System Call Sandboxing: Enhancing Security Through Analysis

Comparing Dynamic and Static System Call Analysis for Diff and SSH

Duco de Bruin[1]

Supervisor: Alexios Voulimeneas[1]

[1]**EEMCS, Delft University of Technology, The Netherlands**

Abstract

Sandboxing is a technique that restricts software applications' access to system resources to limit unintended harmful behaviour. These measures may include limiting the number of system calls that can be used. This paper compares dynamic and static analysis methods for determining the necessary syscalls, focusing on the applications SSH and diff. The contributions of this research include a custom dynamic analysis approach, a comparison to an existing static solution, and insights into their strengths and weaknesses. Furthermore, it is explained how the use of execution phase separation, a technique that involves analyzing each phase of a program separately, can be used to further refine the system call set. The results of the experiments reveal that both methods can effectively decrease the attack surface, each eliminating over 60% of unnecessary system calls. On the one hand, static analysis covers all possible use-cases but includes calls that are never used. On the other hand, dynamic analysis provides a more realistic set based on actual use-cases, but may miss some edge cases. Moreover, it was found that execution phase separation works well and can reduce the amount of system calls required in the main working phase of SSH by 79%.

1 Introduction

Sandboxing is a way of creating an environment for a software application that has limited, controlled access to system resources. This is done to reduce the damage of unintended, potentially harmful behaviour. It limits the possibilities for adversaries to use an existing, compromised application for performing an attack on the rest of the system [1]. These restrictions may include reducing the amount of system calls that can be made. Linux has a built in functionality called Secure Computing (seccomp) that allows for specifying, per application, what system calls can be inferred with so called filters [2]. However, manually constructing these sets of allowed calls is time-consuming and error prone [3].

There are two primary methods for analysing programs to find the required subset of system calls. The first is static analysis, which involves inspecting the code and/or binary to find which system calls are expected to be used [3], [4], [5], [6]. Dynamic analysis, on the other hand, utilizes realistic scenarios to find which system calls are actually being used during runtime [5].

A tool that can be used to further refine the system call set for both static and dynamic analysis, is execution phase separation. It allows for even stricter sandboxes, since the restricted syscalls are adapted based on the execution phase of a program [7]. For instance, the 'execve' syscall is often only needed in the initialization phase and can be blocked after that. This tool ensures the tightest restriction at each stage of the execution.

Prior work has mainly focused on static analysis and using the results to create filters for seccomp [3], [4]. The 'Sysfilter' tool uses static analysis to find an overestimation of the required syscall set [4]. Notably, Canella et al. proposed a hybrid combination called 'Chestnut', which combines the different methods in a two phase approach. The first phase is used to find all system calls with static analysis, and the second phase is used to further refine this set with dynamic analysis [3]. Additionally, the benefits of execution phase separation have been extensively explored in literature, demonstrating its effectiveness in enhancing sandboxing mechanisms [4], [6], [7]. Another technique for reducing a program's attack surface is software debloating, which utilizes dynamic analysis to identify and remove unnecessary code segments, thus enhancing its security [8]. Research has also extended sandboxing techniques to Docker containers, with various strategies enhancing container security [5], [9].

This research attempts to provide an understanding of the differences between static and dynamic analysis techniques for creating syscall filters. Furthermore, it explores how execution phase separation can provide an even stricter policy. To address these objectives, this paper poses the following research question:

"How do dynamic and static analysis methods compare in identifying required system calls for applications, and how can execution phase separation further refine the system call sets?"

In order to achieve these objectives, this paper will be limited to two specific applications: diff and SSH [10], representing varying complexities.

The contributions of this research include a simple, custom approach for dynamic analysis using Strace [11], [10]. This custom method and the static analysis tool Sysfilter are then used to conduct syscall analysis on diff and SSH. The resulting sets of both will be analysed and compared, giving insights into their strengths and weaknesses. Moreover, it is explored how the use of execution phase separation can be used for SSH. This will involve identifying different phases in SSH as well as finding the system calls per phase.

The structure of this paper is as follows: Section 2 outlines the methodology used for the experiments, explaining the tools and applications involved. Section 3 presents the results of the dynamic and static analysis. Section 4 describes the process and results of execution phase separation for SSH. Section 5 compares the findings and discusses the limitations of the two methods. Section 6 briefly addresses responsible research. Finally, section 7 provides the conclusions and gives suggestions for future work.

2 Methodology

This section focuses on the methodology of this research. First, the selected applications for the analysis will be listed. Then, the tools used for both dynamic and static analysis will be explained. Next, the environment in which the experiments were conducted is discussed. Finally, the approach for execution phase separation will be explained.

2.1 Application Selection

To limit the scope of this research, two applications were chosen to be analysed.

The first one is diff [10], a small binary that compares files line by line, indicating any differences between them. Due to its relatively simple functionality and codebase, diff is a good first candidate for analysis, since the amount of syscalls it uses will most likely also be limited. Additionally, the number of execution paths for dynamic analysis is not very high. This simplicity makes diff an ideal starting point for understanding the basics of system call sandboxing without the complexity of larger programs.

The second application that was selected is Secure Shell (SSH) [10], a protocol for secure remote login and command execution over an encrypted connection. Unlike diff, SSH has a much more complex codebase and functionality, including network communication and encryption mechanisms. This complexity makes SSH another excellent candidate for analysis because it allows evaluation of how larger applications benefit from system call sandboxing. Moreover, SSH consists of multiple execution phases, which will be explored later in this paper.

2.2 Strace

Dynamic analysis of the applications will be done using the Strace [11], [10] tool. It is a system call tracer that intercepts and records the system calls made by a selected program during execution. By running an application under Strace, a detailed log of all system calls, along with their parameters and return values, is generated.

For the sake of this research, the focus is solely on the syscall names. Therefore, the data needs to be parsed for easier evaluation. To achieve this, a simple utility called sparse was developed in C++, which filters out unnecessary information and ensures that duplicate entries are removed. The resulting set of system calls is then presented for analysis. On top of that, a shell script (traceparse.sh) was written to streamline the process, allowing for the execution of Strace and sparse with a single command and storing the results in a file. The code and documentation for both tools can be found on GitHub [12].

To properly find all the system calls that the chosen applications require, it is essential to run the tracer on a variety of inputs to cover as many possible execution paths. Once all of these traces are obtained and parsed, they need to be combined into a single dataset. To achieve this, a tool called unique-file-merge was developed in C++. This tool generates the final set, which contains all the system calls needed, by taking the union of the smaller sets. The code and documentation of this tool can be found on GitHub [13].

2.3 Sysfilter

Sysfilter is a static binary analysis tool for x86-64 Linux which can be used to generate system call filters [4]. It consists of two main components: an extraction tool and a filter enforcement tool. For this research, the focus lies on the extraction tool, which can be used to identify the system calls that are likely to be used by a program, given its binary. This tool provides an over-approximation of the calls invoked by the application.

The result of running Sysfilter extraction on a binary is a JSON file containing a list of numbers that correspond to specific syscalls. In order to be able to compare this with the results from dynamic analysis, it is required to parse this file by replacing the numbers with their corresponding syscall names. This will be done with a Python program, using the list of syscalls for x86-64 provided in Strace's GitLab [11]. The result of this is a JSON file containing the list of syscall names. The program that converts the syscall numbers to names, including code and documentation, can be found on GitHub [14].

2.4 Environment Selection

All experiments were conducted on the x86-64 architecture [15] using Ubuntu 22.04 LTS [16]. The primary reason for this choice is that Sysfilter was developed for the x86-64 architecture. This ensures that the tool will work as intended. Ubuntu 22.04 LTS was selected due to its stability and ease of use.

2.5 Execution Phase Seperation

The methodology for execution phase separation involves manually inspecting the syscall traces produced by Strace for the SSH client. This process includes analyzing the sequence of system calls to identify patterns or transitions that differentiate between the various phases of execution. To help with this identification, the execution of the program was manually stopped at key points:

1. **Before Logging In:** The execution was paused before actually logging in, to capture the initial setup and initialization syscalls.
2. **Directly After Logging In:** The execution was stopped immediately after the login process to capture the syscalls associated with the session establishment.
3. **After Performing Tasks:** Finally, the execution was paused after performing several tasks to capture the working phase syscalls.

This makes it easier to identify which phases can be distinguished and which syscalls belong to each. It could also be useful to look at or modify the code to identify key transition points, but that is outside the scope of this paper.

3 Analysis Results

This section presents the findings from the dynamic analysis of system calls on diff and SSH using Strace [11], [10]. Furthermore, the results from static analysis using Sysfilter [4] will be shown. For both static and dynamic analysis, the percentage of system calls compared to the total number of system calls present in x86-64 is shown in figure 1 per application.

3.1 Diff

Dynamic Analysis

Dynamic analysis on the diff application revealed a set of system calls that are likely to be needed during its execution. This set was discovered using the methodology outlined in the previous section.

It is to be noted that this set may not be complete as it was obtained on a limited number of inputs and might, therefore, miss some edge cases. However, the most important functionalities of diff were used to generate these sets so it is likely close to complete and can therefore be used for further analysis.

The dynamic analysis resulted in a set of 24 system calls. This is only 7% of all 335 system calls available in the x86-64 architecture. This relatively small percentage indicates that diff only utilizes a small subset of the overall system functionality. This was, of course, expected, as it only has limited functionality.

The calls found cover basic operations like file handling (e.g., openat, read, write, getdents64), process control (e.g., execve), memory management (e.g., brk, mmap, mprotect, munmap) and some other essential functions.

The complete resulting set of system calls found by the dynamic analysis can be found in Appendix A. Using this set, a filter could be made in order to sandbox this application. By restricting syscalls to this specific subset, the attack surface would be significantly reduced without compromising the application's main functionality.

Static Analysis with Sysfilter

In contrast with dynamic analysis, static analysis resulted in a significantly larger set of system calls. This was to be expected, as Sysfilter is designed to over-estimate the set of syscalls needed by the application [4]. However, this also means that we are certain that at least all the necessary syscalls are included. It is also noteable that the analysis was very easy and didn't require doing any manual work.

The static analysis resulted in a total of 59 system calls being identified. This is only 18% of all 335 system calls available in the x86-64 architecture. This is more than the dynamic analysis yielded, but this is expected. It is still small enough to reduce the attack surface significantly when using the set as a filter.

The calls cover operations like file handling (e.g., openat, read, write, getdents64), process control (e.g., execve), memory management (e.g., brk, mmap, mprotect, munmap), system information (e.g., arch_prctl, uname, prlimit64, sysinfo), networking (e.g., socket, connect, sendto, recvfrom) and many more.

The complete resulting set of system calls found by the static analysis can be found in Appendix A.

3.2 SSH

Dynamic Analysis

Dynamic analysis of the SSH application produced a larger set of system calls than the one for diff. This was expected, considering that SSH is a much more complex application with functionalities like networking and encryption.

It was decided to split the result into two sets, since there is a significant difference between the two use-cases of SSH. One set is the result of using SSH as a client: remoting into another device. The other set was the result of using SSH as a server, where another device was used to remote into the device under test. By dividing the analysis into client and server components, the difference in system call behavior of each aspect of the SSH application can easily be analysed.

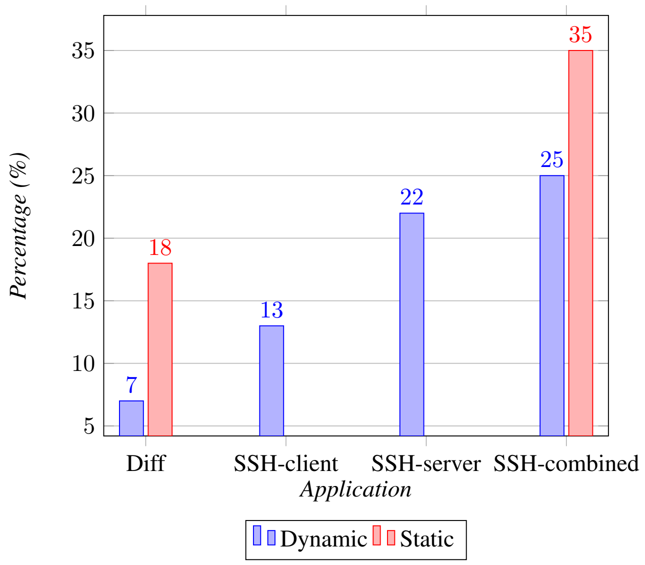

Figure 1: Percentage of all x86-64 system calls required per application, per analysis method.

It is important to note that the results obtained from dynamic analysis might not be complete because certain paths might have been missed. However, the main functionalities were used for testing so it is likely to be close to complete.

Analysing the clientside part of SSH was straightforward, as Strace could be invoked on the application directly. SSH was then used to remote into another machine, where various tasks were performed to get as many required system calls as possible. Notably, the specific tasks performed on the remote machine did not impact the resulting set of system calls.

It was revealed that SSH requires 43 system calls when used to remotely access another machine. This is only 13% of the total system calls. Considering that this application has complex functionalities, this is still a relatively small amount. This suggests that there are many system calls that are only required in very specific use-cases. It also means that system call sandboxing is likely still effective in reducing the attack surface significantly.

This list of system calls covers a wide range of operations. These include networking (e.g., socket, connect, setsockopt), file handling (e.g., openat, read, write, getdents64, lseek), process control (e.g., execve, exit_group getpid), memory management (e.g., brk, mmap, mprotect, munmap), polling (e.g., poll ppoll) and many more.

Analyzing the server-side aspect of SSH presented some challenges, mainly because the program runs in the background. It was therefore necessary to identify the process ID (pid) of the 'sshd' process. This was achieved using 'ps aux' and locating the process names: 'sshd: /usr/sbin/sshd'. Strace was then invoked on this process to record the system calls. The purpose of this specific process is to actively listen for incoming connections and establish them. Upon connection by an external device, a new process is spawned, identifiable by the name: 'sshd: username [priv]', where username represents the user that was remoted into. Subsequently, Strace was applied to this process to capture the rest of the system calls.

This analysis resulted in a set of 74 system calls. Even this is only 22% of all system calls available. This result indicates that most work for the remote connection is done on the server side, on the host. The list of system calls covers mainly networking(e.g., accept, bind, setsockopt, recvmsg), but also thread management (e.g., futex, tgkill, sched_getaffinity), polling (e.g., poll, ppoll), user management(e.g., chown, setfsuid, setgroups) and much more.

The final resulting set of system calls for both client and server side as well as the combined set, can be found in Appendix A.

static Analysis with Sysfilter

Unlike the dynamic analysis for SSH, it was not possible to split the resulting set of system calls into a client and server part. This is, because there is only one binary to analyze. Either way, the results are still useful for sandboxing the entire application.

When applying Sysfilter to the binary for SSH, some challenges were presented. Despite having all the required symbols installed, the tool was unable to parse the binary. This issue is known to occur with certain hand-coded assembly functions and requires a manual override. The Egalito framework, which Sysfilter largely relies on for its binary analysis, sometimes struggles with these kinds of functions [4]. However, the tool provides a couple of overrides for some common libraries, and using these overrides resolved the problem in this case. The tool was therefore executed with the following command to address the parsing issue:

```
EGALITO_PARSE_OVERRIDES=overrides
app/build_x86_64/sysfilter_extract /usr/bin/ssh
```

The result of the analysis was a set of 111 system calls, which is 35% of all system calls available in x86-64. This is a large percentage compared to the previous results, but it is still a relevant reduction when used for sandboxing. The list of system calls cover inter-process communication(e.g., pipe, dup), networking(e.g., bind, listen, connect, sendmsg), scheduling(e.g., schet_getpararm, schet_get_priority_max) and timers(e.g., timerfd_settime) and many more.

The complete set of system calls can be found in Appendix A

4 Execution phase seperation

For SSH, it was possible to identify multiple execution phases and find system call sets per phase. The process and outcomes for this are described in the following section.

4.1 Phases in SSH

As explained in the methodology, by manually stopping the execution of the SSH client at several key points, it was possible to identify several phases. By inspecting the traces, it was possible to find their respective functionalities. The phases are as follows:

1. **Initialization Phase:** In this phase, the client prepares all necessary components to establish a connection and then initiates the connection to the server.
2. **Authentication Phase:** The user is prompted to enter their credentials. These are then verified by the server.
3. **Session Establishment Phase:** Once the user is verified, the session is created. This includes some extra preparation.
4. **Working Phase:** This is the main part of the application where the actual tasks are performed. The user can now interact with the remote system. For simplicity, this also includes exiting the application.

It was challenging to determine the boundaries of the authentication phase due to its smaller size and close integration with the initialization phase. Therefore, it was decided to be included as part of the initialization phase.

Moreover, it may be possible that there are more smaller phases within the identified phases. However, it was decided not to further separate them, as the results of using just 3 phases was already very promising.

4.2 Phases in Traces

The phases can be identified by specific system calls, which mark their beginnings and endings. It was decided to use these seemingly logical points, but the boundaries could also be slightly shifted.

The initialization phase starts at the beginning of the process and ends after the user has entered their password. This can be recognised by two consecutive syscalls that are reading and then printing a newline character:

```
read(5, "\n", 1)  = 1
write(5, "\n", 1) = 1
```

the session establishment phase occurs after this and before the next phase.

The working phase begins when the first text is printed to the client console. For instance, when remoting into a Raspberry Pi, this is indicated by the following system call:

```
write(5, "Linux rpi 6.1.21-v7+ #1642
SMP M"..., 418) = 418
```

4.3 Resulting Syscall Sets

Having identified these phases in the traces, it is now possible to identify the required system calls. The sparse [12] program was used on the subsets of the traces to obtain the syscall sets. This resulted in three, partially distinct sets of varying sizes, which can be found in Appendix B.

Figure 2 provides a comparison of the percentage of system calls required per phase. This is not only shown in comparison with the total number of system calls in x86-64 but also in relation to the number of system calls required for the entire SSH-client.

The initialization phase required the most system calls, with 36, which is 86% of the system calls required for SSH client. This suggests that the setup process involves a substantial number of system interaction, likely related to establishing connections and security measures.

In contrast, the session establishment phase requires only 19 system calls, representing 45% of the total calls typically required by the SSH client. This shows that it still requires a

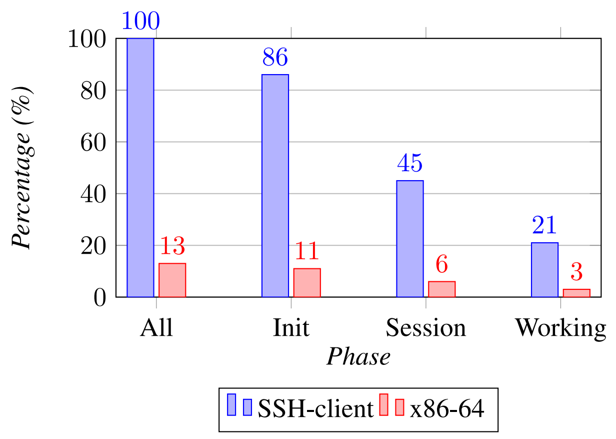

Figure 2: Percentage of system calls required per phase of SSH. Compared to either the number of required system calls for SSH-client or to all available in x86-64.

lot of system interactions, but this is expected since this phase is also part of the setup.

Notably, the working phase is the smallest set, only requiring 9 system calls. This indicates that most system calls are required during setups and not necessarily during the usage of the application. Only 21% of the system calls normally needed for the SSH client are required. This is a considerably small portion, thus significantly reducing the potential vulnerabilities during the application's main usage.

5 Discussion

The results show how dynamic and static analysis can be used to find a set of required system calls for an application. This section will compare the dynamic and static analysis and discuss limitations of both methods. A tool that combines both methods will be discussed as well.

5.1 Comparison

Both static and dynamic analysis provide valuable insights into the required system calls of applications. They do, however, function in different ways and produce different results.

Diff

Dynamic analysis results in a much smaller set of required system calls than static analysis with Sysfilter [4]. As can be seen in Figure 1, there is a difference of 11% between the two sets. It is interesting to see that, for diff, the dynamic set is not necessarily a subset of the static set. It is largely a subset, but there are some minor differences.

Specifically, there are four calls that appear in the result of dynamic analysis but not in the result of static analysis. These are:

- arch_prctl: This system call is architecture-specific and is used for thread management. It might not have been captured by the static analysis because it is dependent on the runtime environment, which is not predictable from the code [10].
- set_tid_address: It sets a pointer to the thread id. Similar to arch_prctl, this call is tied to the runtime behaviour, which cannot be predicted beforehand [10].
- access: This checks the user's permission for a file. This call might be invoked by the operating system instead of the application, so it is not evident in the code [10].
- rseq: Standing for Restartable Sequences, this system call also has to do with thread management. Therefore, it can also not be predicted beforehand [10].

As it appears, some system calls handling thread management and access rights are not able to be detected by static analysis as they are likely only called by the runtime environment. This could also mean that they are not always called but only in specific situations.

The set of system calls that was only found by static analysis is less interesting. This set contains 39 extra system calls, which are mostly not actually used by diff. It contains some calls that are certainly wrong, like several regarding networking(e.g., socket, connect, sendto, recvfrom). This type of call is unexpected, considering that diff operates on local files and doesn't require network functionality. This unexpected inclusion shows the tendency of Sysfilter to overestimate the required system calls. Leaving these system calls available in a sandbox, while not required, could lead to extra vulnerabilities. This is important to consider when using this static analysis tool.

On the contrary, there are some syscalls that could be right, like rt_sigreturn, which handles signaling [10]. This call may not have been found by dynamic analysis because no signals were used during execution, highlighting the downside of dynamic analysis.

SSH

Again, static analysis with Sysfilter yielded much more system calls required than dynamic analysis. The difference, similar to diff, is around 10%, which could indicate that this is the overhead of the tool. However, more applications would need to be tested to confirm this hypothesis. Even though the percentage differences are nearly the same, the actual variation between the sets is even larger than for diff. This is expected, as SSH already requires more system calls.

To begin with the system calls that appear in the result of dynamic analysis but not in the result of static analysis. There are 20 of these, which can be split into 2 main groups:

- Process/Thread Management: restart_syscall, clone3, setpriority, setfsuid, capset, close_range, rseq, arch_prctl, setfsgid, capget, getpriority
- I/O Operation: epoll_ctl, ppoll, timerfd_settime, epoll_wait, epoll_create1, timerfd_create, readlinkat, chown

Although there are more syscalls of each type, the overall result is comparable to the findings for diff. Once again, many process and thread management calls are not found by the static analysis, due to the dependency on the runtime environment or conditions.

For example, ppoll_create may have been left undetected because it is a Linux-specific API that depends on the environment and is thus not used on every operating system. Also chown, capable of changing ownership of a file, may not have been found by static analysis due to its dependency on the runtime conditions. Additionally, rseq remains undetected by Sysfilter, as seen with diff, since it is unpredictable just from the code.

There were many calls that were only found by Sysfilter and not by dynamic analysis. There were a total of 47 extra calls, that are mainly due to the overestimation of the tool. Some examples of system calls which are likely not required by SSH in reality are: mkdir, sync_file_range, gettimeofday, add_key, but also many more. These are system calls that don't seem to be in line with the functionalities of SSH. Their presence is due to dependencies or libraries containing them, but they may not be actively used in the program's execution.

Some system calls could be right, however. Some examples of this are: exit, listen, select and likely some more. These seem to correspond to the behaviour of SSH. Especially, the 'listen' system call seems to be crucial when waiting for clients to connect to the server, so it is odd that dynamic analysis did not detect it.

5.2 Limitations of Dynamic Analysis

Dynamic analysis produces a set of syscalls necessary for realistic use-cases of applications. This means that the resulting number of system calls will be as limited as possible. Which is a good thing, since this will reduce the attack surface of the application significantly.

On the contrary, it is hard to guarantee that all possible execution paths were followed during analysis. It might therefore occur that an application in a sandbox with this set as a filter might not function properly in some scenarios. Nevertheless, with a sufficiently large and diverse set of inputs, a near-perfect set of system calls can be achieved, ensuring reliable performance within a sandbox. This does, unfortunately, require a lot of manual labour and proper insights into the execution of applications.

A challenge of dynamic analysis is the difficulty in determining the best approach to analyzing certain programs. For example, a background task like SSH or other services might consist of multiple small components, which will all need to be analysed individually in order to obtain complete results. This makes the analysis process more complicated and prone to errors.

5.3 Limitations of Static Analysis

Static analysis, on the other hand, is guaranteed to provide a complete set of system calls for the analysed application. Every possible system call that can be called from the code will be found. This means that using this set as a filter for a sandbox will most likely not limit the functionalities of the application. Consequently, it also means that system calls, which might never be reached, will also be found. These syscalls will therefore also be included in the filter for a sandbox, leading to a larger vulnerability gab for adversaries to exploit.

In certain cases, like with diff, Sysfilter is easy to use. It only requires the execution of one command while providing the path to the binary. However, this is not always the case, as many larger applications like SSH require the correct debug symbols to be installed on the system for the analyser to be able to understand the binaries. This requires manually identifying which debug packages are used by a certain application and then installing them. Even that doesn't always work, and then it is necessary to manually provide overrides for specific functions. In some cases, these overrides are provided by the tool itself, but in other cases a lot of manual work and research will be required to find them.

5.4 Combining Both Methods

An effective method that addresses the limitations of both approaches, involves combining the two. This is done by a tool known as Chestnut [3]. Initially, Chestnut employs standard static analysis on the binary (p1), just like Sysfilter [4]. The difference comes in the second phase (p2), where dynamic analysis is used to reduce the set of system calls even more.

For this study, it was attempted to use Chestnut on diff and SSH in order to gain insights in what results this approach might yield. Unfortunately, Chestnut's documentation proved to be inadequate, making it hard to use. Luckily, the paper by Canella et al. provides some results that can be used for this comparison.

According to their findings, which can be seen in comparison with dynamic analysis and static analysis by Sysfilter in Figure 3, static analysis of the binary of SSH resulted in 36 system calls. This set was then reduced to 16 through the second, dynamic phase.

It is worth noting that the binary analysis by Sysfilter produced almost two times more system calls than the analysis by Chestnut. A possible explanation for this difference, is that Sysfilter does a deeper analysis of the libraries that are potentially used in the code and thus finds more (not actually required) syscalls.

Even more intriguing is that the final result after p2 is smaller than the set found by dynamic analysis. This is puzzling, because all calls identified by the dynamic analysis proposed by this study are sure to be used, as they were discovered while the application was running. One possible explanation is that certain system calls were not considered by Canella et al. Even then, it is still hard to explain why the difference is so big and what was not considered. Unfortunately, without inspecting the actual set of syscalls found by Chestnut, the answer cannot be provided.

While these results were not available for SSH, the observations from diff indicate that the resulting set would also be smaller. This could very well be the case for most application, since this method seems to combine the best of both static and dynamic analysis, yielding tight result sets.

6 Responsible Research

6.1 Ethical Concerns

All tools and methods used in this paper were designed to avoid harming existing systems. The aim of this research was to contribute positively to cybersecurity. Most importantly, all experiment results were reported accurately and without falsification.

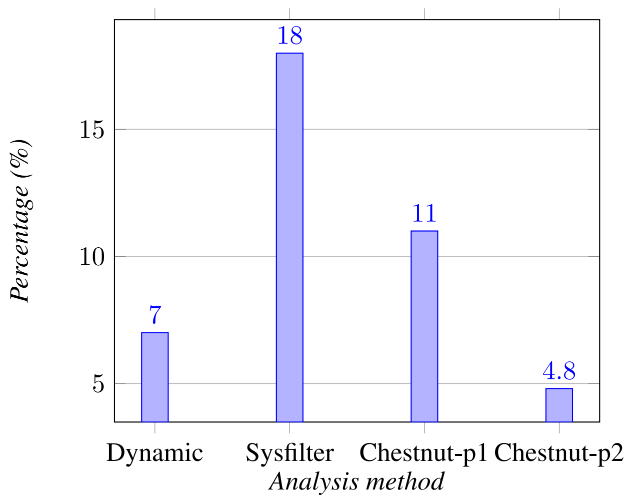

Figure 3: Percentage of all x86-64 system calls required for diff per analysis method.

6.2 Reproducibility

To ensure reproducibility, all steps of the experiments were documented. All tools and software used, including sysfilter, SSH and custom-developed tools like sparse, are publicly available. For the custom-developed tools, all code is publicly available, well commented and Readme files with usage instructions are provided. This transparency enables other researcher to replicate and verify this study.

7 Conclusions and Future Work

In conclusion, this paper has explored how approaches for dynamic and static analysis of system calls compare with regards to system call sandboxing. By evaluating a custom dynamic analysis method using Strace [11], [12], [13] against the static analysis tool Sysfilter [4], this study provides insight into their strengths and weaknesses.

The results indicate that dynamic and static analysis significantly reduce the attack surface, with each method eliminating more than 60% of unnecessary system calls for both SSH and diff. This indicates that using either method for sandboxing would result in a safer environment with a reduced attack surface.

Static analysis identified a larger set of system calls, covering every possible scenario. However, it also found additional calls that are present in libraries, for example, but are never actually used by the application. In contrast, dynamic analysis provided a tighter and more realistic set of calls based on actual behaviour. The downside of dynamic analysis is that the resulting set might miss some edge cases, overlooking rarely used calls. Thus, when using either method for sandboxing, there is a trade-off between maintaining full functionality and achieving maximum attack surface reduction.

To maximize security, a hybrid approach combining both methods, such as used by the Chestnut [3] tool, appears promising. This method mitigates the drawbacks of the individual approaches, while leveraging their strengths.

Moreover, using in depth techniques such as execution phase separation can further enhance system call reduction, particularly for larger applications like SSH. By ensuring that only the essential system calls are available in each phase, execution phase separation offers an even better approach to reducing attack surface reduction. It was found that the working phase of the SSH client only requires 21% of what the entire application normally requires.

7.1 Future Work

While this research offers insight into how effective dynamic and static analysis are, several aspects remain to be explored for future research. To gain even more insights in how each method performs, future research could extend the analysis to a broader range of applications.

Additionally, incorporating fuzzing techniques, as suggested by Canella et al. [3], could be useful in ensuring full coverage of all execution paths during dynamic analysis. By subjecting applications to diverse inputs, fuzzing can uncover potential edge cases that may not be captured by manual analysis.

While this study primarily focused on identifying system calls and quantifying the reduction they would yield, there is potential value in developing a tool. It should be capable of automatically performing the explained custom dynamic analysis, as well as filter creation for sandboxing, solely based on a set of provided inputs.

The manual identification of execution phases is a labor-intensive and error-prone task. A tool that automates this is Syspart, which employs static and dynamic analysis to sandbox phases separately for several server applications [17]. Future work should explore methods of automating this process for other application. This could potentially also be done using machine learning to identify key transition points.

This study observed approximately a 10% overhead of system calls for Sysfilter compared to dynamic analysis. Future research should investigate whether this overhead remains consistent across multiple applications and how it can be reduced.

References

[1] V. Prevelakis and D. Spinellis, "Sandboxing applications," in *2001 USENIX Annual Technical Conference (USENIX ATC 01)*, (Boston, MA), USENIX Association, June 2001.

[2] "A seccomp overview." LWN.net [Online]. Available: https://lwn.net/Articles/656307/. (accessed May 5, 2024).

[3] C. Canella, M. Werner, D. Gruss, and M. Schwarz, "Automating seccomp filter generation for linux applications," in *CCSW 2021*, 2021.

[4] N. DeMarinis, K. Williams-King, D. Jin, R. Fonseca, and V. P. Kemerlis, "sysfilter: Automated system call filtering for commodity software," in *RAID 2020*, 2020.

[5] S. Ghavamnia, T. Palit, A. Benameur, and M. Polychronakis, "Confine: Automated system call policy generation for container attack surface reduction," in *RAID 2020*, 2020.

[6] S. Ghavamnia, T. Palit, S. Mishra, and M. Polychronakis, “Temporal system call specialization for attack surface reduction,” in *USENIX Security 2020*, 2020.

[7] Z. et al., “Building dynamic system call sandbox with partial order analysis,” in *Proceedings of the ACM on Programming Languages*, vol. 7, 2023.

[8] Q. et al., “Razor: A framework for post-deployment software debloating,” in *Proceedings of the 28th USENIX Security Symposium*, 2019.

[9] S. Yang, B. B. Kang, and J. Nam, “Optimus: association-based dynamic system call filtering for container attack surface reduction,” *Journal of Cloud Computing*, vol. 13, 2024.

[10] M. Kerrisk, “man7.” [Online]. Available: https://man7.org. (accessed Apr. 28, 2024).

[11] D. Levin, “Strace.” GitLab. [Online]. Available: https://gitlab.com/strace/strace. (accessed May 27, 2024).

[12] D. de Bruin, “sparse.” GitHub. [Online]. Available: https://github.com/DucodB/sparse. (accessed Apr. 28, 2024).

[13] D. de Bruin, “unique-file-merge.” GitHub. [Online]. Available: https://github.com/DucodB/unique-file-merge. (accessed May 4, 2024).

[14] D. de Bruin, “syscall-numbertoname.” GitHub. [Online]. Available: https://github.com/DucodB/syscall-numbertoname. (accessed May 27, 2024).

[15] R. Chapman, “Linux system call table for x86_64.” R. Chapman’s Blog. [Online]. Available: https://blog.rchapman.org/posts/Linux_System_Call_Table_for_x86_64/. (accessed May 3, 2024).

[16] “Jammy jellyfish release notes.” Ubuntu. [Online]. Available: https://discourse.ubuntu.com/t/jammy-jellyfish-release-notes/24668. (accessed May 17, 2024).

[17] V. L. Rajagopalan, K. Kleftogiorgos, E. Göktaş, and G. P. Jun Xu, “Syspart: Automated temporal system call filtering for binaries,” in *CCS 2023 - Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, 2023.

A Results of Experiments

A.1 System Call List from Dynamic Analysis on Diff

ACCESS, ARCH_PRCTL, BRK, CLOSE, EXECVE, EXIT_GROUP, FCNTL, GETDENTS64, GETRANDOM, LSEEK, MMAP, MPROTECT, MUNMAP, NEWFSTATAT, OPENAT, PREAD64, PRLIMIT64, READ, RSEQ, RT_SIGACTION, SET_ROBUST_LIST, SET_TID_ADDRESS, SIGALTSTACK, WRITE.

A.2 System Call List from Static Analysis on Diff

ACCESS, ARCH_PRCTL, BRK, CLOSE, EXECVE, EXIT_GROUP, FCNTL, GETDENTS64, GETRANDOM, LSEEK, MMAP, MPROTECT, MUNMAP, NEWFSTATAT, OPENAT, PREAD64, PRLIMIT64, READ, RSEQ, RT_SIGACTION, SET_ROBUST_LIST, SET_TID_ADDRESS, SIGALTSTACK, WRITE, POLL, IOCTL, WRITEV, SCHED_YIELD, MREMAP, MINCORE, MADVISE, DUP, DUP2, GETPID, SOCKET, CONNECT, SENDTO, RECVFROM, RECVMSG, BIND, GETSOCKNAME, SETSOCKOPT, CLONE, EXIT, WAIT4, FCNTL, GETCWD, READLINK, GETTIMEOFDAY, SYSINFO, SIGALTSTACK, SCHED_GETPARAM, SCHED_SETSCHEDULER, SCHED_GETSCHEDULER, SCHED_GET_PRIORITY_MAX, SCHED_GET_PRIORITY_MIN, GETTID, TIME, FUTEX, SCHED_GETAFFINITY, GETDENTS64, CLOCK_GETTIME, EXIT_GROUP, TGKILL, OPENAT, NEWFSTATAT, SET_ROBUST_LIST, PIPE2, PRLIMIT64, GETRANDOM

A.3 System Call List from Dynamic Analysis on SSH

Clientside

ACCESS, ARCH_PRCTL, BRK, CLOSE, CLOSE_RANGE, CONNECT, DUP, DUP2, EXECVE, EXIT_GROUP, FCNTL, FUTEX, GETDENTS64, GETPEERNAME, GETPID, GETRANDOM, GETSOCKNAME, GETSOCKOPT, GETUID, IOCTL, LSEEK, MMAP, MPROTECT, MUNMAP, NEWFSTATAT, OPENAT, POLL, PPOLL, PREAD64, PRLIMIT64, READ, RSEQ, RT_SIGACTION, RT_SIGRETURN, SET_ROBUST_LIST, SET_TID_ADDRESS, SETSOCKOPT, SOCKET, STATFS, UMASK, UNAME, UNLINK, WRITE

Serverside

ACCEPT, ACCESS, ALARM, BIND, BRK, CAPGET, CAPSET, CHOWN, CLONE, CLONE3, CLOSE, CONNECT, DUP2, EPOLL_CREATE1, EPOLL_CTL, EPOLL_WAIT, EXIT_GROUP, FCNTL, FTRUNCATE, FUTEX, GETDENTS64, GETEGID, GETEUID, GETGID, GETGROUPS, GETPEERNAME, GETPID, GETPRIORITY, GETRANDOM, GETSOCKNAME, GETSOCKOPT, GETTID, GETUID, IOCTL, KEYCTL, LSEEK, MMAP, MPROTECT, MUNMAP, NEWFSTATAT, OPENAT, PIPE2, POLL, PPOLL, PREAD64, PRLIMIT64, READ, READLINK, READLINKAT, RECVFROM, RECVMSG, RENAME, RESTART_SYSCALL, RT_SIGACTION, RT_SIGPROCMASK, RT_SIGRETURN, SENDMSG, SENDTO, SETFSGID, SETFSUID, SETGROUPS, SETPRIORITY, SETREGID, SETRESGID, SETRESUID, SETREUID, SETSOCKOPT, SOCKET, SOCKETPAIR, TIMERFD_CREATE, TIMERFD_SETTIME, UMASK, WAIT4, WRITE

Combined

ACCEPT, ACCESS, ALARM, ARCH_PRCTL, BIND, BRK, CAPGET, CAPSET, CHOWN, CLONE, CLONE3, CLOSE, CLOSE_RANGE, CONNECT, DUP, DUP2, EPOLL_CREATE1, EPOLL_CTL, EPOLL_WAIT, EXECVE, EXIT_GROUP, FCNTL, FTRUNCATE, FUTEX, GETDENTS64, GETEGID, GETEUID, GETGID, GETGROUPS, GETPEERNAME, GETPID, GETPRIORITY, GETRANDOM, GETSOCKNAME, GETSOCKOPT, GETTID, GETUID, IOCTL, KEYCTL, LSEEK, MMAP, MPROTECT, MUNMAP, NEWFSTATAT, OPENAT, PIPE2, POLL, PPOLL, PREAD64, PRLIMIT64, READ, READLINK, READLINKAT, RECVFROM, RECVMSG, RENAME, RESTART_SYSCALL, RSEQ, RT_SIGACTION, RT_SIGPROCMASK, RT_SIGRETURN, SENDMSG, SENDTO, SET_ROBUST_LIST, SET_TID_ADDRESS, SETFSGID, SETFSUID, SETGROUPS, SETPRIORITY, SETREGID, SETRESGID, SETRESUID, SETREUID, SETSOCKOPT, SOCKET, SOCKETPAIR, STATFS, TIMERFD_CREATE, TIMERFD_SETTIME, UMASK, UNAME, UNLINK, WAIT4, WRITE

A.4 System Call List from Static Analysis on SSH

READ, WRITE, CLOSE, STAT, FSTAT, LSTAT, POLL, LSEEK, MMAP, MPROTECT, MUNMAP, BRK, RT_SIGACTION, RT_SIGPROCMASK, RT_SIGRETURN, IOCTL, PREAD64, READV, WRITEV, ACCESS, PIPE, SELECT, SCHED_YIELD, MREMAP, MADVISE, DUP, DUP2, PAUSE, NANOSLEEP, ALARM, GETPID, SOCKET, CONNECT, ACCEPT, SENDTO, RECVFROM, SENDMSG, RECVMSG, SHUTDOWN, BIND, LISTEN, GETSOCKNAME, GETPEERNAME, SOCKETPAIR, SETSOCKOPT, GETSOCKOPT, CLONE, VFORK, EXECVE, EXIT, WAIT4, KILL, UNAME, FCNTL, FLOCK, FSYNC, FTRUNCATE, GETDENTS, GETCWD, CHDIR, RENAME, MKDIR, RMDIR, LINK, UNLINK, READLINK, CHMOD, FCHMOD, UMASK, GETTIMEOFDAY, SYSINFO, GETUID, GETGID, SETUID, SETGID, GETEUID, GETEGID, SETSID, SETREUID, SETREGID, GETGROUPS, SETGROUPS, SETRESUID, SETRESGID, STATFS, SCHED_GETPARAM, SCHED_SETSCHEDULER, SCHED_GETSCHEDULER, SCHED_GET_PRIORITY_MAX, SCHED_GET_PRIORITY_MIN, PRCTL, SYNC, GETTID, TIME, FUTEX, SCHED_SETAFFINITY, SET_TID_ADDRESS, CLOCK_GETTIME, CLOCK_GETRES, EXIT_GROUP, TGKILL, ADD_KEY, KEYCTL, OPENAT, NEWFSTATAT, SET_ROBUST_LIST, PIPE2, PRLIMIT64, SENDMMSG, GETRANDOM

B Results of Execution Phase Separation

B.1 System Call List for Initialization Phase SSH

ACCESS, ARCH_PRCTL, BRK, CLOSE, CLOSE_RANGE, CONNECT, EXECVE, FCNTL, FUTEX, GETDENTS64, GETPEERNAME, GETPID, GETRANDOM, GETSOCKNAME, GETUID, IOCTL, LSEEK, MMAP, MPROTECT, MUNMAP, NEWFSTATAT, OPENAT, PPOLL, PREAD64, PRLIMIT64, READ, RSEQ, RT_SIGACTION, SET_ROBUST_LIST, SET_TID_ADDRESS, SETSOCKOPT, SOCKET, STATFS, UMASK, UNAME, WRITE

B.2 System Call List for Session Establishment Phase SSH

CLOSE, DUP, DUP2, FCNTL, GETPID, GETSOCKNAME, GETSOCKOPT, IOCTL, MMAP, NEWFSTATAT, OPENAT, POLL, PPOLL, READ, RT_SIGACTION, SETSOCKOPT, UMASK, UNLINK, WRITE

B.3 System Call List for Working Phase SSH

CLOSE, EXIT_GROUP, GETPID, IOCTL, MUNMAP, POLL, READ, RT_SIGACTION, WRITE