An analysis of system call set extraction tools on configurable Linux binaries

Comparing the performance of various system call set extraction tools on various configurations of the busybox application

Bryan van der Mark[1]

Supervisor: Alexios Voulimeneas[1]

[1]**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: Bryan van der Mark
Final project course: CSE3000 Research Project
Thesis committee: Alexios Voulimeneas, Przemyslaw Pawelczak

Abstract

System calls are a primary way in which applications to communicate with the kernel. This is to allow them to perform sensitive tasks, however, an application will typically not require all of the system calls available to function properly. Despite this, the Linux kernel allows a program to perform any system call it wishes. This is bad for security, as it allows an attacker full access to the kernel after gaining code execution in a vulnerable program. By extracting a minimal set of system calls for a given program, we can sandbox it and only allow those system calls to be executed, greatly reducing the attack surface. In this paper, we analyze existing solutions that address system call set extraction. In particular, we will focus on applying these to configurable binaries. That is, binaries which can be compiled with a variety of different settings. For this paper, we have chosen to analyze `cat` as a minimal example, and `busybox` as the configurable application. We compile `busybox` in the following configurations, among variations: the default configuration, a configuration containing a minimal set of features and a configuration containing a maximal set of features. We analyze the performance of the tools *Binalyzer*, *Sysfilter* and *Confine* on these binaries. We see that *Confine* has significantly worse performance than both *Binalyzer* and *Sysfilter*. We also see that *Sysfilter* has better performance than *Binalyzer* when the complexity of the `busybox` binary is increased. We conclude that *Sysfilter* outperforms *Binalyzer* on binaries without debug symbols, while the opposite is true when performing analysis on binaries with debug symbols.

1 Introduction

System calls are a predominant way for programs to communicate with the operating system. For example, if a program needs to write to a file, it can do so via a system call. The operating system then checks if the program has sufficient permissions to do its requested action, and then performs the action if permitted. By default, a program is allowed to perform any system call, even if it does not require it to function properly. This is usually not a problem, however, the program could be compromised by an attacker via (for example) a buffer overflow attack. When this happens, the attacker can issue system calls on behalf of the compromised application and gain further privileges. To limit an attacker's capabilities after an application has been compromised, we can limit the set of system calls an application is allowed to execute to contain only the ones it needs for it to function. However, identifying this set manually is a labour intensive task, which is why research has been conducted to automate this process. These solutions include Chestnut[2], Sysfilter[3], Confine[4] and a temporal specialisation-based approach[5]. This research aims to answer the following research question: How do Chestnut's *Binalyzer*, *Sysfilter* and *Confine* compare when performing system call set extraction on various configurations of the `busybox`[9] application?

Section 2 gives additional background information. Section 3 contains a detailed explanation of previous work and states our main contributions. Then, section 4 gives more insight into the used methodology. An overview of the experimental setup and results is given in section 5. A reflection on the ethics and reproducibility of this research is given in section 6. Finally, a discussion is presented in section 7 and section 8 contains a conclusion.

2 Background

2.1 System call sandboxing

As software has gotten more complex, the attack surface of applications has gotten bigger. Memory safety violations are among the most prominent attack vectors, these allow an attacker to read or write data to memory that they are not supposed to have access to. This, in turn, allows modification of program behaviour. For example, an attacker could overwrite a function pointer stored on the stack, allowing diversion of program execution to any executable memory address. These memory safety violations accounted for 14% of all vulnerabilities reported to the Common Vulnerabilities and Exposures database[7]. Along with other techniques, memory safety violations provide attackers a powerful framework to gain arbitrary code execution. However, this execution is confined to the program which was vulnerable to memory safety related attacks. In order for an attacker to have an effect outside of this program, they will need to make changes to the system that is running the program. In a majority of cases, this will involve system calls. As of the time of writing, the default behaviour of Linux-based operating systems is to execute any system calls that a program invokes. This violates the principle of least privilege[8], as it is extremely unlikely for a program to require every available system call (335 at the time of writing) to function properly. This provides an opportunity for attackers, as they can execute arbitrary system calls after gaining code execution. To reduce the attack surface, system call sandboxing can be employed to restrict the set of system calls that are allowed to be executed to just the ones strictly required by the program. This can be done via *seccomp*[1], which ensures that any system call a program makes conforms to a filter set by the program before being executed. This paper will investigate existing solutions that promise to automatically detect this set of system calls.

2.2 Busybox

`busybox`[9] is an application that is often used in the context of embedded systems, where memory and storage space are limited. `busybox` is a single binary that includes compact (space efficient) versions of various other commonly used Linux binaries, called applets or functions. This means that it is an appealing option for firmware developers, as they can use `busybox` as an all-in-one solution when creating the file system for their firmware image (assuming their project runs on the Linux kernel). When they add the binary to the firmware, they get access to every applet that is compiled in the `busybox` binary. However, due to space constraints or

other factors, a developer may want to remove specific binaries from the `busybox` build. This can be done at compile time, as the `make config` and `make menuconfig` commands will give the user (firmware developer in this example) access to configuration options, where specific applets can be included or excluded from the compilation process.

3 Previous work and our contributions

Chestnut

Chestnut[2] is a utility that can be used to perform system call set extraction. This utility consists of three parts, *Sourcealyzer*, *Binalyzer* and *Finalyzer*. These parts can all be used independently from each other, or in combination. *Sourcealyzer* analyzes the source code of an application and uses this information to figure out an upper bound on the set of system calls the application may use. That is, it will always produce a super-set of the actual set (note that $\mathcal{A} = \mathcal{B} \implies \mathcal{A} \supset \mathcal{B}$). *Binalyzer* has the same goal, however it analyses a compiled binary instead of source code. This means that it has access to less information, but in return, it is able to work on any program (even closed-source applications). *Binalyzer* also over-approximates the actual system call set. *Finalyzer* is an optional dynamic component of *Chestnut*, which runs a specified binaries and keeps track of which system calls it performs. This can be useful in cases where static analysis is not possible, for example if the binary contains self-modifying code. *Finalyzer* is meant to be used to further refine sets found by *Sourcealyzer* and *Binalyzer*.

Confine

Confine[4] is a utility that is used to create system call sets specifically for Docker containers. It does this by first running the Docker container and analyzing which applications it starts up. Then, much like *Binalyzer*, it analyzes all of the binaries that were found during this dynamic analysis phase. Finally, it produces a set of system calls that could potentially be called in the container's lifetime.

Sysfilter

Sysfilter[3] has many similarities to *Binalyzer*. Although the implementations are different, for the purposes of this paper, these differences are not of concern. *Sysfilter* takes a binary program as input, and produces a set of system calls that the program could potentially use as output. Again, this set is an over approximation of the true set of system calls used by the application.

Temporal Specialisation

Temporal Specialisation is a technique proposed in *Temporal system call specialization for attack surface reduction*[5]. Instead of analysing an entire application at once, it proposes that many applications can be analysed in separate phases. For example, a web server will often have an initialization phase, where it reads configuration files and sets up child processes for various tasks. Then, once this is done, it may transition to a "serving phase", where it will read network requests and create responses to them. These two phases may have a different set of system calls which they require. By analysing these separately, one can further restrict the set of system calls available during a specific phase.

Our contribution

As explained in previous sections, this work deals with automated extraction of system call sets. The aim of this work is not to develop a new way to achieve this, but rather to analyze the previously mentioned existing solutions in the specific case of `cat`[6] and `busybox`[9].

4 Methodology

This paper will analyze three different methods of extracting system call sets on various configurations of configurable software. Namely, Chestnut[2], Sysfilter[3] and Confine[4]. More specifically, Chestnut's *Binalyzer* will be used.

These programs will be analyzed by having them analyze two programs, `cat` and `busybox`. The analysis of `cat` is used as a simple baseline, while the main part of the analysis will be done by examining the previously mentioned solutions on `busybox`. As explained previously, `busybox` can be compiled in various configurations. In this work, we will explore various of these configurations and investigating the effectiveness of the previously mentioned solutions on said configurations.

Readers may note the omission of *Sourcealyzer*, *Finalyzer* and the temporal specialisation-based approach. The omission of *Sourcealyzer* is due issues with compiling a patched version of `clang`, which is required to run *Sourcealyzer*. *Finalyzer* was not used as it required a binary to have *seccomp* filters installed before it can analyze the binary. However, the project's own *Binalyzer* was unable to install these filters. The temporal specialisation-based approach was not used due to the nature of `busybox`. `busybox` does not have clearly defined stages, as the user can choose to invoke a different applet every time they run `busybox`. Therefore, it does not make much sense to analyze `busybox` with this approach.

5 Experimental Setup and Results

5.1 Setup

Git repository

We have put all code and data discussed in this paper in a Git repository, which is available at https://github.com/bbvandermark/CSE3000SystemCallSandboxing

Versions of used tools

At the time of writing, the latest Git commits of the tools used have SHA hashes beginning with:

- *Chestnut* (*Binalyzer*): $839c39d$
- *Sysfilter*: $1469319b$
- *Confine*: $a75dad3$

Busybox configuration

In order to analyze `busybox`, various different configurations of the program will be compiled. In order to do this, the command `make menuconfig` is used to access the configuration options. The following configurations are created:

- **Default**: Running `make menuconfig` will generate a pre-populated `.config` file with the default settings for `busybox`. This will be regarded as the default configuration.

- **Minimal**: This configuration will consist of setting all boolean settings to `N`, meaning that everything that can be disabled, is disabled.
- **Maximal (excluding debugging options)**: This configuration will consist of setting all boolean settings to `Y`, except for those listed under the *Debugging Options* category. In addition, all applets and their options are enabled. This is with the exception of:
  - Support mounting NFS file systems on Linux < 2.6.23
  - Support RPC services
  - support PAM (Pluggable Authentication Modules)

  These are disabled due to compilation issues with these settings enabled.
- **Maximal (including debugging options)**: This configuration is the same as **Maximal (excluding debugging options)**, however, it has all debugging options set to `Y`, with the exception of:
  - Enable runtime sanitizers (ASAN/LSAN/USAN/ etc...)
  - Abort compilation on any warning

  In addition, no additional debugging library is provided to the compiler in this configuration.
- **All applets**: This configuration combines the previous configurations. It takes the settings from the **Default** configuration and the applet configuration from the **Maximal** configuration.
- **Cat**: This configuration is meant to simulate a scenario where `busybox` contains a single applet, namely `cat`. This configuration is the same as **Minimal**, however, the `cat` applet is enabled. In addition, the default settings for `cat` are kept. This means that the additional options that appear when enabling `cat` are kept as is. Coincidentally, this means that all settings are enabled, as the default behaviour is to enable all settings. We have chosen `cat` as we can then compare the found system call set to a set generated by analysing the version of `cat` bundled with Ubuntu 22.04.
- **Cat (minimal)**: This configuration is the same as **Cat**, however, all settings related to `cat` are disabled. This means that the following settings are disabled:
  - Enable -n and -b options
  - cat -v[etA]

All of the configuration files can be found in the Git repository associated to this paper, under the `busybox_configs` directory.

Busybox binaries

Compiling `Busybox` using the `make` command results in two binaries being created, `busybox` and `busybox_unstripped`. The main difference between these two binaries is that `busybox` is compiled without debug symbols, while `busybox_unstripped` has debug symbols included. For *Binalyzer* and *Sysfilter*, both of these binaries are analyzed. However, the unstripped binary is omitted from analysis in *Confine* due to the extensive time *Confine* requires for analysis.

Confine containers

Confine requires the user to have a Docker container that they wish to analyse. As this research deals with standalone binaries, this requires some pre-processing. This is done by creating a Docker container that runs the binary that we wish to analyze. This is done by copying the binary to a Docker image based on Ubuntu 22.04, followed by copying a script that manages running the binary. This is important, as *Confine* analyzes a container over a period of time. By having an associated script that runs the binary, we can ensure that the program is running through all phases of *Confine*'s analysis. In addition, the script also makes sure that all relevant parts of the program are executed. As this can vary depending on the programs usage, two variations of scripts are made, and each binary is analyzed with each variation. These variations are:

- **Minimal**: This variation does a minimal amount of work and only ensures that the binary is ran inside the container for at least 5 minutes. For the standalone and `busybox` versions of `cat`, this means using `echo` to direct a dummy input to the standard input of the program. This process is then repeated after sleeping for one second. For the remaining configurations of `busybox`, the script repeatedly calls the binary without arguments and sleeping for one second.
- **Regular use**: This variation attempts to emulate regular usage of the program. For the standalone and busybox versions of `cat`, this is understood as reading input from a pipe and reading from a file on disk. For the other `busybox` configurations, the busybox binary is invoked various times with different applets. The applets are chosen such that they capture, among others, file system navigation (`ls`), file interaction (`touch`, `rm`) and network interaction (`ifconfig`, `ping`).

As with the busybox configurations, the details of these scripts can be found in the Git repository associated to this paper, under the `confine_scripts` directory.

Running analysis tools

To ensure reproducible results, the analysis is done inside Docker containers wherever possible. This is done for *Binalyzer* and *Sysfilter*. *Confine*, however, is run without containerization. This is because *Confine* itself analyses Docker containers by running them. As running Docker containers inside of a Docker container is not trivial and could add additional variables to the analysis, it has been decided to run *Confine* on the host operating system. The Docker images use Ubuntu 20.04 as a base image, and install only the required dependencies to run the tools needed for analysis.

Analysing results

In order to parse results, the set of system call numbers generated for each binary by each tool will be gathered. As *Binalyzer* and *Sysfilter* both perform static analysis on a standalone binary, and they both over-approximate the sets, it is

interesting to take the intersection of these two sets. This results in another super-set of the actual system call set, but with potentially better bounds. *Confine* performs analysis on entire Docker containers, and therefore it may include system calls of background processes that were ran during the container's lifetime, which is why the set that it generated is excluded from the previously mentioned intersection. For *Confine*, multiple analyses are performed with different variations of runner scripts, as described previously. As *Confine* uses dynamic analysis (alongside static analysis), the resulting sets of these analyses are then combined together in a union. For completeness, a final set is constructed that contains the intersection of the previously constructed sets.

Any analysis that resulted in an empty set is marked as *Did Not Finish*. As every program analyzed in this research writes to the standard output, every set generated must include the `write` system call. Therefore, empty sets are invalid. When a set is invalid, we exclude it from any intersections that are calculated. For example, if a set $\mathcal{S}$ is invalid, then we regard $\mathcal{B} \cap \mathcal{S}$ to be equivalent to $\mathcal{B}$. This is because the intersection is taken as a way to further refine the set of system calls acquired, and since an invalid analysis gives no further information to the actual set of system calls, it can be disregarded.

5.2 Results

In the following tables, the following symbols are assigned to the various sets for brevity:

Set extracted by	Alias
Binalyzer	$\mathcal{B}$
Sysfilter	$\mathcal{S}$
Confine (Minimal variation)	$\mathcal{C}_\mathcal{M}$
Confine (Regular use variation)	$\mathcal{C}_\mathcal{R}$
$\mathcal{C}_\mathcal{M} \cup \mathcal{C}_\mathcal{R}$	$\mathcal{C}$

In table 1, one can see the size of the acquired system call sets for stripped binaries. From these results, we can see that the sets $\mathcal{C}_\mathcal{M}$, $\mathcal{C}_\mathcal{R}$ and $\mathcal{C}$ have the same size. As $\mathcal{C} = \mathcal{C}_\mathcal{M} \cup \mathcal{C}_\mathcal{R}$, we can conclude that the two variations on the runner script do not make a difference in the generated system call set. Furthermore, the sets generated by *Confine* are always much higher than the sets generated by *Binalyzer* and *Sysfilter*. *Sysfilter* produced the smallest set in every experiment, however, it did not complete the analysis of `busybox` in the maximal configuration. Despite *Binalyzer* producing a bigger set, it over-approximated a different set of system calls. This is evidenced by the fact that $\mathcal{B} \cap \mathcal{S}$ is smaller than both $\mathcal{B}$ and $\mathcal{S}$ in all analyses that completed. Another interesting observation is that, despite *Confine* generating a significantly larger set, it was still able to avoid over-approximating a few system calls that *Binalyzer* or *Sysfilter* were not able to avoid. This can be seen by noting that $\mathcal{B} \cap \mathcal{S} \cap \mathcal{C}$ is always one or two elements smaller than $\mathcal{B} \cap \mathcal{S}$, except for the the binaries that *Sysfilter* was not able to analyse. This is investigated later in the section. One can also see that the standalone version of `cat` was approximated to need 48 and 47 system calls by *Binalyzer* and *Sysfilter*, respectively. For both configurations of the `busybox` binary compiled with only *cat*, *Binalyzer* approximated a set with 10 more system calls in it. This is in contrast to *Sysfilter*, which generated a set that contained one less system call than the standalone binary. The pattern of *Binalyzer* approximating more than *Sysfilter* appears to be more extreme when more applets are added to the compilation. In the **Minimal** configuration, *Binalyzer* generates a set which contains 12 more elements than *Sysfilter*. However, in the **Default** configuration, this number is 106. In the configuration where all applets are compiled into `busybox`, *Binalyzer* approximates 110 more system calls.

In table 2, one can see the size of the acquired system call sets for unstripped binaries. Noteworthy is the sizes of the sets generated by *Binalyzer*, which did not change from the results obtained with stripped binaries, indicating that *Binalyzer* was not able to capitalize on the additional information provided by the debug symbols. However, this is not the case for *Sysfilter*, which performed worse on unstripped binaries than stripped binaries. Notably, in the **All applets** configuration, it went from 50 system calls found in the stripped binary, to 176 in the unstripped binary. *Sysfilter* performed worse than *Binalyzer* on every analysis, with the exception of both **Maximal** configurations. This is especially interesting considering that it was not able to complete the analysis of these binaries without debug symbols.

In table 3 we mention the specific system calls that have been removed by intersecting the set extracted by *Confine* with the intersections of the *Binalyzer* and *Sysfilter* sets. More specifically, only the resulting sets with a cardinality between 0 and 10 are shown. This gives more insight into the cases where *Confine* was able to identify system calls which were not used, which the other two approaches did not. From this table, we can see that the *sched_yield*, *gettimeofday* and *sendmmsg* system calls were able to be removed from the system call set thanks to *Confine*. From the appendix given in [2], we learn that *sendmmsg* is an equivalent of the *sendto* and *sendmsg* system calls. By inspecting $\mathcal{B} \cap \mathcal{S} \cap \mathcal{C}$, we indeed see that these equivalents exist in this set. This means that *Confine* only generated only the *sendmmsg* system call, while *Binalyzer* and *Sysfilter* included equivalent system calls in their sets. Interestingly, *sched_yield* and *gettimeofday* are not listed as having equivalent system calls, meaning that *Confine* was the only program that was able to identify that these system calls were not used.

The full results, along with the specific system calls in each set, are located in the Git repository associated to this paper. Specifically, a JSON representation of the results can be found in `results.json`. Along with this, the specific binaries used in this analysis are given in the `targets` directory.

6 Responsible Research

6.1 Ethics

This research deals primarily with the fields of cybersecurity. In particular, it deals with the topic of preventing the exploitation of vulnerabilities in applications. Cybersecurity is mainly concerned with protecting sensitive data. When a vulnerability is found, an application often needs to be updated in order to patch said vulnerability. However, it is not guaranteed that every user of the application will install this update. This means that a malicious party may read research

Binary	$n(\mathcal{B})$	$n(\mathcal{S})$	$n(\mathcal{C}_{\mathcal{M}})$	$n(\mathcal{C}_{\mathcal{R}})$	$n(\mathcal{C})$	$n(\mathcal{B} \cap \mathcal{S})$	$n(\mathcal{B} \cap \mathcal{S} \cap \mathcal{C})$
busybox_all_applets_default_settings	160	50	234	234	234	49	48
busybox_cat	58	46	148	148	148	33	32
busybox_cat_minimal	58	46	148	148	148	33	32
busybox_default	156	50	231	231	231	48	47
busybox_maximal_excluding_debugging_options	178	*DNF*	237	237	237	178	178
busybox_maximal_including_debugging_options	177	*DNF*	237	237	237	177	177
busybox_minimal	58	46	146	146	146	33	32
cat	48	47	147	147	147	31	29

Table 1: System call set sizes for stripped binaries

Binary	$n(\mathcal{B})$	$n(\mathcal{S})$	$n(\mathcal{B} \cap \mathcal{S})$
busybox_all_applets_default_settings_unstripped	160	176	154
busybox_cat_minimal_unstripped	58	62	46
busybox_cat_unstripped	58	62	46
busybox_default_unstripped	156	173	150
busybox_maximal_excluding_debugging_options_unstripped	178	47	47
busybox_maximal_including_debugging_options_unstripped	177	47	47
busybox_minimal_unstripped	58	62	46

Table 2: System call set sizes for unstripped binaries

Binary	$(\mathcal{B} \cap \mathcal{S}) \setminus (\mathcal{B} \cap \mathcal{S} \cap \mathcal{C})$
cat	sched_yield, gettimeofday
busybox_all_applets_default_settings	sendmmsg
busybox_minimal	sched_yield
busybox_default	sendmmsg
busybox_cat_minimal	sched_yield
busybox_cat	sched_yield

Table 3: System calls removed from $\mathcal{B} \cap \mathcal{S}$ by intersecting with $\mathcal{C}$ in cases where $0 < n((\mathcal{B} \cap \mathcal{S}) \setminus (\mathcal{B} \cap \mathcal{S} \cap \mathcal{C}) < 10$

about a vulnerability and decide to exploit any instance of this application which is still on a vulnerable version. This, in turn, means that publishing a paper of this kind can be ethically questionable. However, since this research focuses on preventing already existing vulnerabilities, this does not apply here. We do not see any reasonable ethical concerns in regards to the methodology used in this research. In addition, we also do not see any ethical implications in publishing this paper.

6.2 Reproducibility

In order to allow readers to reproduce the results gathered in this paper, we have uploaded all of the files used to gather these results in a Git repository. Notably, this Git repository contains Dockerfiles that automatically run analysis using *Binalyzer* and *Sysfilter*. The repository also contains a Dockerfile which attempts to run analysis with *Sourcealyzer*. As noted, we did not get this method to work, however, we have included the associated Dockerfile in case a reader wants to verify this behaviour themselves. By using Docker, we can ensure that analysis is run in exactly the same way on any machine. However, this is with the exception of the various `apt install` commands issued within these Dockerfiles. If the behaviour of the installed packages changes after the time of writing, then the results may not be the same anymore. However, this is extremely unlikely, as the installed packages have large user-bases (such as the `python3` package). This makes (major) changes in the API of these packages very unlikely. Even still, if this were to happen, a reader could change the Dockerfiles to install specific versions of the packages that coincide with the date on the front page of this paper. This should ensure that the results gathered by the reader will be exactly the same as the results shown in this paper. This comes with the exception of *Confine*, which, as discussed in section 5, was ran on the host operating system instead of in a container. As *Confine* runs a Docker container itself to perform its analysis, we believe that this is enough to make these experiments reproducible. We have included the script that creates the containers which are analysed by *Confine* in the repository, which provides additional reproducibility. Along with this, we mention the hashes of the latest Git commits of *Binalyzer*, *Sysfilter* and *Confine* in section 4. This helps ensure that a reader is able to run the exact same version of these tools as was used in this paper. Every action performed to perform the experiments has been made into a script, which a reader could run to ensure they are performing the exact same steps as have been performed to acquire the results shown in this paper. These scripts have also been added to the Git repository. Additionally, the compiled binaries are

also present in the git repository, meaning that variations in compiler output can be avoided by using the provided binaries instead of compiling new binaries with the given configuration files.

7 Discussion

In this paper, we have analyzed the performance of various system call set extraction tools on `cat` and `busybox` binaries. This has been done by running *Binalyzer*, *Sysfilter* and *Confine* on `cat` and various configurations of `busybox`. These configurations are chosen such that they capture a `busybox` binary with a minimal amount of applets, a maximal amount of applets and the default set of applets. On top of this, the configuration with the maximal amount of applets was split into three configurations, one with default settings, one with all settings enabled except for debugging options, and one with all settings enabled including debugging options. We have found that *Binalyzer* performs worse than *Sysfilter* on stripped binaries, while the opposite is true with unstripped binaries. This is with the exception of the configuration containing all applets and all settings, where *Sysfilter* was unable to complete analysis on a stripped binary, but outperformed *Binalyzer* on an unstripped binary. Furthermore, we have seen that *Confine* performs significantly worse than both *Binalyzer* and *Sysfilter* on every experiment that has been performed. Sets extracted by *Binalyzer* scale significantly worse than those extracted by *Sysfilter* with an increasing amount of applets compiled in `busybox`. However, *Binalyzer* produced consistent results across stripped and unstripped binaries. In addition, it was able to complete the analysis on all binaries.

The result that *Confine* performs worse than *Binalyzer* and *Sysfilter* was expected, since it performs analysis on every binary that was run during the lifetime of the Docker container that it analyzed. The resulting set is then the union of all the individual sets. As *Binalyzer* and *Sysfilter* perform analysis on a single binary, one can expect that the sets they extract are significantly smaller.

An interesting result is that *Sysfilter* performed worse than *Binalyzer* on unstripped binaries, while it performed better on stripped binaries. However, unexpectedly, on the stripped binaries that *Sysfilter* was not able to analyze, it completed the analysis on their unstripped counterparts with better performance than *Binalyzer*.

8 Conclusions and Future Work

In this paper, we have analyzed `cat` and various configurations of the `busybox` program. We have also answered the research question, how do Chestnut's *Binalyzer*, *Sysfilter* and *Confine* compare when performing system call set extraction on various configurations of the `busybox` application? Namely, *Confine* performs considerably worse than both *Binalyzer* and *Sysfilter* due to performing analysis on an entire Docker container, rather than a isolating analysis to a single binary. *Sysfilter* outperforms *Binalyzer* on stripped `busybox` binaries, but the opposite is true for unstripped binaries, with the notable exception of performing better on binaries that it was not able to analyzed when they were stripped. *Sysfilter* showed the best scaling when more applets were included in the compilation of `busybox`, meaning its extracted set of system calls grew the slowest when adding more functionality to `busybox`. *Binalyzer* did not produce a different set of system calls when invoked on a stripped binary compared to an unstripped binary.

For future work, one could analyze the performance of *Confine* when using a lightweight Docker image as a base for analyzing a binary. As well as this, the anomalous result of *Sysfilter* performing better than *Binalyzer* could be further investigated.

References

[1] seccomp - operate on secure computing state of the process. https://www.man7.org/linux/man-pages/man2/seccomp.2.html, May 2024.

[2] Claudio Canella, Mario Werner, Daniel Gruss, and Michael Schwarz. Automating seccomp filter generation for linux applications, 2020.

[3] Nicholas DeMarinis, Kent Williams-King, Di Jin, Rodrigo Fonseca, and Vasileios P. Kemerlis. sysfilter: Automated system call filtering for commodity software. In *23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020)*, pages 459–474, San Sebastian, October 2020. USENIX Association.

[4] Seyedhamed Ghavamnia, Tapti Palit, Azzedine Benameur, and Michalis 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)*, pages 443–458, San Sebastian, October 2020. USENIX Association.

[5] Seyedhamed Ghavamnia, Tapti Palit, Shachee Mishra, and Michalis Polychronakis. Temporal system call specialization for attack surface reduction. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 1749–1766. USENIX Association, August 2020.

[6] Torbjorn Granlund and Richard M. Stallman. cat - concatenate files and print on the standard output. https://www.man7.org/linux/man-pages/man1/cat.1.html, March 2024.

[7] Conor Pirry, Hector Marco-Gisbert, and Carolyn Begg. A review of memory errors exploitation in x86-64. *Computers*, 9(2), 2020.

[8] J.H. Saltzer and M.D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, 1975.

[9] Denys Vlasenko. Busybox: The swiss army knife of embedded linux. https://git.busybox.net/busybox/, 2024. Commit: 2d4a3d9e6c1493a9520b907e07a41aca90cdfd94.