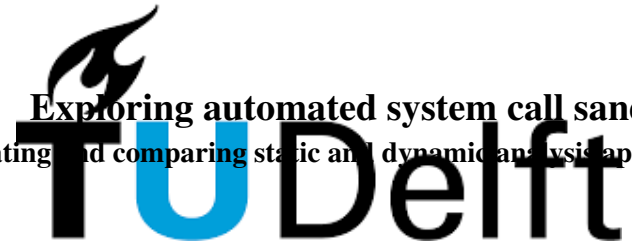

Exploring automated system call sandbox policy generation

Investigating and comparing static and dynamic analysis approaches to generate system call policies

Benjamin Selyem

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 21, 2024

Name of the student: Benjamin Selyem
Final project course: CSE3000 Research Project
Thesis committee: Alexios Voulimeneas , Przemysław Pawełczak

Abstract

System call sandboxing is the idea to restrict the set of system calls an application is able to invoke. This reduces the attack surface available to an attacker exploiting the binary, and adheres to the principle of least privilege, giving entities the minimum required permissions needed to perform their function.

The key goal is to automatically identify which system calls to block, since it is a complex, manual task requiring great insight into the program and its dependencies.

This paper investigates and compares various static analysis based solutions in this field, such as sysfilter [3], Confine [5] and Chestnut [2], by measuring their accuracy and analysis time. Furthermore a simple dynamic analysis based solution is created for the sake of comparison with the previously mentioned tools. The tools are evaluated on a small set of commonly used Linux applications, such as *ls*, *sqlite* and *Redis*, and the results are reported.

In addition to the aforementioned tools, temporal specialization [6], a solution which considers multiple execution phases is also investigated and compared with a purely dynamic analysis solution having support for multiple phases of execution.

The research shows that although dynamic analysis underapproximates the set of required system calls it can adapt to a custom usage profile. Additionally, although static analysis is slower and more complex, the research explores areas of improvement such as precomputing or multiple threads.

1 Introduction

Operating systems allow applications to perform privileged operations through the use of system calls. The idea of system call sandboxing is to restrict the set of available system calls for each application to the minimum required set in order to adhere to the principle of least privilege [8] and reduce the potential attack surface in case the application has a vulnerability.

Existing work focuses on generating the required set of system calls automatically, by statically analyzing the call graph and control flow of the binary [3, 5] or performing compile-time analysis [2]. Another approach that is taken is to dynamically analyze the application [9] and obtain the set of used system calls that way. A similar and related topic in the area of attack surface reduction using dynamic analysis is removing unused parts of applications (debloating) [4, 7]. Overarching these approaches, some existing work tries to establish different phases of execution to generate a finer grained set of system calls to block per phase [6].

While static analysis is able to generate accurate results, with slight overapproximations [9], it is often costly to perform [2] and techniques can quickly become complicated as the size of the application scales. On the other hand dynamic analysis techniques tend to result in less accuracy and underapproximation [3], while being simpler to scale to larger applications. This raises a question about what exactly the trade-off is between these 2 different approaches in terms of runtime and accuracy, and when should one be preferred over the other.

The aim of this work is to explore the difference between these 2 approaches and to report the runtime and accuracy of various different implementations that rely mostly on static analysis. More specifically this work poses the following 3 research questions:

- **RQ1:** How can we use dynamic analysis to gather a list of system calls for selected applications, so that they can be used as a basis for comparing other static analysis based solutions?
- **RQ2:** How do different static analysis based solutions compare in terms of accuracy and runtime when considering applications that have a single execution phase?
- **RQ3:** How do different static analysis based solutions compare in terms of accuracy and runtime when considering applications that have multiple execution phases?

The main contributions of this research are:

- A simple dynamic analysis implementation that generates a list of required system calls for a given application.
- Analysis in terms of runtime and accuracy of some existing static analysis based solutions.
- To conclude, a comparison between the dynamic and static analysis approach.

Section 2 gives more background information on the topic, after which, in section 3, the experiment setup is described in detail. Section 4 provides the runtime and accuracy results, and afterwards section 5 reflects on the ethical aspects and reproducibility of the research. Section 6 reasons about the results of the research and gives explanations for the observations. To wrap up section 7 finishes with concluding thoughts and possible future work.

2 Background and Related Work

System calls are the interface the kernel provides to applications to perform privileged operations and interact with hardware devices. This mechanism allows for applications to exist in isolation while running simultaneously, and protects hardware components from malicious behavior to a certain extent. Normally an application is allowed to invoke any system call and so it has the privileges of the user who started the application.

The principle of least privilege demands that every unit of execution is given the minimum amount of privileges that still allows it to fully function [8]. This is ideal since in the case of a user error, an application bug or a vulnerability the potential damage is minimized without decreasing the usability of the given application. The regular flow of system calls thus does not adhere to this principle, since not every application needs access to every system call.

System call sandboxing gives the ability to disallow invoking certain system calls for a given application that are not going to be required. This ability allows applications to better adhere to the principle of least privilege, and thus can greatly reduce the attack surface available to potential attackers. However the configuration is a tedious, manual process that needs to be done on a per application basis and requires highly detailed understanding of the given program and its dependencies. This is a possible explanation for the low adoption rate of this technology in open source projects [1].

Automating the system call policy generation process has been the focus of recent work in order to make system call sandboxing easier to use and to potentially increase the adoption of this technique. A variety of solutions have been proposed that can be split into 2 main categories, static and dynamic analysis. Both approaches aim to automatically discover the set of used system calls by analyzing the given program and its dependencies.

Static analysis solutions [2, 3, 5] inspect the source code or the binary of the given program without executing it, and identify the list of used system calls that way. This approach tends to slightly overestimate the set of required system calls [9], while still leading to a considerable attack surface reduction. The main issue of static analysis is that it does not scale well as program size increases, resulting in longer analysis time and more complicated setup as the amount of dependencies grows.

Dynamic analysis solutions [9] trace the given program while it is being executed and record the invoked system calls that way. This approach tends to underestimate the set of required system calls, unless the given program is extensively exercised during analysis to explore as many program states as possible that may invoke system calls that have not been recorded previously. Thus the main issue of dynamic analysis is the reliance on a test suite which explores adequately many program states, however in return dynamic analysis is simpler to scale to larger applications.

Temporal specialization [6] uses another novel technique that does not strictly fall into any of the 2 main categories. The core idea is to consider that applications tend to have multiple phases of execution that require their own set of system calls and achieve a more fine-grained policy. The model works well especially in the case of server applications, which tend to have a first phase that sets up the server and then a serving phase, responsible for handling clients.

With many solutions tackling the challenge in various different ways the need for a study to compare them is motivated, and this is the gap that the research is aiming to address. In addition to comparing various proposed implementations and analyzing the underlying techniques, this study offers the chance to see how these tools fare 1-2 years after their release. Furthermore, a small program analysis tool based on dynamic analysis is provided for the sake of comparison with the seemingly more popular static analysis based techniques.

3 Experiment setup

During the experiment 5 analysis tools were configured and 3 separate programs were inspected. The environment for the experiment was a Docker container based on the *ubuntu:18.04* image. The kernel is that of the host machine, which can be identified by: *6.9.1-arch1-1*. The test computer had 16GB of RAM and an *AMD Ryzen 7 4800H* 8 core CPU. The analysis timing information was collected by the *time* command, except for one case, which is specifically stated below. More information about what steps of the process got timed for each of the tools can also be found below.

The 3 programs under analysis are listed below:

- **ls** - was chosen since it is a simple, frequently used program. It is part of the bigger *coreutils* package, of which version 8.28 was used.
- **sqlite** - was chosen since it is a commonly used application database. It is more complicated than *ls* and therefore invokes more system calls. Version 3.22 was used.
- **redis-server** - was chosen to test a larger server application. Since it is a server application it also deals with sockets, unlike the previous 2 applications. Version 4.0.9 was used with the default configuration.

The *Dockerfile* for setting up the analysis tools, and patches made to the projects (where applicable) can be all found on the project's source code repository (https://github.com/sbencoding/syscall-sandboxing-research).

3.1 Dynamic analysis solution

A simple tracer was designed by the author to test out and report the results of a purely dynamic analysis based solution. The tracer is based on the *ptrace* system call and requires a Linux kernel version of at least 5.3 to function, since it relies on the *PTRACE_GET_SYSCALL_INFO* operation.

Since dynamic analysis requires the program under test to be exercised in order to explore as many program states as possible, a simple framework and scripts to provide input to and gather the results from the analysed programs is also included. In the case of *ls* and sqlite the input cases are manually determined, while in the case of redis the *redis-benchmark* utility included with the redis-server package was used, with the *-n 10* option.

The dynamic tracer also supports the multi phase execution model in order to be compared with temporal specialization [6]. The tool allows specifying the exact instruction at which a phase transition should be considered, by passing the offset of the instruction from the start of the binary as a command line argument. The tracer supports up to 10 phase transition points, however this is easy to extend if needed.

In the case of *ls* the call instruction to *clear_files* found within the *main* function was chosen as the phase transition point, since most initialization is done before this point, and the work of listing files and folders is done afterwards. For sqlite, the first instruction of the *shell_exec* function was chosen, as this is the part where it starts dealing with the query after loading the database and handling other command line options. Finally the first instruction of the *aeMain* function is

the transition point chosen for redis, which is the same transition point chosen by temporal specialization.

The timing information includes running the simple framework, executing all the defined analysis cases, gathering and printing all results.

The source code for the tracer, as well as the test framework with the test cases used is available in the project's source code repository.

3.2 sysfilter

The sysfilter [3] analysis program, which relies mainly on static analysis, was setup inside the docker container using the instructions given by the authors. After the project was built, the programs under analysis were ran through the compiled analysis tool, and the results were gathered.

The timing information includes only the runtime of the analysis program, which also prints the results.

3.3 Confine

The Confine [5] analysis program, which relies mainly on static analysis, was ran on the host machine, outside of a docker container, since the tool itself is going to spawn containers, where the programs under test are ran. 3 docker containers were analysed, one by one, each based on the *ubuntu:18.04* image and running one of the programs under analysis. Furthermore to make an adequate comparison between the analysis tools the author has modified the source code to include include the syscalls from the program that was being tested and its dependencies, as opposed to all programs the container runs. It also needs to be mentioned that *sysdig*, a monitoring tool Confine relies on, crashed on the host system when launching a docker container therefore *execsnoop* was used as a *monitoringtool*.

In addition to the instructions of the authors on running the tool, the above mentioned *monitoringtool* options was used, as well as *-d* to get debug output. The input JSON file that defines which containers to test was a simple file containing only the container which ran a given program under analysis.

The timing information collected includes the time to analyze extracted binaries and dependencies and to generate a list of used system calls. This specifically excludes the time to run the container and extract the binaries and dependencies from it, in order to make a better comparison to the other tools. The time was measured using the *time.time_ns()* python function.

3.4 Chestnut

The Chestnut [2] project, which relies mainly on static analysis, has 2 main components performing analysis. *Sourcealyzer* extracts the used system calls through a compiler pass by patching the LLVM compiler, while the *Binalyzer* scripts operate on already compiled binaries. The project authors provide *Binalyzer* for cases when using *Sourcealyzer* is not feasible, for example because of incompatibility with the LLVM compiler or lack of available source code [2].

Optionally a second phase of analysis called the *Finalyzer* can be performed either after the aforementioned 2 analysis tools or as a standalone analysis process. It is based on dynamic tracing, and its goal is to unblock system calls which were not detected by the static analysis tools, such as in the case when the program under analysis dynamically loads another process as a child. It is intended to be used during development in a safe environment, and requires the tester to manually decide whether a system call should be allowed that was not detected by static analysis.

Finalyzer is not evaluated in this research, since it requires manual intervention to decide if a system call outside the list of allowed system calls can be made or not. Furthermore, to which extent the applications work after blocking the detected system calls is outside the scope of this research and is indicated as future work.

The project was built and ran inside the docker container. The *Binalyzer* part got patched by Petr Khartskhaev, to solve an issue where the *syscalls.py* script did not work. The experiment evaluates both *syscalls.py* and *cfg.py* on the binaries included with the Ubuntu distribution, as well as the statically built binaries, which are generated by the patched clang compiler explained below.

Furthermore the author has made some modifications to the *Sourcealyzer* part of the project. First, the LLVM download and patch script was modified to apply the patch without user interaction and to also build clang and lld. Next, the number of concurrent build jobs were also reduced to 10, since the build process ran into out of memory issues on the computer performing the build.

Afterwards *musl* was setup (commit hash 007997299248b8682dcbb73595c53dfe86071c83), since all dependencies of programs under test need to be rebuilt using a new compiler flag, and building *glibc* using the patched clang compiler did not work due to some missing GNU extensions. Then the *musl-clang* wrapper was modified to use the patched linker, instead of the system default, this is required for the intended program analysis to be done. Furthermore a copy of the wrapper was made that supports static building, since the default one has command line options which imply dynamic linking.

Next libseccomp was rebuilt (commit hash 9da5d174e3ef219baab020a79c789f2075ace45c) against *musl*, because a chestnut object, that needs to be built into the final analyzed program, depends on it, however the final analyzed program is not built against *glibc* like the installed libseccomp-dev package is. In the following step the aforementioned chestnut object was rebuilt against the new libseccomp, and that is the final step to setup the tool.

From here the programs under analysis need to be rebuilt, with the *-static -lseccomp -fautosandbox* compiler flags, for the tool to perform analysis and include the used system calls in the resulting binaries.

Timing information for *Sourcalyzer* includes the time to clean build the programs under analysis, using 16 concurrent *make* jobs where applicable. For *Binalyzer* it includes running the given analysis script from start to completion.

3.5 Temporal Specialization

The temporal specialization [6] project, which relies mainly on static analysis, was setup and ran inside the docker container. To setup the container the docker file provided by the project's repository was used as a basis, but with the specific

ubuntu:18.04 base image. This docker file was modified to download and build the *gold linker*, which is required by the project to generate bitcode, needed for the analysis of binaries.

Furthermore *createSyscallState.py* was modified to properly find the generated CFG files, without the use of the optional *–libdebloating* parameter. Additionally *run.sh* was modified to only perform the system calls analysis, and do so without the *–libdebloating* flag. The reasoning behind the changes is to skip the analysis using *libdebloating*, which the original project does for the sake of comparison, but it is a tool not in the scope of this research. Similarly, statistics about exploit mitigation are disabled, since this research does not consider such a metric.

The pre-generated bitcode for redis was using a different version than this research, therefore a new bitcode file was generated with the same settings, for the desired version. Bitcode for sqlite was generated, and *shell_exec* was chosen to be the entry function of the worker phase. *ls* did not get tested using this tool, due to difficulties of specifying a proper worker phase entry point for this application.

For this project 2 kinds of timings are collected, both using the *time* shell command. First, the time required to generate the bitcode, which means a full clean build of the tested programs, using 16 concurrent *make* jobs where applicable. Second, the time required to perform the analysis on the generated bitcode, which includes CFG generations and extracting the used system calls.

4 Results

This section details the results of the experiment, which was set up as described by the previous section. The purpose of this section is to state how each tool performed and reasoning about the results is deferred to the discussion section. The reported metrics are the analysis time and the number of blocked system calls.

4.1 Dynamic analysis solution

As described in the experiment setup, the dynamic tracer supports both single phase and multi phase model of execution. The following subsections describe the results for each of these 2 models.

Single phase model

Table 1: Blocked system call count and analysis runtime for the dynamic analysis solution implemented for this research, using the single phase execution model

Program	Blocked system call count	Runtime
ls	345	104ms
sqlite	341	344ms
redis	346	652ms

The dynamic analysis based tracer, written for this research is quick, however it blocks many system calls. The analysis for *ls* took around 100ms for executing 8 cases and gathering all results. Out of the 373 system calls available on the host system it blocks 345, as depicted in Table 1, blocking system calls frequently used in attacks such as *mprotect* and *execve*.

For sqlite and redis the performance is similar, in the 300-400 ms range, blocking 341 and 346 system calls respectively. Just like in the case for *ls*, system calls frequently used when exploiting binaries have been blocked, however due to the nature of the applications the *read* and *write* system calls still allow for some data oriented attacks.

Multi phase model

Table 2: Blocked system call count and analysis runtime for the dynamic analysis solution implemented for this research, using the multi phase execution model. The second column depcits the number of blocked system calls in the first phase, followed by the second phase.

Program	Blocked syscalls per phase	Runtime
ls	345 / 354	108ms
sqlite	341 / 357	361ms
redis	346 / 357	725ms

It is important to note, that all system calls made by phase 1 are also included in the allowed system calls of phase 0. This means that although phase 0 only uses 17 unique system calls for *ls*, it also consideres system calls that are used exclusively by phase 1, hence the lower number of blocked system calls. This is done because of how *seccomp* operates, only allowing a stricter policy, after the first one is installed.

In the case of *ls*, considering a second execution phase allows for blocking 9 more system calls after phse 0 is done, as depicted in Table 2, and finishes within 108ms, similar to the speed of the single phase model.

For sqlite, the increase in blocked system calls is even higher, with 16 additional calls blocked in the second phase and analysis finishing within 361ms. In the case of redis 11 additional system calls are blocked and anlysis is done in 725ms.

4.2 sysfilter

Table 3: Blocked system call count and analysis runtime for sysfilter.

Program	Blocked system call count	Runtime
ls	293	12.4s
sqlite	269	17.35s
redis	275	13.06s

The analysis for *ls* took around 12 seconds, identifying 293 system calls to be blocked, as depicted in Table 3. Analysis of sqlite took slightly more time with 17 seconds and managed to block 269 system calls. For redis the analysis finished a bit faster with 13 seconds, blocking 275 system calls.

4.3 Confine

ls was analysed in around 1.3 seconds, identifying 201 system calls to be blocked, as shown in Table 4. Analysis for sqlite took slightly longer, with 2.2 seconds, resulting in 188 system

Table 4: Blocked system call count and analysis runtime for Confine.

Program	Blocked system call count	Runtime
ls	201	1.38s
sqlite	188	2.21s
redis	189	1.54s

calls to be blocked. With redis the analysis took a bit less time with 1.5 seconds, and 189 system calls were blocked.

4.4 Chestnut

The following section details the results for the Chestnut project. Since it has multiple components and the usage of statically linked binaries compared to dynamically linked binaries makes a significant difference in some results, more subsections are suitable for this project.

Sourcealyzer

Table 5: Blocked system call count and analysis runtime for chestnut (Sourcealyzer).

Program	Blocked system call count	Runtime
ls	318	4.53s
sqlite	296	1.67s
redis	286	4.54s

Sourcealyzer produces 318 blocked system calls in around 4.5 seconds, when compiling *ls*, as Table 5 shows. For sqlite the analysis time drops to around 1.7 seconds, and 296 system calls are blocked as a result. Redis took around 4.5 seconds to analyze, after which 286 system calls got blocked.

Binalyzer - syscalls

Table 6: Blocked system call count and analysis runtime for chestnut (Binalyzer - syscalls).

Program	Blocked system call count	Runtime
ls (dynamic)	93	6.37s
ls (static)	319	951ms
sqlite (dynamic)	99	10.768s
sqlite (static)	298	3.01s
redis (dynamic)	99	9.49s
redis (static)	292	3.05s

In this subsection binaries were analyzed using the *syscalls.py* script. For the dynamically linked *ls* the analysis took 6.7 seconds and blocked only 93 system calls, in comparison the same analysis on the statically linked binary took 900 ms, but managed to block 319 system calls, as Table 6 depicts.

A similar trend between the static and dynamic linkage can be observed with the other 2 programs, with the dynamic sqlite binary taking 10 seconds to analyze and blocking 99 system calls and the static version taking only 3 seconds, and blocking 298 system calls.

Redis took 9 seconds to analyze using the dynamic version, but only 3 seconds when using the static version. 99 and 292 system calls got blocked respectively.

Binalyzer - cfg

Table 7: Blocked system call count and analysis runtime for chestnut (Binalyzer - cfg).

Program	Blocked system call count	Runtime
ls (dynamic)	-	5.52s
ls (static)	318	11.88s
sqlite (dynamic)	-	1.43s
sqlite (static)	297	40.22s
redis (dynamic)	-	53.3s
redis (static)	292	60.52s

In this subsection binaries were analyzed using the *cfg.py* script. For all 3 programs the analysis of the dynamically linked libraries did not yield any blocked system calls, with *ls* taking 5 seconds to analyze and redis taking 53 seconds. Additionally for sqlite the analysis failed in 1.4 seconds, due to *angr* not being able to construct the CFG.

For the statically linked binaries the performance is better. *ls* took 11.8 seconds to analyze and 318 system calls got blocked, as shown in Table 7. For sqlite the analysis took 40.2 seconds and blocked 297 system calls. In the case of redis the analysis finished in 1 minute and blocked 292 system calls.

4.5 Temporal specialization

Table 8: Blocked system call count and analysis runtime for temporal specialization. The second column depcits the number of blocked system calls in the first phase, followed by the second phase. The thid column contains the runtime for the bitcode generation, followed by the runtime for the full analysis

.

Program	Blocked syscalls per phase	Runtime
sqlite	305 / 311	1.91s / 2m 18.37s
redis	287 / 295	5.97s / 2m 12.78s

As explained in the experiment setup, *ls* was not analyzed using this tool, as there was no good candidate function to mark as the worker phase.

In the case of sqlite, adding a second phase allows to block 6 additional system calls, as show in Table 8. Bitcode generation finished under a reasonable 1.91 seconds, however the analysis part took over 2 minutes.

For redis, 8 additional system calls can be blocked in the second phase. Bitcode generation took around 6 seconds, while the analysis lasted over 2 minutes, like in the case of sqlite.

5 Responsible Research

Due to the topic of this research it is difficult to write about ethical aspects in the classical sense. No user data has been collected, no human subjects were interviewed during the research and the results likely do not directly impact any end

users. Therefore it is more beneficial to consider the indirect impact on users and consider the ethical aspects through these lenses.

The technologies discussed in the paper help with protecting software and thus protecting its users, but in some cases it can prevent software from functioning correctly. Thus the key aspects this section aims to cover are:

- Ensure that the security guarantees of the presented technologies is clearly communicated, so to not create false expectations.
- Ensure that potential limitations of the techniques are properly discussed and communicate when and how the protection techniques may prevent a given application from performing its functions properly.
- Ensure that the results presented are acquired through a reproducible research experiment.

To make sure the paper does not set false expectations, first the underlying concepts of all the presented protection mechanisms are detailed in the Background and Related Work section. The section gives the reader a better understanding of how the techniques protect against potential attacks and how far these protections extend, and what types of vulnerabilities and attacks are not covered by them. Furthermore, for each approach the advantages and disadvantages are discussed, so the compromise between security and functionality is clear and this is corroborated by the results of the experiment.

Naturally there is a chance that the tools discussed in the paper lead to certain functions of applications being restricted, which may even lead to the whole application becoming unusable. To this end the paper explains the drawbacks of the various approaches to solving the problem, underlining that in case dynamic analysis tools grossly underestimate the set of required system calls, the application can stop functioning. Furthermore it is important to highlight that most of these works are in their research phase and are not recommended to be deployed in production systems just yet. It is important that readers of this paper understand this risk, otherwise businesses and end users can be negatively impacted by the outage of a given service.

Therefore it is important, due to the reasons outlined above, that readers of this paper are aware of how the experiment was conducted in order to have a precise understanding of how these technologies perform and what are their limitations. To this end the experiment setup section goes into great detail about how all the programs under test were setup, how the analysis tools were setup, what the test environment looked like. This is ensured by providing precise version information about each library and application that was used, and version information about the operating system, important packages and Linux kernel. The paper also describes what kind of information was measured during the experiment, and how this information was measured. In the end, following all steps precisely should lead to the same experimental results and should increase the confidence of the readers in the presented results.

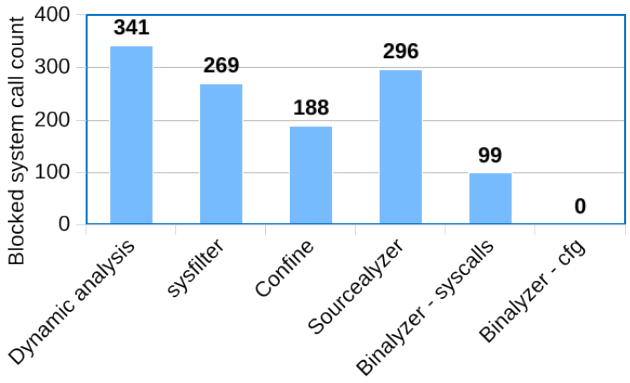

Figure 1: Blocked system call count for all investigated analysis tools for sqlite.

6 Discussion

This section explores the results stated in the Results section and attempts to find explanations or new connections between topics. The following subsections first investigate the set of blocked system calls among the different tools, then look at the runtime of the analysis. Next the setup and usability of the tools is investigated, and lastly the results of the multi phase execution model are discussed.

6.1 System call sets

When looking at the results it is evident that the dynamic analysis solution developed for this research blocks the largest amount of system calls, as shown on Figure 1. However it is important to mention that this result is only as strong as the analysis cases that are exercising the program under test, and therefore it is possible that some execution path is missed. This leads to underestimating the set of required system calls, and if these system calls were truly blocked it might stop the applications from working under certain conditions that were not explored during testing. It is difficult to come up with a comprehensive analysis suite, since the behavior depends not only on the inputs to the program but also on the state of the system as a whole.

In comparison static analysis tools block less system calls, and they potentially overestimate the set of required system calls, depending on how much information about the program is available and what techniques are used. However the Sourcealyzer part of Chestnut achieves a close number of blocked system calls to the dynamic analysis solution, demonstrating that the degree overestimation is minimal in some cases. On the other end of the spectrum, Binalyzer with *syscalls.py* on dynamically linked binaries blocks significantly less calls than any other solution.

Perhaps another positive note about dynamic analysis is that it is possible for the blocked set of system calls to be learned from actual usage of the program. If only a part of the built-in functionality of a program is used in a given system, it can be safer to block system calls related to the unused functions. This concept is not captured by purely static analysis based tools, which first slightly overapproximate the re-

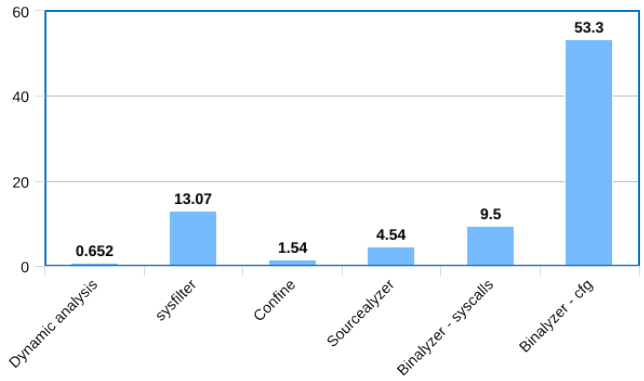

Figure 2: Analysis times for all investigated analysis tools for redis.

quired set of system calls and then offer no solution to reduce the size of this set.

6.2 Analysis time

From the aspect of analysis time it can be seen that the dynamic analysis solution takes the least amount of time, as shown in Figure 2, therefore it leaves room for more analysis cases to be added, until the analysis time approaches that of the static analysis tools.

The third fastest solution is confine with 1-2 seconds spent per program on average. This is because confine contains a pre-computed call graph for certain libc versions, and thus skips the steps where most analysis tools spend their time, analyzing what system calls the libc functions make that are used by the given program. Therefore a promising technique to speed up analysis could be precomputing results for popular libraries and dependencies.

The next fastest solution is perhaps surprisingly the Sourcealyzer part of Chestnut, with 1-4 seconds of analysis time per program. Although this may be unexpected a first, a significant speedup is achieved by utilizing multiple cores. Since the Sourcalyzer is implemented as an LLVM pass it has the same parallelization options as other compiler tasks, and thus allowing 16 concurrent jobs while testing greatly favors solutions which take advantage of multiple cores. In comparison, all of the other solutions relied on single core execution, therefore another opportunity to increase analysis speeds is to utilize multiple cores.

Perhaps the slowest solution is the Binalyzer part of Chestnut, specifically using the *cfg.py* script. This demonstrates the CFG construction is a time consuming task, and also nicely demonstrates that as program size scales up from *ls* to redis, execution time also increases from 11 seconds to 1 minute. This further highlights the benefit that static analysis solutions aim to find faster, but potentially less accurate ways to find out which functions are called, and which system calls are connected to those functions.

6.3 Setup & Usability

Although the above detailed metrics are of most importance, the adoption of these technologies depends on how easy they

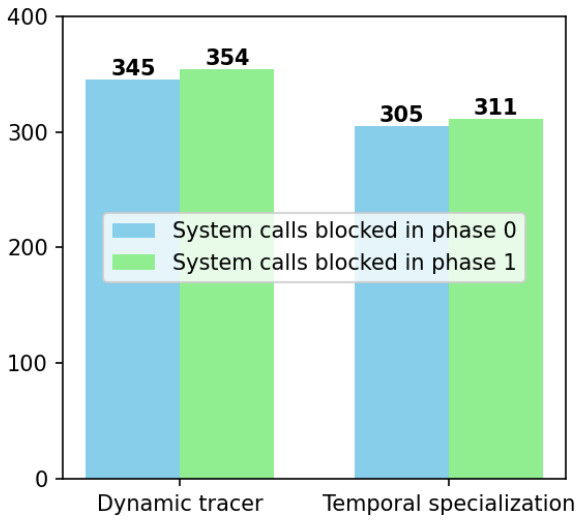

Figure 3: Blocked system call count per phase for the dynamic tracer and temporal specialization

are to setup and use. Additionally it is an important aspect to consider when choosing a tool. For example a longer analysis time might be acceptable if the setup and use is much easier compared to the other tools.

The dynamic analysis solution is relatively simple, it is easy to compile and run. The analysis cases are easily expandable, however it is tedious to come up with an analysis suite that adequately covers the program space. Additionally, the risk of mistakenly blocking a system call might outweigh the advantages of the easier setup and use.

In the case of sysfilter the setup is relatively easy and the analysis software worked well on the tested binaries. A potential problem here is the distribution or package versions used, as running the program on ubuntu 16.04 leads to the analysis tool crashing during analysis.

For Confine the setup is similarly easy and the analysis software worked well, and although on the test system there were problems with *sysdig*, luckily other monitoring tool options were also supported. Here a potential problem is with containers that are not long running, and exit within 60 seconds of starting. The analysis tool is designed for long running containers and attempts to analyze a simple container that exits almost immediately after starting have been unsuccessful.

In the case of chestnut, the setup is pretty cumbersome, as no guide or detailed instructions are available on how to setup a system to work with chestnut. The setup is more time consuming compared to other tools, due to the large amount of compilation work. Additionally the implementation is restrictive in the compiler used, its version, and the need for static compilation.

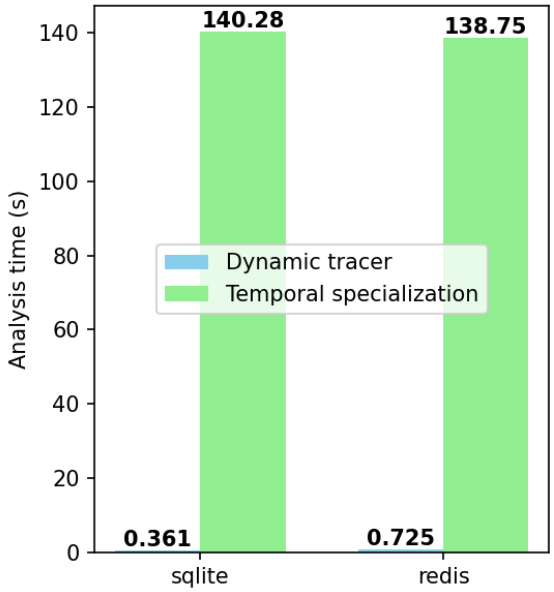

Figure 4: Total analisys time for the dynamic tracer and temporal specialization on sqlite and redis

6.4 Multi phase execution model

Overarching the static and dynamic analysis approaches, the multi phase execution model shows further improvements in the amount of blocked system calls compared to the single phase execution model. Although the key motivation behind this model is server applications, the results show that this technique can be applied to non server applications as well, such as sqlite. In general this approach extends nicely to applications, which first have a section initializing the program, handling user input and reading configuration files, and another section which performs the wanted action. However it is important to note that using this technique raises the question about what the optimal transition point is, and how to determine it automatically.

In terms of the difference between the static and dynamic analysis approach, the conclusions are similar to that of the single phase execution model. Dynamic analysis tends to underestimate the set of required system calls, while static analysis overestimates this set a bit, as shown in Figure 3. However dynamic analysis still has the advantage of being magnitudes faster in this case when compared to the temporal specialization tool, as can be seen in Figure 4.

An interesting approach of the dynamic tracer is the ability to define phase transition at specific instructions. Although doing so is rather cumbersome, and subject to change across different version of the binary, it offers greater accuracy when selecting a transition point. Furthermore, as in the case of *ls*, a transition point may not always be encapsulated in a separate function, therefore offering this level of specificity can be helpful in cases where the multi phase model is not intended to be used.

Another interesting idea not explored in this research is the ability to provide multiple phase transition points, which the dynamic tracer supports. For example this may allow splitting applications into setup, serving and cleanup phases, or even more depending on the application at hand. A potential problem is that seccomp would require earlier phases to allow all system calls of all further phases, since once a seccomp profile is set, it can only be further restricted.

7 Conclusions and Future Work

In the end there is no obvious winner between static and dynamic analysis. As the research shows, both approaches have their own advantages and drawbacks in terms of accuracy, runtime and ease of use. While static analysis slightly overapproximates, techniques using more information about the program reduce this problem, and analysis times can also be reduced by compromising a bit on accuracy. And although dynamic analysis usually underapproximates, sometimes this effect may be beneficial and should not be overlooked.

To summarize, this research explored the area of system call sandboxing, comparing various implementations in the area. To compare the approaches, an experiment was set up and carried out, after which the results such as accuracy and runtime were collected. To finalize the research reflects upon the results aiming to answer the research questions it set out to investigate.

A dynamic analysis based implementation was proposed in order to evaluate how well a purely dynamic analysis based solution would perform, since this approach is less popular in the field. The results show that such an approach is faster compared to other solutions, and blocks more system calls, although at the cost of accuracy. A key problem of the approach is the underestimation of the set of required system calls, which relies on there being a sufficient amount of test cases that explore many possible program states. A strong positive of this approach is the ability to create custom system call filters based on usage pattern, thus achieving more security.

Static analysis based tools, namely sysfilter, Confine and chestnut, were set up and tested. The results show that each implementation blocks a different number of system calls and does so with varying analysis time. The research discusses the potentials to speed up analysis time and looks at the advantages and drawbacks of these tools, also taking into account the ease of setup and usability. A key problem of these tools is the difficulty to setup and maintain them, when switching distributions or upgrading a compiler or library. A strong positive of these tools is that they are able to generate a relatively precise set of system calls to filter, independent of the given application.

Additionally the multi phase model of execution was investigated in both the static (temporal specialization [6]) and dynamic analysis context (the dynamic tracer with multi phase support). The research shows that static and dynamic analysis have the same drawbacks and advantages as in the case for a single phase execution model. Furthermore, based on the results the multi phase execution model is more effective com-

pared to the single phase counterpart, and can be deployed even on applications, which do not follow the general life cycle of server programs. A key problem of this approach is the need to identify the optimal transition points. A strong positive is the ability to have an even more restrictive filter for certain parts of the execution.

It is possible to perform more research in this area by improving techniques based on this research. Based on the results, approaches which precompute and cache call graphs and other results for commonly used libraries tend to spend less time analysing, such as in the case of Confine or chestnut. Furthermore, the research shows that incorporating all CPU cores to do the analysis can be beneficial in reducing analysis times.

To improve multi phase execution the research highlights 2 core ideas. First, defining transition points on the instruction level, which although cumbersome, offers the possibility to split large functions into phases, allowing the technique to be applied to programs such as *ls*, which would not be considered multiple phase. Second, the ability to define multiple transition points as opposed to one, although not investigated in this research, should allow for greater flexibility and even more precise filters.

Additionally, this research did not focus on to what extent the programs remain functional after applying the given filters, due to lack of time. However to increase confidence in these solutions, and to find potential problems in the various techniques it would be important to investigate this aspect as well.

References

[1] Maysara Alhindi and Joseph Hallett. Sandboxing adoption in open source ecosystems, 2024.

[2] Claudio Canella, Mario Werner, Daniel Gruss, and Michael Schwarz. Automating seccomp filter generation for linux applications. In *Proceedings of the 2021 on Cloud Computing Security Workshop*, CCSW '21, page 139–151, New York, NY, USA, 2021. Association for Computing Machinery.

[3] Nicholas DeMarinis, Kent Williams-King, Di Jin, Rodrigo Fonseca, and Vasileios P. Kemerlis. sysfilter: Automated system call filtering for commodity software. In *International Symposium on Recent Advances in Intrusion Detection*, 2020.

[4] Masoud Ghaffarinia and Kevin W. Hamlen. Binary control-flow trimming. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, page 1009–1022, New York, NY, USA, 2019. Association for Computing Machinery.

[5] Seyedhamed Ghavamnia, Tapti Palit, Azzedine Benameur, and Michalis Polychronakis. Confine: Automated system call policy generation for container attack surface reduction. In *International Symposium on Recent Advances in Intrusion Detection*, 2020.

[6] Seyedhamed Ghavamnia, Tapti Palit, Shachee Mishra, and Michalis Polychronakis. Temporal system call specialization for attack surface reduction. In *Proceedings of the 29th USENIX Conference on Security Symposium*, SEC'20, USA, 2020. USENIX Association.

[7] Chenxiong Qian, Hong Hu, Mansour Alharthi, Pak Ho Chung, Taesoo Kim, and Wenke Lee. Razor: A framework for post-deployment software debloating. In *Proceedings of the 28th USENIX Security Symposium*, Proceedings of the 28th USENIX Security Symposium, pages 1733–1750. USENIX Association, 2019. Publisher Copyright: © 2019 by The USENIX Association. All rights reserved.; 28th USENIX Security Symposium ; Conference date: 14-08-2019 Through 16-08-2019.

[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] Zhiyuan Wan, David Lo, Xin Xia, Liang Cai, and Shanping Li. Mining sandboxes for linux containers. In *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pages 92–102, 2017.