked by PWD and NGINX. System call data were collected and analyzed across different phases of application operation. This phase-specific monitoring was crucial for identifying which system calls were essential to each part of the application lifecycle/runtime.

Threat level	Dangerous system calls
1. Full control of the system	chmod, fchmod, chown, fchown, lchown, execve, mount, rename, open, link
2. Denial of service	umount, mkdir, rmdir, umount2, ioctl, nfsservctl, truncate, ftruncate, quotactl, dup, dup2, flock, fork, kill, iopl, reboot, ioperm, clone
3. Used for subverting the invoking process	read, write, close, chdir, lseek, dup, fcntl, umask, chroot, select, fsync, fchdir, llseek, newselect, readv, writev, poll, pread, pwrite, sendfile, putpmsg, utime
4. Harmless	getpid, getppid, getuid, getgid, geteuid, getegid, acct, getpgrp, sgetmask, getrlimit, getrusage, getgroups, getpriority, sched getscheduler, sched getparam, sched get

Figure 1: Threat model of syscalls

Categorization and Analysis of System Calls

The system calls logged by strace were categorized according to their operational significance in each application phase. This analysis not only facilitated a deeper understanding of application behavior but also helped in formulating a minimal and precise set of necessary system calls for each stage, thereby enhancing both security and performance. We tried to avoid too much granularity, as theoretically everything can be scoped down to each syscall but to keep it on the operational (human-readable) level.

3.3 Application of Automated Tools

Use of Automated Tools

Despite initial considerations, the *confine* was ultimately not used due to its dependency issues, reliance on deprecated packages, and the potential variability in results arising from different execution environments compared to other tools. The analysis thus focused on employing *chestnut* and *sysfilter* to generate system call policies. These tools were applied to analyze the behavior of both *PWD* and *NGINX* within their respective *Docker* environments.

Criteria for Evaluating Generated Policies

The policies generated by *chestnut* and *sysfilter* were evaluated based on several key criteria:

- **Completeness:** Ensuring that the policies included all essential system calls identified during the manual tracing process.
- **Minimality:** Limiting the policies to only include necessary system calls, thereby preventing excessive permissiveness that could increase security risks.
- **Security Efficacy:** Assessing the effectiveness of each policy in minimizing the potential attack surface by restricting access to non-essential system calls. We embrace the following threat model for the syscalls (See Figure 1) from the study "Assessing vulnerability exploitability risk using software properties" [8]
- **Performance Impact:** As we didn't consider the actual dynamic analysis tool (due to focus on *Binalyzer* for *chestnut*), we assessed how long the generating policy took before the first execution.

4 Experimental Results

Detailed data can be accessed publicly at: https://github.com/kubapat/Research-project

4.1 Analysis of System Call Data

PWD System Call Analysis

The application PWD was observed using `strace` to log its system call usage across various operational phases. The critical system calls were found to be mostly involved with file and environment handling. The Table 1 summarizes the essential system calls for each distinguished operational phase.

Both *sysfilter* and *chestnut* static analyses overestimated the required system calls, including those not utilized during the application's standard operation. Respectively 41 for *chestnut* and 49 for *sysfilter*, whereas at most 31 calls were required. Most of the syscalls that were above the necessity were syscalls classified in the Level 4 - Harmless category (see Figure 1), but there were also a few of the ones classified on Level 2. See Appendix 1 for those generated syscall policies.

NGINX System Call Analysis

NGINX, being a complex web server, exhibited a diverse range of system call usages across its lifecycle, significantly more varied than those of PWD. The phase-specific system calls are detailed in the table (See Table 2). We managed to divide the *NGINX* lifecycle into 11 phases that provide the best balance between granularity and functionality, namely each phase represents a high-level feature while it's not scoped to the system call level so it's achievable to be created by humans and distinguished by a dynamic filter. The identification and division of these phases were achieved through manual analysis of the triggered syscalls using *strace*. By looking at the specific syscalls, their arguments, and their order of call and by following the NGINX codebase we were able to come up with the following phases:

- Execution initiation
- Loading shared libraries
- reading configuration files
- initialization of logger
- set up of worker processes (analyzed just for single worker process)
- open of necessary files and directories
- creating and configuring sockets
- setting up signal handlers
- worker process initialization
- entering event loop
- accepting and serving requests (just static content)

As for the PWD once again *chestnut* and *sysfilter* whitelisted way too many system calls. We came up with just 52 unique system calls while *chestnut* with 107 syscalls and *sysfilter* with 115 sycalls. See Appendix 2 for those generated syscall policies.

4.2 Tool Efficacy Comparison

The evaluation of *sysfilter* and *chestnut* highlighted significant differences in their effectiveness across various application scenarios:

Execution Phase	System Calls Involved
Execution Initiation	`execve, brk`
Loading Shared Libraries	`access, openat, fstat, mmap, close`
Memory and Environment Setup	`arch_prctl, mprotect, munmap, brk`
Reading Configuration	`getcwd, stat`
Output Handling	`fstat, write, close`
Error Handling	`write`
Process Termination	`close, exit_group`

Table 1: Essential system calls for PWD across different execution phases.

Execution Phase	System Calls Involved
Execution Initiation	`execve, brk, arch_prctl`
Loading Shared Libraries	`access, openat, read, fstat, mmap, close, mprotect`
Reading Configuration Files	`openat, fstat, pread64`
Initializing Logging	`openat, fstat, futex`
Setting Up Worker Processes	`clone, set_robust_list, getpid, close, setsid, umask, dup2, rt_sigprocmask, socketpair, ioctl, fcntl`
Opening Necessary Files and Directories	`openat, fstat, pread64, getdents64, close`
Creating and Configuring Sockets	`socket, setsockopt, ioctl, bind, listen`
Setting Up Signal Handlers	`rt_sigaction`
Worker Process Initialization	`setgid, setuid, prctl, rt_sigprocmask`
Entering Event Loop	`epoll_create, eventfd2, epoll_ctl, socketpair`
Accepting and Serving Requests	`epoll_wait, gettimeofday, accept4, recvfrom, stat, openat, fstat, writev, write, close, setsockopt`

Table 2: Essential system calls for NGINX across different execution phases.

Sysfilter's analysis:
Sysfilter, while effective for static applications like PWD, often included unnecessary system calls in its policies for NGINX. This not only resulted in overly permissive policies but also increased the risk of security vulnerabilities by not adapting to the unique needs of each phase.

Chestnut's analysis via Binalyzer:
Although Chestnut did not employ dynamic techniques beyond its static analysis capabilities, it offered a more refined system call policy for NGINX by identifying and limiting system calls more closely aligned with those observed in manual analysis. However, the lack of dynamic adjustment meant that unforeseen runtime behaviors could not be effectively mitigated.

It's worth noticing that *NGINX* has way more use scenarios than we covered and most probably results have such a significant gap (over 50% compared to just 30% in the case of *PWD*) due to rather specific use case applied in the manual policy crafting compared to capabilities of *NGINX* that were discovered by static analyzers while crafting the policy. But the key difference in the case of *NGINX* wasn't overestimation of allowed syscalls but non-phase specific syscall policy which doesn't consider setup stages and thus exposes a lot of dangerous (Level 1 and 2 - See Figure 1) syscalls to the application that has direct interface with the outer environment.

5 Responsible Research

Cybersecurity research and its innovative developments are benefiting greatly from utilizing open-source tools and software especially when practiced in support of the very principles of transparency and collaboration. The very fact that is the nature of open-source technology which requires community input to improve the security and stability of projects via collective expertise is its best feature. The research, where we used such open-source tools as *strace*, *sysfilter*, and *chestnut* on open-source Linux platform are considered the best ethical methods of promoting the ability to replicate the research and the trust in the results. These are not only available to other researchers but also to the public at large. The latter can make use of them in a variety of ways: duplicate studies, affirm results and push the limit of the analysis to new and original directions. The broad availability of information thus ensures that users are informed and able to secure their systems against evolving threats, which is undoubtedly of primary importance, overshadowing any monetary interests.

Contrarily, the implementation of systems with strict security measures like syscall sandboxing, on the one hand, is a vital aspect that includes extensive ethical considerations. Whereas the main purpose of security measures is to prevent the system from unauthorized access or preventing its misuse, the same restrictions can sometimes have other implications like disturbing the livelihood of legitimate users or various privacy issues. Moreover, for instance, the introduction of over-restrictive sandboxing might result in less effective applications, which, in turn, can cause low performance and frustration in users' interaction with the system. The even more serious matter would be that, if implemented inappropriately, these measures can be used as tools for surveillance

or control if not strictly in terms of carrying out just the necessary security operations safety protocols besides them.

6 Discussion, Conclusions and Future Research

6.1 Implications of Findings

The findings of this study underscore the intricate balance between system security and application performance that system-call sandboxing seeks to manage. By analyzing the behavior of applications through manual and automated system call filtering tools like *sysfilter* and *chestnut*, it is evident that static tools, while useful, may not fully capture the dynamic nature of complex applications like NGINX, where range of the syscalls is so wide that runtime-wide filter may be too broad. Conversely, the manual approach, although more labor-intensive, often results in more finely tuned security policies, but doesn't apply to everyday work and doesn't scale to bigger, more complex applications. It also may differ per OS that we run our software on.

The research highlights the significant advantage of a hybrid sandboxing approach, where static analysis is complemented by dynamic adjustments based on real-time application behavior. Such an approach could potentially offer the best of both worlds: the efficiency and predictability of static tools with the adaptability of dynamic monitoring. In real-world applications, this could translate to enhanced security without a significant compromise in performance, especially in environments where applications undergo frequent updates and changes.

Integrating a hybrid approach in operational settings, however, would require tools that can seamlessly transition between static analysis and dynamic monitoring. The potential for such tools could be revolutionary, providing a robust framework for securing applications in a manner that is both proactive and reactive to emerging threats.

6.2 Reflection on the Study's Conclusions

The conclusions of this study were drawn from a rigorous analysis of system call data collected via *strace* and the subsequent comparison of this data with the policies generated by both *sysfilter* and *chestnut*. This method allowed for a detailed understanding of the strengths and limitations of existing tools and paved the way for considering more complex and flexible approaches like hybrid sandboxing, which joins the performance of static analysis with the adaptability of dynamic scenario.

The process underscored the importance of considering the operational context of applications when designing security measures. For instance, the distinct behaviors observed in PWD versus NGINX during the study illustrate that a one-size-fits-all approach to sandboxing is **NOT** feasible. Customization and adaptability are crucial for effective security.

6.3 Possible Explanations of Results

The differential effectiveness of *sysfilter* and *chestnut* observed in the study could be attributed to the inherent design of these tools. *Sysfilter*'s and *chestnut* tendencies to include unnecessary system calls in its policies might be due to its reliance solely on static analysis, which does not account for the dynamic operational states of applications like NGINX. Chestnut's better performance, in some cases, suggests that even limited use of dynamic insights (as in static analysis informed by some runtime data) can enhance policy accuracy.

The superior results of the manual analysis demonstrate that current automated tools still lack some of the nuanced decision-making capabilities of human analysts, particularly in complex scenarios. This suggests that possibly further development in AI and machine learning, which could draw conclusions like humans from the manual work and decide upon the possible importance of selected syscall could be a way to go in advancing the field of automated system call policy generation.

6.4 Limitations and Challenges

This study is not without its limitations. The scope was confined to a comparative analysis of sysfilter and chestnut against manual methods, which might not fully represent the broader array of tools available in the field. Furthermore, the experimental setup was limited to Docker environments and might not entirely mimic the operational conditions found in varied real-world scenarios.

Implementing dynamic sandboxing effectively in high-performance environments poses several challenges, primarily related to the computational overhead and potential latency introduced by real-time system call monitoring and policy adjustment. This can be particularly problematic in environments that handle large volumes of transactions or data, where even minimal delays can accumulate significant operational impacts.

In conclusion, while the current tools and methodologies for system call sandboxing offer substantial security benefits, there is a compelling case for developing more advanced systems that integrate the precision of manual analysis with the efficiency of automated tools.

References

[1] David Brumley et al. Automatic detection of security vulnerabilities in system calls. *Journal of Computer Security*, 16(5):569–594, 2008.

[2] C. Canella et al. Automating seccomp filter generation for linux applications. In *Proceedings of the Cloud Computing Security Workshop (CCSW)*, Orlando, FL, 2021.

[3] Zakir Durumeric et al. The matter of heartbleed. *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, 2014.

[4] Dawson R. Engler et al. System call analysis and security. In *Proceedings of the IEEE Symposium on Security and Privacy*, Oakland, CA, USA, 2003.

[5] S. Ghavamnia et al. Confine: Automated system call policy generation for container attack surface reduction. In *Proceedings of the Annual Symposium on Research in Attacks, Intrusions, and Defenses (RAID)*, London, UK, 2020.

[6] Ian Goldberg et al. A secure environment for untrusted helper applications: Confining the wily hacker. In *Proceedings of the 6th USENIX Security Symposium*, San Jose, CA, USA, 1996.

[7] Sandeep Kumar et al. Complexity and vulnerability: Perspectives in software systems security. *IEEE Transactions on Software Engineering*, 36(4):516–529, 2010.

[8] Awad A Younis Mussa, Yashwant Malaiya, and Indrajit Ray. Assessing vulnerability exploitability risk using software properties. volume 24, page 11, 04 2015.

[9] Donald et al. Porter. Temporal system call specialization for attack surface reduction. In *Proceedings of the 2020 IEEE Symposium on Security and Privacy*, San Francisco, CA, USA, 2020.

[10] Niels Provos. Improving host security with system call policies. In *Proceedings of the 12th USENIX Security Symposium*, Washington, D.C., USA, 2003.

[11] Richard Smith et al. Eternalblue: A prominent tool in significant cyber incidents. *Journal of Internet Security*, 25(13):748–760, 2017.

A Appendices

A.1 Appendix One

Policies generated for *PWD*

- *Chestnut*: Found 41 syscalls [0, 1, 3, 4, 5, 6, 8, 9, 10, 11, 12, 13, 14, 15, 17, 20, 21, 28, 36, 38, 39, 60, 62, 63, 72, 79, 80, 81, 89, 96, 158, 164, 186, 201, 202, 217, 228, 231, 234, 257, 262] read write close stat fstat lstat lseek mmap mprotect munmap brk rtsigaction rtsigprocmask rtsigreturn pread64 writev access madvise getitimer setitimer getpid exit kill uname fcntl getcwd chdir fchdir readlink gettimeofday archprctl settimeofday gettid time futex getdents64 clockgettime exitgroup tgkill openat newfstatat
- *Sysfilter*: [0,1,3,4,5,6,7,8,9,10,11,12,13,14,15,16,20,24,25,28,32,39,41,42,44,45,47,49,51,54,60,62,72,78,79,80,81,96,99,186, 201,202,228,229,231,234,257,262,302]

A.2 Appendix Two

Policies generated for *NGINX*

- *Chestnut*: Found 107 syscalls [0, 1, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 23, 24, 28, 29, 30, 32, 33, 36, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 53, 54, 55, 56, 57, 59, 60, 62, 63, 67, 72, 77, 79, 80, 82, 83, 84, 86, 87, 89, 90, 92, 95, 96, 99, 102, 105, 106, 107, 110, 112, 125, 137, 141, 142, 144, 147, 149, 157, 158, 164, 186, 201, 202, 203, 213, 217, 218, 221, 228, 230, 231, 232, 233, 234, 235, 257, 262, 273, 288, 290, 296, 302, 318] read write close stat fstat lstat poll lseek mmap mprotect munmap brk rtsigaction rtsigprocmask rtsigreturn ioctl pread64 pwrite64 readv writev access select schedyield madvise shmget shmat dup dup2 getitimer setitimer getpid sendfile socket connect accept sendto recvfrom sendmsg recvmsg shutdown bind listen getsockname socketpair setsockopt getsockopt clone fork execve exit kill uname shmdt fcntl ftruncate getcwd chdir rename mkdir rmdir link unlink readlink chmod chown umask gettimeofday sysinfo getuid setuid setgid geteuid getppid setsid capget statfs setpriority schedsetparam schedsetscheduler schedgetprioritymin mlock prctl archprctl settimeofday gettid time futex schedsetaffinity epollcreate getdents64 settidaddress fadvise64 clockgettime clocknanosleep exitgroup epollwait epollctl tgkill utimes openat newfstatat setrobustlist accept4 eventfd2 pwritev prlimit64
- *Sysfilter*: [0,1,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,24,25 50,51,52,53,54,55,56,58,59,60,61,62,63,72,77,78,79,80,82,83,84,87,8 114,115,116,117,119,126,130,137,141,143,144,145,146,147,157,186 257,262,273,288,290,296,302,307,318]

A.3 Appendix Three

Generative AI (specifically ChatGPT) was used as an aid during writing of this paper. It was mainly used for the sake of explaining hard concepts to me, explaining C++ errors and LaTeXformatting. Some of the prompts that were used:

- Can you explain to me the difference between SCMPACTKILLPROCESS and SCMPACTKILL in the seccomp library?
- Can you format the following excel table into LaTeXone? (It was the table with aggregated syscalls per phase)
- I'm running into issues installing docker on Ubuntu 24.04 for rootless access, can you troubleshoot?
- Can you convert this citation into IEEE .bib format?