System Call Argument Filtering for Interpreted Languages

S.L. Maquelin

Technische Universiteit Delft

TU Delft

System Call Argument Filtering for Interpreted Languages

by

S.L. Maquelin

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on July 1st 2022.

Student number:	5402840	
Project duration:	September 6, 2021 – July 1, 2022	
Thesis committee:	Dr. A. Zarras,	TU Delft, supervisor
	Prof. G. Smaragdakis,	TU Delft
	Prof. D. Spinellis,	TU Delft

An electronic version of this thesis is available at `http://repository.tudelft.nl/`.

TU Delft

Preface

This thesis explores the possibilities of system call argument filtering for interpreted languages. Starting this thesis, my knowledge of system call filtering was limited. Only after some initial literature research did I understand that the focus would mainly be on policy generation. During my research, I investigated two options to adapt the policy generation from system calls to system call arguments.

This thesis completes my Cyber Security specialisation of the Computer Science Master at the TU Delft. As the course Network Security had interested me, I asked one of the teachers, Apostolis Zarras, to be my thesis supervisor. He helped me find a subject by handing me several papers to examine and even more when I could not find the right one. The Saphire paper finally gave me inspiration for a thesis topic.

The start of my master's in September 2020 was marked by corona. A year later, at the beginning of my thesis, corona still influenced the course of events. During my thesis, all contact was online, which went well after a short adjustment period. In line with the rest of my masters, the thesis defence will be online, in the presence of my thesis committee, Dr Apostolis Zarras, Prof Georgios Smaragdakis and Prof Diomidis Spinellis.

I want to thank Marwan for our weekly meetings in which he motivated me to do a bit better every time and question my work. Furthermore, I wish to thank my supervisor for his guidance throughout the thesis and the weekly meetings, especially important during corona, as they encouraged contact with fellow students.

My boyfriend Luuk, my sister Eva, and my parents have always supported me and believed in me, at times even more so than I did. Thanks to their support and help, I enjoyed this time and created an even better thesis. Finally, I thank Adriaan, a friend from the start of this master's. He was always available to discuss technical details and have us help each other out.

I wish you pleasant reading.

S.L. Maquelin
Groningen, June 2022

Abstract

Interpreted applications are often vulnerable to remote code execution attacks. To protect interpreted applications, we should reduce the tools available to the attackers. In this thesis, we investigate the possibilities for the automation of policy generation for interpreted applications in terms of system call arguments. These policies are used for system call argument interposition. We compare two approaches working on the interpreter to find if any of these two can provide meaningful policies. The first is dynamic analysis, and the second is static analysis, which uses symbolic execution.

The symbolic execution was least effective as it provides policies only for a small portion of the system call arguments, less than ten per cent, and hinders normal execution of applications with these policies. The dynamic analysis solution fares better, providing a restriction for about forty per cent of the system call arguments. We conclude that automatic policy generation of system call arguments for interpreted applications is a meaningful endeavour.

Contents

1 Introduction

1.1. Problem statement

Creating software means building on the work of others. As a result, software projects often have many dependencies and are more extensive than can be seen at first glance. An example of this is the Log4j program, which is used by many others. A vulnerability in the Log4j program [16] left a lot of software vulnerable to attacks. Specifically, the Log4j program contained a Remote Code Execution (RCE) vulnerability, which allows an attacker to run arbitrary code from a remote location. In about ninety per cent of the RCE attacks, an attempt at crypto mining is made [8], but an RCE vulnerability can have even more devastating effects. The vulnerability in Citrix Netscaler [29] announced at the end of 2019 was an RCE vulnerability that left thousands of companies around the world vulnerable, including hospitals. An RCE attack can serve as a starting point for confidential data retrieval or ransomware attacks.

Web applications are very interesting to attackers as they provide services and hold sensitive information, such as customer and financial data. The majority of web apps are written in an interpreted language [7]. Often web apps are managed through content management systems, such as WordPress, which use an interpreted language. Focus on enhancing security for interpreted applications can help minimise the impact of many RCE attacks.

1.2. Existing solutions

An RCE attack relies on system calls to interact with the system's resources. Applications can often use all system calls provided by the operating system while needing only a (small) subset to perform their tasks. Especially for web applications, which deal with untrusted input from a client, regulating the access to system calls will improve security. Therefore, the regulation of system call access is the focus of research on minimising the impact of RCE attacks. Researchers want to apply the Principle of Least Privilege (PoLP) applied to system calls. PoLP dictates that an application should have access to those resources necessary for proper execution but no more than that. When applying this principle, the first task is to deduce which resources, in this case system calls, are needed for the proper execution of an application. The research community found that system call interposition is an appropriate solution to minimise the impact of RCE attacks.

System call interposition intercepts system calls before they can be executed and check if they comply with the preset rules. These rules can be allowlists or blocklists, or take a more complex form. Many different implementations for system call interposition are proposed [15, 19, 20]. These systems work with a predefined rule set on which system calls to (dis-)allow. Therefore, the automation of generating such policies is a current field of interest [12, 13]. The generation of policies for interpreted applications is more complicated than for compiled applications since the interpreter creates an extra level of abstraction. In [11] a framework called Saphire is proposed to tackle this.

It is possible to apply PoLP in a stricter sense by filtering system calls on their arguments. For example, a policy could state which arguments are allowed for a particular system call. Such policies would decrease the attack space even further. However, filtering system call arguments is more complex than just filtering system calls. The amount of literature on this subject is limited. Systrace [25], Shredder [22], Abhaya [24], and [26] take on the task of determining the system call arguments used in the normal execution of the program to

enhance their system call interposition. They all work with compiled applications, but they differ in aspects such as tackling source code versus binaries and their analysis type: static or dynamic.

1.3. Limitations of existing solutions

The existing literature gives us a good idea of how to tackle the minimisation of the attack surface for RCE attacks. Moreover, it shows us that it can be taken even further by including system call arguments for interposition. However, the application to the setting of interpreted applications has not yet been tackled.

Saphire provides an approach for system call interposition for interpreted applications but does not work with system call arguments. Where Saphire uses primarily static analysis for the policy generation, this could also be replaced by Systrace to take care of the system call arguments. The downside to Systrace is that it works with dynamic analysis and therefore requires test input for each application to generate policies. The Shredder framework is also not suitable to replace the policy generation in Saphire as it is aimed at the Windows operating system. The third framework tackling system call arguments is Abhaya, which only works with source code and can not analyse the interpreter binary. The last framework working with authenticated system calls might be used to analyse the interpreter binary. However, changes should be made to account for its structure – because the execution path of the interpreter heavily depends on the input – and for its size. Even if this adjusted framework or another could replace the policy generation in Saphire, we still have no idea if that would yield good results.

1.4. A new approach - an extension to the Saphire framework

It is not clear if filtering the arguments of system calls in the setting of interpreted languages is feasible and worth the effort. To get a better idea about the effectiveness, we aim to evaluate to what extent automation of policy generation on system call arguments allows for valuable system call argument interposition. The research question of this paper is thus “To what extent can system call argument filtering be applied to interpreted applications?". To answer the research question, we need to specify the requirements for satisfactory filtering. We will combine the answers to three subquestions to answer our research question. These subquestions are:

1. What percentage of arguments can be filtered?
2. How important are these arguments with respect to security?
3. Does the filtering impact the normal execution of the application?

To answer the research question, we build upon the Saphire framework. The Saphire framework already takes care of the filter creation and application to the kernel, so our focus will mainly be the policy generation for the system call arguments to create adapted filters. Furthermore, the Saphire framework is implemented for the interpreted language PHP, which will thus be used for our research. We consider both a static and a dynamic analysis version for the policy generation. We test both options as the output differs a lot, and we want to find out which of these two, if any, is best suited for the task.

The system call argument interposition that we look into for this research could be applied to all interpreted applications in a Linux environment. However, it would be most applicable to web applications because they deal most with client input and are thus more vulnerable to an RCE attack. Also, the interposition can be applied to containers. Docker, for example, allows users to create their own (seccomp) filter [1]. Docker provides a general filter but tailoring to the specific application used in the container could provide even better security.

1.5. Contributions

In summary, we make the following contributions:

- We utilise the Saphire framework and show which parts need to be extended to create system call argument interposition. The most important part is the analysis of the interpreter, which is partly responsible for the policy generation.
- We replace Saphire’s analysis on the interpreter with two different forms, a static analysis in combination with symbolic execution and a dynamic analysis.

- We evaluate both approaches based on the above subquestions and using three PHP applications: phpMyAdmin, Drupal and Joomla. The dynamic analysis solution has promising results, restricting around 40 per cent of the system call arguments when applied to each application.

1.6. Reading guide

The reader can expect to find the following in this thesis. Chapter 2 describes concepts such as system calls, remote code execution, system call interposition, interpreted applications, and dynamic and static analysis for background information. It also provides a summary of the Saphire framework. The system architecture discussed in Chapter 3 is split into a design and an implementation part and elaborates on the additions to the Saphire framework. Chapter 4 evaluates and compares the two options for the generation of policies on system call arguments. We discuss these results to answer the research questions and the limitations of both the architecture and experiments in Chapter 5. The literature related to system call interposition and automation of policy generation is discussed in Chapter 6. Chapter 7 concludes the thesis.

2

Background information

This chapter provides background information on key concepts used in the Saphire framework and our extension of this framework. These key concepts are system calls; RCE attacks, for which we want to minimise their impact; system call interposition, the tool to obtain this minimisation; interpreted applications, which determine the scope of our research; and dynamic and static analysis, necessary to automate the policy generation for the interposition. The Saphire framework is described in Section 2.5.

2.1. System calls and their arguments

System calls provide an interface to the operating system. For example, they can be used to interact with the file system. Both read and write operations as well as setting permissions are coordinated with system calls. Which system calls exactly are implemented depends on the operating system. Normally, every system call is available to any program, and (almost) every program makes use of system calls. The number of system calls used by a program is only a (small) portion of the system calls provided by the operating system, which can be hundreds. System calls are powerful because they can (re)set permissions and execute files. For example, `chmod(const char *pathname, mode_t mode)` changes the mode bits of a file, such as permission bits and set-group-ID [21]. Most important are the arguments. Depending on which values for the arguments `pathname`, indicating the file, and `mode` are given, the system call can make minor or major changes. Replacing the read-write permissions with read-only permission on a cat picture may not be very important. However, allowing other users to access intellectual property documents may cause more problems.

When a system call is used, the system call number, identifying the system call, is written to a predetermined register. For the x86-64 architecture, this is the rax register. The arguments of the system call also go into predetermined registers. For example, the rdi register always stores the first argument on x86-64.

2.2. Remote Code Execution

Arbitrary Code Execution (ACE) means the ability to execute arbitrary commands/code. If this can be done on a remote machine, it is called Remote Code Execution (RCE). In an RCE attack, the attacker gains this ability. As the attacker can execute any program, it can read from the filesystem, possibly stealing intellectual information or (salted) passwords. An attacker can also encrypt all files, launching a ransomware attack. In an RCE attack, a vulnerability in the system is exploited. Such a vulnerability is called an RCE vulnerability. This type of vulnerability is often exploited in web applications. The attacker has access to the client application and wants access to the back-end, where often confidential information is stored. An example of an RCE vulnerability is the one in Drupal, a web content management system, in 2014 named Drupalgeddon. “The database abstraction API did not properly construct prepared statements, which allowed remote attackers to conduct SQL injection attacks" [23] which could be used to add an admin user [33]. Detection of such vulnerabilities is complicated. Vulnerabilities are often found only after exploitation. Preventing RCE attacks without knowledge of the vulnerability is hard as well. Another strategy can be to minimise the impact an RCE attack can have. This thesis uses this last strategy.

2.3. System call interposition

System call interposition can regulate the system calls used by a process, increasing security. Normally, without interposition, system call execution works as follows. When a program operating in the user space invokes a system call, it sends a request to the kernel. The system call is executed in the kernel, and its result is returned to the program. Using system call interposition, the system call is evaluated after the program's invocation and before the kernel can execute it. The evaluation of the system call determines whether or not it is allowed to be executed. The evaluation can be a simple allowlist approach or of a more complex nature. System call interposition allows for more control over system calls and, therefore, better security. A downside to system call interposition is the complexity of the evaluation part. It is hard to decide which system calls should or should not be allowed. When a system call is stopped, the program invoking it can crash.

There exist many systems for system call interposition, and they can operate in the user space [20] or the kernel space. The kernel-based systems, see Figure 2.1, are most common. The category a system belongs to depends on where the requested system call is checked for permission: user-space or kernel-space. An example of a kernel-based system is seccomp. Seccomp stands for secure computing. It was first introduced in the Linux kernel in 2005 and has evolved a lot since then. Seccomp uses the Berkeley Packet Filter language to create the filters that are loaded in the kernel. Seccomp is a basic non-automated system call interposition system, allowing the user to write rules per system call. It also allows restrictions on system call arguments with the limitation that they cannot be pointers. This limitation is because pointer arguments are stored in user space, unlike integer arguments that are stored in kernel space. The address of the pointer arguments is then communicated to the kernel. However, between the time of checking the arguments against the policy and the time of executing the system call with pointer arguments, the memory of the pointer arguments can be changed, creating race conditions. The limitation of seccomp to not work with pointer arguments prevents these race conditions; more on this topic can be found in Section 4.3 of [17].

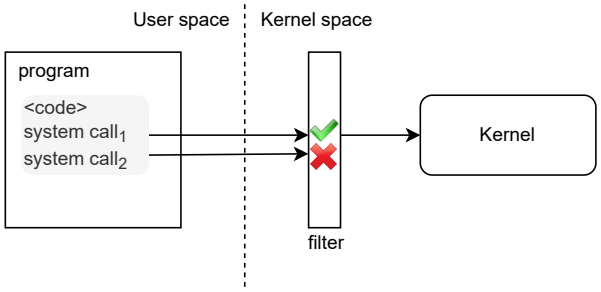


Figure 2.1: Schematic example of kernel-based system call interposition.

2.3.1. Policy generation

The crafting of policies that the filter for system call interposition uses to decide is manual or automatic. When a program is large, it is hard to do manual generation. The manual generation is very complex in the case of source code of a compiled application or an interpreted program. The system calls do not become apparent until the compiler and interpreter are taken into account. Therefore, the automation of policy generation is integral to a well-performing system call interposition. Automated policy generation for system call arguments is more complex than for system calls only. The arguments may depend on user input or be determined by many consecutive operations.

Since wrong or incomplete policies may be generated, the decisions based on these policies may be incorrect. These incorrect decisions are divided into false positives and false negatives. When a policy dictates that a system call should be allowed while it is not needed for the application's normal execution, it is called a false negative. Opposite, when a filter blocks a system call during normal execution of the application, it is called a false positive.

2.4. Interpreted applications

Most web applications are interpreted. At the start of February 2022, W3Techs reported that over 75% of the top 10 million websites use PHP for the server-side and the most popular content management system in use by far is WordPress, which is written in PHP [7]. The position of interpreted languages in web apps can

be attributed to the number of large frameworks allowing for easy web development and the ease of agile development in interpreted languages.

Two parts are needed to run an interpreted application: the application source code and an interpreter. An interpreter translates source code into machine instructions, same as compilers do. However, the interpreter will execute the machine instructions immediately instead of storing them in a binary for later use [10]. A schematic overview is shown in Figure 2.2. An advantage of interpreted applications is the ease of distribution. A binary is machine-specific, while an interpreted program can be executed on any machine with an appropriate interpreter. Due to the extra layer, the interpreter, the analysis of an interpreted application becomes more complicated.

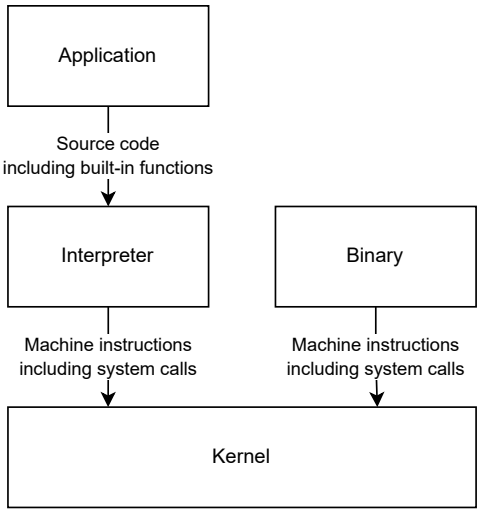


Figure 2.2: Schematic overview of the three levels in interpreted programs.

2.5. Saphire

To the best of our knowledge, the only system call interposition framework that focuses on interpreted applications is Saphire [11]. This thesis will build on this framework, and therefore the reader must have an idea of the general layout. The Saphire framework provides a “generic approach for automatically deriving system-call policies for individual interpreted programs". The framework is implemented for PHP programs and creates a seccomp filter for the derived system call policies.

The automatic derivation of system call policies for interpreted applications is more complicated than for compiled applications. This is due to the extra layer of the interpreter. Both interpreted and compiled applications use built-in functions, which in turn use system calls. To determine which system calls are used by a compiled application, one can inspect the binary. However, a binary is not available for interpreted programs. To determine the system calls used in an interpreted program, both the source code and the interpreter must be inspected. The source code should be searched for built-in function calls, and, inspecting the interpreter, one can determine which system calls are needed for these built-in functions.

Saphire consists of three stages, focusing on the interpreter binary, the application source code, and the kernel. The first two stages generate the system call policies, and the third stage uses these policies to generate filters which are applied to the kernel. Figure 2.3 shows an overview of the framework.

The first stage applies both static and dynamic analysis to the interpreter binary with debugging symbols. The goal is to create a list of necessary system calls per API function. The static analysis is applied first, where the binary is inspected line by line, and a call graph is created. Then, the dynamic analysis enhances the result of the static analysis and handles indirect calls.

The second stage is applied to the interpreted application source code. The API calls per script are extracted from the source code and combined with the list from stage one. The result of this stage is a list of system calls for each script in the application. The source code is parsed to identify all the API calls, and an abstract syntax tree is created and analysed.

The third stage receives the result of the second stage and creates a seccomp filter for each of the scripts allowing only those system calls that are on the list.

When multiple applications use Saphire or an application is updated, stages 2 and 3 need to be rerun. However, the output of stage 1 stays the same until the interpreter is updated and is therefore only run once.

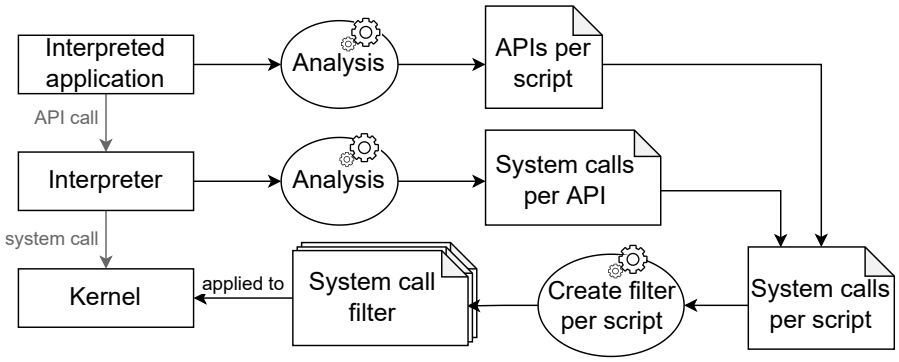
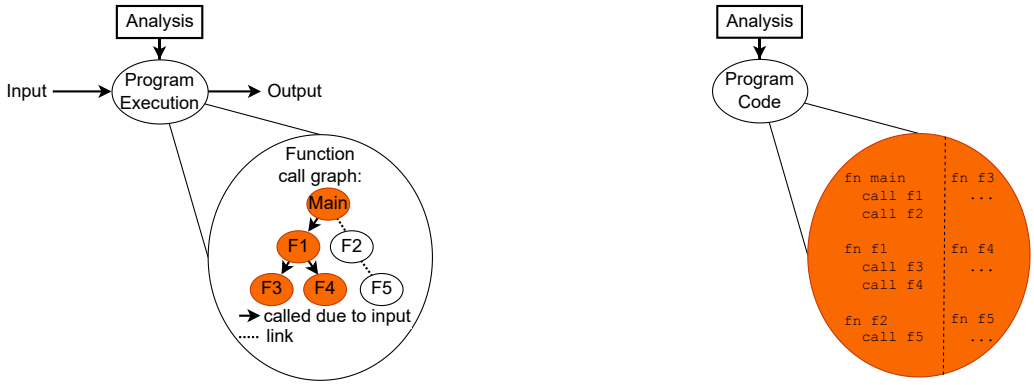


Figure 2.3: Schematic overview of Saphire framework.

2.6. Dynamic versus static analysis

In the section discussing Saphire, static and dynamic analysis are mentioned. In Saphire, the interpreter binary is analysed to find system calls. Static and dynamic analysis can also be used on programs to do error detection. In this section, their definitions will be discussed and the (dis)advantages of both methods and how they can be combined.

Static analysis is some form of analysis of a computer program such that the program does not need to be executed. Dynamic analysis, on the other hand, does need the program to be executed. Consequentially, the performance of dynamic analysis depends a lot on the input given to the program for execution. Obtaining good test input, in the sense that it reflects real-life input and rare outliers, is therefore essential. Figure 2.4 visualises the difference between static and dynamic analysis and the coverage (in orange) that both methods can obtain.



(a) Dynamic analysis

(b) Static analysis

Figure 2.4: Schematic overview of dynamic and static analysis for comparison

The downside of dynamic analysis immediately shows an advantage of static analysis. Static analysis often allows for more extensive code coverage. However, depending on the goal of the analysis, the unavailability of input can also be a disadvantage. The execution path often depends on the input, and the analysis becomes more complicated when it is not available. Symbolic execution deals with the lack of input by assuming symbolic values where input is used. When any constraints on the symbolic value are found, this information will be stored in an expression. Every execution path allowed by the symbolic value is analysed. However, when the instruction pointer depends on a symbolic value that is not restricted in any way, the analysis should be done for every possible value, leading to a possibly infeasible number of paths to be analysed.

The symbolic execution is sometimes considered a form of static analysis and other times of dynamic analysis. In this paper, we will regard it as static analysis because the information to execute the symbolic execution is gathered by static analysis and it does not require any input, unlike dynamic analysis.

Since static and dynamic analysis have converse up- and downsides, combining them can lead to a better result. Often, both can be performed, and their results, such as errors, or, in the case of Saphire, system calls, can be added together.

3

System architecture

This chapter discusses both the design and the implementation. Some design choices are discussed for the design, the reasoning that led to these choices and the actual design – the most dominating design part building on top of the Saphire framework. As Saphire provides a basis for our design, we will indicate to which parts of the framework we make our additions. Besides the Saphire framework, both existing literature and the research questions discussed in Chapter 1 gave rise to the additional design choices that were made. The design has two main parts, dynamic analysis and static analysis. Of these two, static analysis requires the most extensive design. Implementation details are discussed thereafter, including a discussion on the chosen tools and more details on the filter creation. The system described below is available at *https://github.com/suzannemaquelin/System-Call-Argument-Filtering-for-Interpreted-Languages*.

3.1. Design choices

There are two design choices to elaborate on before discussing the system's design. The first is to restrict only integer arguments and to exclude pointer arguments. The second is using static analysis and dynamic analysis on the interpreter binary.

The first choice is due to the related work [17]. It describes the pitfalls of system call interposition and points out that it is hard to restrict pointer arguments as it can create race conditions. Restricting pointer arguments without taking additional steps would falsely advertise security. An example of such additional steps is authenticated system calls, discussed in Section 6.3.4. Due to the implementation of Saphire, using seccomp filters, we do not attempt to take such steps and will not be restricting pointer arguments.

The second design choice is led by the research question “To what extent can system call argument filtering be applied to interpreted applications?". Working with the Saphire framework, the most important change we make is on the analysis of the interpreter. We take two approaches and compare them to give an informed answer to the research question. Using a hybrid analysis, Saphire applies both static and dynamic analysis. Our first approach extends the static analysis with symbolic execution. Our second approach is to replace the static analysis of Saphire with only dynamic analysis.

3.2. Design

To understand the extensions made to the Saphire framework, we show a schematic overview of the framework and indicate extensions with a blue E in Figure 3.1. The first stage of Saphire exists of the analysis on the interpreter, the second row in the figure, and results in the system calls per API function. The extension on the analysis part creates an extended output with system calls and their arguments per API. Saphire analyses the interpreted application in the second stage, retrieving the APIs per script. This stage is not changed for our extension. The results of the first and second stages are entered into a database. In the third stage of Saphire, a script retrieves information from the database to create system call filters that are loaded into the kernel. Since the extension changes the information in the database, this third stage is also adapted and creates filters for system calls and their arguments. The following sections discuss the extension of the first stage of the Saphire framework. The third part has only implementation consequences and will be discussed in Section 3.3.

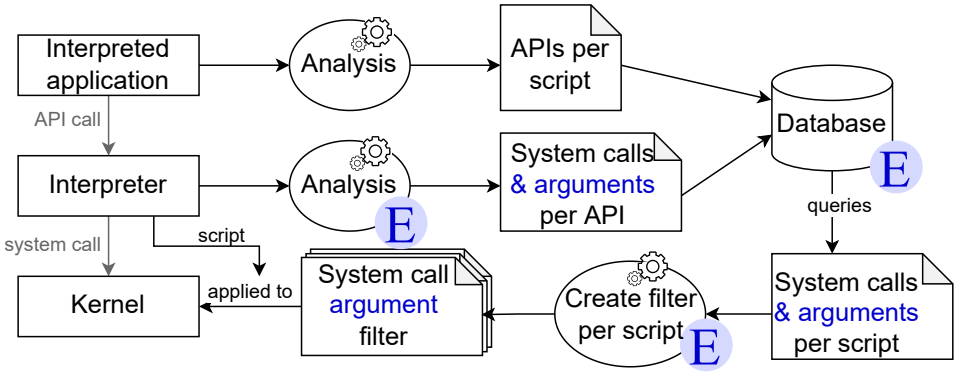


Figure 3.1: The Saphire framework with extension markings

3.2.1. Dynamic analysis on the interpreter binary

Dynamic analysis of the interpreter binary requires the execution of the interpreter and the recording of the system calls and arguments. Hence, all system calls used during execution and the values for their arguments will be recorded. However, due to the nature of dynamic analysis, this set is probably incomplete. Some arguments are more likely to have an incomplete set of values than others. Those that are likely to be incomplete should not be restricted as this would cause false positives, where a system call argument being blocked erroneously, during interposition. We identify five groups to divide the arguments into and evaluate for each whether they should be included in the interposition:

- pointer
- flag, mode and memory protection
- length, size and offset
- file descriptor
- other

We have already decided to leave the pointer arguments out of the interposition. The flag, mode and memory protection arguments are significant for security as they restrict permissions. Therefore, it would be preferable to use these for the interposition. The length, size, and offset group is a bit more complicated. It would be good to restrict such arguments, but it could be difficult to get reliable values for such arguments using dynamic analysis because they may depend on the input. Therefore the interposition will be tested with and without this group of arguments. The fourth group exists of file descriptor arguments that regularly depend for their value on the output of other system calls. Often, file descriptors cannot be reliably determined and thus will be left out of the analysis. The other group will be part of the analysis as there are no reasons as of yet to exclude them.

In conclusion, the values for the argument that fall in the category flag, mode and memory protection or other will be used during the analysis. File descriptor and pointer arguments will be disregarded during analysis, and the length, size, and offset arguments will be further investigated in Chapter 4.

3.2.2. Static analysis on interpreter binary

The original Saphire framework already uses static analysis to determine the largest part of the system calls. The structure created for this analysis will be used and built upon. The extension exists of three parts, where the first relies heavily on Saphire while the second and third are entirely new.

Compile information on API functions. The first component of the static analysis on the interpreter binary aims at collecting certain information for each API function. This information includes which functions and system calls are reachable by each API function. The information is grouped per API function because of how interpreted applications work. An interpreted application depends on API functions to perform system calls. The API functions an application uses are thus indicative of which system calls it uses. The mapping from API functions to system calls, combined with knowing which API functions an application calls, determines

which system calls are indispensable. The result of this component is a mapping from API to system calls and reachable functions. The second part of the mapping, the reachability part, is an addition to Saphire. The next component makes use of both parts.

Determine viable paths to system calls. Multiple paths may exist from an API function to a certain system call. Figure 3.2 shows an example. This component aims to determine which paths lead to a certain system call within the set of functions reachable by the API. We only want to include functions that are reachable for the given API because the system call arguments need to be determined and restricted per API call. Thus the paths are computed for a given system call and API function. It is important to find every path, as each can create different arguments for a system call.

Instead of determining the whole path leading from API function to system call, subpaths of a certain length are found. The whole path may be rather long, and only the last few functions leading to the system call will be inspected for the next part of the analysis. A subpath includes the last part of a path with the system call at the end, but not necessarily the first part including the API function.

As shown in Figure 3.3, a system call can be called in multiple functions and, therefore, can have multiple sets of subpaths. In this figure, for the example, the sets of subpaths of length 2 are {(function III, function I), (function III, function II)} and {(function V, function IV)}.

The sets of subpaths per system call per API function is the output of this component.

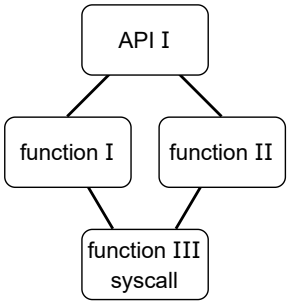


Figure 3.2: Multiple paths leading to one system call for one API

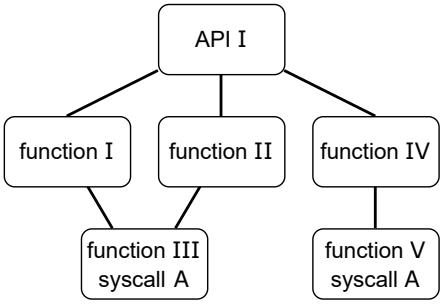


Figure 3.3: The same system call in multiple functions

Identify syscall argument values. In the third component, the values of the system call arguments are to be determined. As discussed in Section 3.1, only non-pointer arguments are considered. For each subpath given by the previous component, the values of the system call arguments will be determined. The determined values may differ per subpath, and thus this component outputs a set of argument values per system call per API function. Identifying the values of the system call arguments cannot be done by just inspecting the code at the system call address. The arguments will be stored in predetermined registers before the system call, and thus we need to reason about the code. Unlike the system call number, which is also stored in a register, the arguments for the system call may depend on other variables. Therefore the reasoning about the arguments could quickly go beyond the few lines leading up to the system call. We need some way of reasoning about the assembly, which we can do using symbolic execution. Recall that symbolic execution does not require any input, but it uses symbolic values instead and reasons about all possible execution paths. Symbolic execution needs to be given a start point and endpoint, which is the address of the system call. Due to the many execution paths in an interpreter, we develop guided symbolic execution. In essence, it exists of multiple symbolic executions, where the endpoint of one is the start point for the next and where all information is transferred between executions. The paths on which we do the guided symbolic execution provide the start and endpoints. The outcome of guided symbolic execution on a path may be one of the following three:

1. Concrete values for (part of) the system call arguments are found.
2. All values found are symbolic.
3. Symbolic execution fails to reach the system call.

The third option may raise some questions. How can the symbolic execution fail to reach the system call? There are several reasons for this to happen. First of all, although we try our best to determine the paths, due to the halting problem, we cannot be sure that the system call is indeed reachable from a given starting point. Secondly, we limit the paths in aspects such as length because we are constrained in memory and time. Third, the symbolic execution may depend on an unconstrained value, a symbolic value for which we do not know its constraints. Hence, instead of two or three options, the number of routes that the symbolic execution may take is infinite. In this last case, the value might be known if the symbolic execution would start at an earlier point as more information may be available, meaning the value may not be unconstrained. Figure 3.4 shows a flowchart on all the choices around the guided symbolic execution.

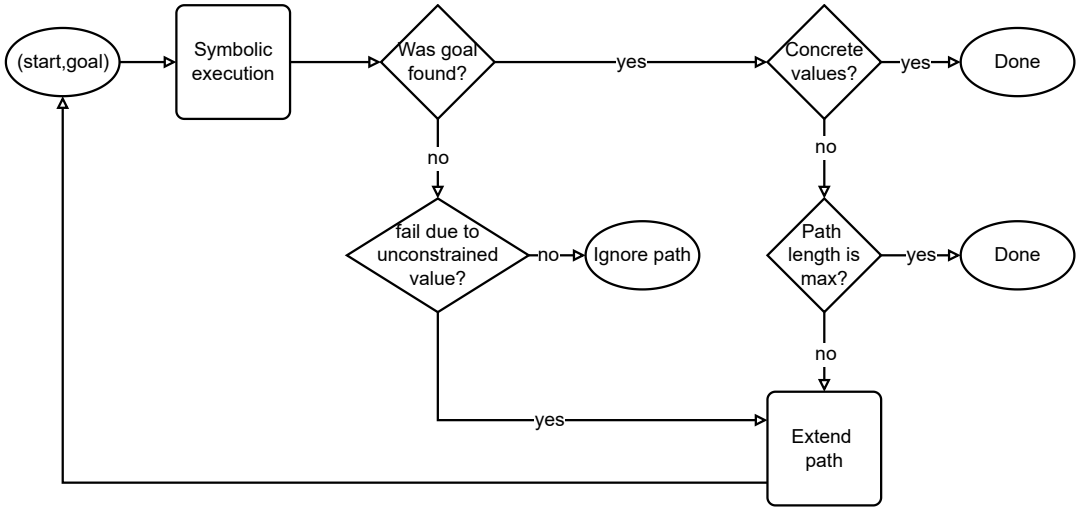


Figure 3.4: Flowchart for the guided symbolic execution

Of the three outcomes, the first outcome, finding concrete values, is desired. However, in the case of the second outcome and some cases of the third outcome, a new path may be constructed (up to a certain length) from the original path on which the guided symbolic execution can be done again. This may lead to a new result. Paths with the third outcome that are not qualified for new symbolic execution are considered inviable and omitted from further investigation. The argument can be restricted only when at least one concrete value is found, and all subpaths result in non-symbolic values or failure to reach the system call.

We give a theoretical example of the symbolic execution outcomes and new path constructions using Figure 3.3. We set the maximum length of a path to two and start with paths of length one. So, for system call A and API I, we consider the subpath sets {(function III)} and {(function V)}. The symbolic execution on the first path outputs all symbolic values for the arguments of system call A. The execution on the second path fails to reach the system call because of an unconstrained symbolic value. Thus, we extend both paths to length two and get {(function III, function I), (function III, function II)} and {(function V, function IV)}. Path (function III, function I) returns concrete values, and symbolic execution on path (function III, function II) fails. Hence, the first output with concrete values will be used, and the second path will be declared inviable. Path (function V, function IV) returns only symbolic values. Since the maximum path length is reached, this output will be used. In Table 3.1 an overview of the guided symbolic execution is shown. Two rows result in the state done and are therefore used in the final computation for the argument values. Due to the symbolic values in the last row, system call A will not be restricted, and the concrete values obtained earlier are discarded.

Table 3.1: Example for guided symbolic execution

Path	**Output**	**Action/State**
(function III)	symbolic values	extend path
(function V)	fails due to unconstrained value	extend path
(function III, function I)	concrete values	done
(function III, function II)	fails	ignore path
(function V, function IV)	symbolic values	done

Overview. The three components are used sequentially to determine as many system call argument values as possible while working with certain restrictions. The first component is only a small addition to Saphire but is essential for the second component to determine which paths belong to a certain API function. The third component uses these paths to determine the start and endpoints for the symbolic execution to determine the values for the arguments of system calls. Figure 3.5 shows the three components.

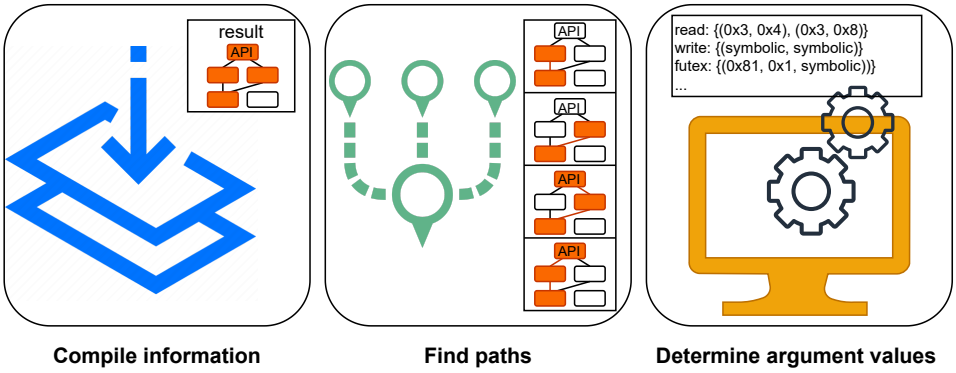


Figure 3.5: The three components of static analysis that work on the interpreter binary.

3.2.3. Design overview

The extension of the first stage of Saphire exists of using either an addition to the static analysis already used by Saphire or the replacement of this analysis by dynamic analysis. The updated analysis of the interpreter leads to a more extensive output, including values for the arguments of the system calls, which is utilized to create a more extensive system call interposition.

3.3. Implementation

In this section, we discuss the implementation details of both dynamic and static analysis as well as the filter creation. One implementation detail used by both analyses is the use of the Linux man pages to retrieve information on the arguments of each system call. This information is then used to determine which arguments will be analysed and restricted and what their index is.

3.3.1. Dynamic analysis

The design section on the dynamic analysis only discussed the argument types included in the analysis, but not the actual analysis. We use the tools strace and Xdebug to analyse and retrieve all system calls and their arguments. Afterwards, the arguments needed for the filter creation will be selected and stored in the database.

Strace and Xdebug. Strace is a diagnostic tool on Linux, which intercepts and records the system calls called by a process and the signals received by a process [21]. Xdebug is a PHP extension and provides us with another tracing feature [27]. It can write every function call, with arguments and invocation location to disk. The combination of the tools strace and Xdebug is chosen for two reasons. The first reason is that Saphire already used them and thus would require minimal effort to adjust for our goals. Second, strace allows us to track both system calls and their arguments, precisely what we need.

While running the interpreter test suite on the interpreter, strace executes. Strace does not require much configuration to work, but a relevant change that we made compared to Saphire is to add the flag `-X raw` to retrieve the system call arguments in raw format. We do not have to convert flags and modes from strings to the appropriate integers using this format. It is important to have all arguments in an integer format as the chosen filter requires this format. Some of the arguments are given in hexadecimal and still need to be converted to integers. Strace gives the information on both executed and failed system calls. The failed system calls are left out of the analysis, including invalid arguments.

Figure 3.6 shows a small overview of the dynamic analysis and the roles of strace and Xdebug in the analysis.

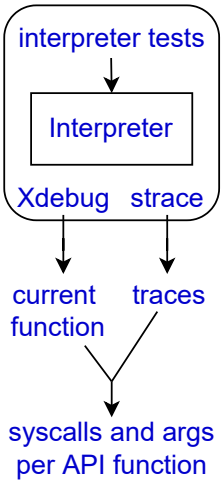


Figure 3.6: The flow of the dynamic analysis

3.3.2. Static analysis

The static analysis is divided into the same parts as in the design: the compilation of information on API functions, which is implemented mainly by Saphire; the determination of viable paths to system calls; and the identification of system call argument values, done through symbolic execution using the tool angr.

Compile information on API functions. Multiple systems in the literature review, Chapter 6, use a control flow graph. Such a graph provides a lot of information but, in turn, takes a lot of work to create, especially for large programs. As the interpreter with all its shared libraries is very large, we choose to use a call graph combined with some heuristics instead, which we discuss in the next paragraph.

Saphire creates the call graph by reading the binary line by line and creating appropriate nodes and edges. The edges are created when a call operation is encountered. Depending on the called address, one or two edges are created from the callee address to the called address and from the called address to the function start address if these are not equal. Saphire also keeps track of the system call addresses. Our extension to this part is a new graph structure to keep track of some additional edges. The new graph makes the traversal easier for the next component. Saphire uses a second function to do a depth-first search on the created call graph. This technique traverses the graph from API to system calls and returns the set of system calls used per API. This function is ideal for determining which functions are reachable from a certain API function. This new information will be used by the second component as well.

Determine viable paths to system calls. In the design, we have determined that a path exists of a number of functions reachable by the given API function, leading to the system call and not necessarily including the API function. To determine which functions make up a path, we start at the system call and check in which function it is called. Then, the start of this function will be included in the path. After that, we recursively check whether other functions call the function last added to the path. Finally, the addresses called by these functions will be added to the path if:

- the functions are reachable from the API function;
- and the called address is smaller than the address last added to the path;
- and the called address and the address last added to the path are not too far apart.

The last two points are heuristics to create a set of paths that have the greatest chance of leading to the system call. We want to limit the number of paths to limit some work. For the first heuristic, the chance of returning to an earlier address during execution is small. For the second heuristic, when the distance between the start and endpoint gets larger, there is a greater chance of encountering an if statement or something similar that makes the instruction pointer skip the endpoint. A parameter determines this distance, and we run some experiments to determine its value.

As we do consider not only the code of the binary but also the shared libraries, the functions are spread out over multiple binaries. Therefore, we have to keep track of addresses relative to the start of the binary and the binary itself.

Identify syscall argument values. The tool angr is used for the symbolic execution. Requirements for a tool are that it provides symbolic execution on binaries, keeps track of register values, and is open source. Angr provides these and is maintained up until this year, 2022. It was created to make a usable tool with a longer lifespan than research prototypes and to incorporate state-of-the-art techniques [30–32].

Angr Configuration. Angr comes with many options and can therefore be tailored to our needs. For example, it allows us to load a binary with all its detected shared libraries and calls this a `Project`. On such a project, there are multiple configuration options, for example, the shared libraries being loaded, as is the case for us, or not. The `Project` is at the root of everything. It is used to create a `Simulation Manager` that runs the symbolic execution and keeps track of the states during this execution. A state contains memory and register values, which may be symbolic. The `Simulation Manager` also comes with several possible configurations. It can add techniques to the symbolic execution, for example. Further angr configurations are:

- The `Project` configuration `SimProcedures` is turned off: we keep angr from trying to replace external calls to library functions by symbolic summaries;
- The `Project` configuration to provide an additional search path for shared libraries is used. The libraries found in this alternative path that we provide contain debugging symbols required by Saphire for analysis.
- Path explosion is prevented by stopping any execution containing 30 or more active paths. A lower number than 30 would suffice, but we want to be conservative so as not to stop valid executions.
- A technique called `MemoryWatcher` is used on the symbolic execution to stop a process when it takes up too much memory. This happens when a path gets stuck in a loop or when a very long path does not reach its goal and takes up a lot of memory in this process.
- Another technique called `LengthLimiter` is used on the symbolic execution to stop the analysis on execution paths of a certain length. This length is determined by another parameter (the second we discussed so far). This heuristic stops execution on paths going in the wrong direction and never reaching the goal address. This parameter must not be taken too small.

Symbolic Execution. A guided symbolic execution is started for each path provided by the previous component. The symbolic execution is given a start address, the first address of the given path, and a goal address, the next address of the given path. A subsequent round of symbolic execution is required when the goal address is not equal to the system call address. Starting a subsequent round of symbolic execution, the state of the last symbolic execution will be written to the new execution as a starting state, and the goal address of the last execution will become the start address of the new execution. The new goal will be the following address in the given path.

One of the three states described in the design part is reached at the end of the guided symbolic execution. Namely, concrete values for system call arguments are found, all values for the arguments are symbolic, or the symbolic execution failed.

When the goal address is found, and the symbolic execution has found the system call, we inspect the values of the registers kept by the symbolic execution. Which registers are checked depends on the number of arguments of the concerned system call. The first argument always resides in the rdi register on an x86-64 architecture with a Linux operating system. The other arguments are linked to a fixed register as well. After inspection of the registers, their values are stored and linked to the system call and corresponding API function.

Overview static analysis. In Figure 3.7 all the components are shown with some of the implementation details. The first component creates a call and reachability graph by line by line analysis of the interpreter; the second component creates a set of viable paths through recursive analysis on the graphs created by the

first component; and the third component, the symbolic execution, reverts back to the second component multiple times before finishing.

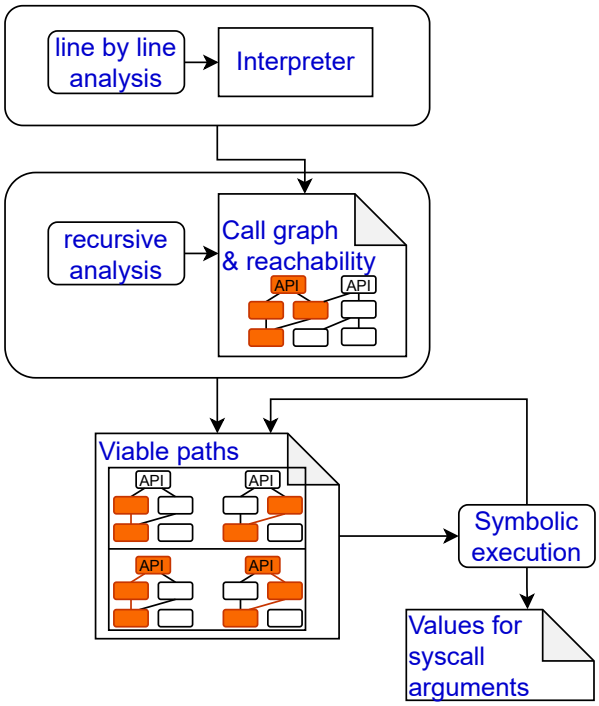


Figure 3.7: Schematic overview of static analysis components with implementation details

3.3.3. Filter creation

The filter creation is not included in the design as it keeps the structure determined by Saphire. We also decided to keep the same tools, namely a SQLite database and seccomp filters, as they fit our purpose. However, the framework allows to switch them for similar tools when preferred. We discuss the implementation details for these tools and why they suffice to achieve our goals.

SQLite database. SQLite is a relational database written in C [6]. Saphire enters the information it needs to keep track of into the database. This includes general information on system calls, such as which number corresponds to which name, and specific information on the system calls encountered in the interpreter. Furthermore, some information about the application is maintained, such as functions, function calls, PHP file paths and resolved includes. In the database, system calls are stored per API function. The used system calls per PHP file are determined using queries for all files that make up the application. These queries make great use of the structure a relational database provides.

Our extension to Saphire creates additional information on the interpreter that is entered into the SQLite database. Thus part of the structure of the database needs to change. Instead of storing each system call per API function, we store each valid option for the values of the system call arguments per system call per API function. SQLite can easily take care of the growth of the database. Since the information in the database changes, a small change is also required in the queries.

Seccomp. The application creating the seccomp filters is a PHP extension, so it can be run on the start-up of the interpreter. It needs to be run on start-up because the filters must be loaded before the interpreter is used. The interpreter supplies the information to determine which filter should be loaded, as the script that wants to use the interpreter is known. The application was changed to create filters that take into account the arguments of system calls. The number of arguments that will be restricted depends on the system call and the availability of values from the static/dynamic analysis. This number will thus depend on the output of an earlier stage. However, in the creation of a seccomp filter, one has to indicate the number of arguments that will be restricted beforehand, during implementation, and not dynamically. To overcome this problem, we

establish that the number of system call arguments we want to restrict never exceeds six and is at least zero and create an implementation for each case. Whenever an input contains more than six argument values, the application fails, and the interpreter will not be started.

4

Evaluation

We do some experiments to answer the research question and understand how well the static and dynamic analysis perform. The three research subquestions that provide an answer to the main question are:

1. What is the percentage of arguments that can be filtered?
2. Is the normal execution of the application impacted by filtering?
3. How important are the filtered arguments with respect to security?

4.1. Experimental setup

We run the experiments in Docker containers with operating system Debian 10, Linux kernel 5.4.0 and on an x86-64 architecture. The PHP interpreter version is 7.1 and was kept equal to the one used in Saphire. We select three of the applications used for the evaluation of Saphire for our evaluation, namely phpMyAdmin (4.8.1), Drupal (7.58) and Joomla (3.7.0). Each of these applications is a content management system. That means they provide an easy setup for creating and managing web pages, often providing modules and add-ons for easy development and a separate administration environment for management. Each of the applications uses a database, for which we have chosen MariaDB (10.3.34). PhpMyAdmin provides a user interface for MySQL administration [5]. Drupal aims at a wider range, allowing users to create websites through the use of frameworks and add-ons [2]. Joomla's purpose is similar to Drupal, allowing users to create a web application. Joomla makes use of a model-view-controller framework [4]. Each of the applications is open source.

4.2. What percentage of arguments can be filtered?

4.2.1. Static analysis

For the static analysis, there are three parameters we need to set a value for: the path length, which determines the number of functions in the path leading up to the system call; the distance, which determines the maximum distance between the starting address and the end address of one step in the path; and the limiter which determines the maximum number of paths within the symbolic execution before terminating. After some testing, the last parameter is set at a conservative value of 300. For the other parameters, we ran several experiments of which the outputs are reported in Table 4.1. The symbolic execution takes several days for these large values; therefore, we do not experiment with larger values.

The percentage of arguments found varies between 8 and 10%. We compute the total number of arguments by summing the number of arguments per system call for each system call used at least once in the interpreter. When the number of functions is kept equal at five, and the distance increases from 350 to 600, the arguments of the fcntl system call can no longer be restricted. Probably a path, which was not considered in the first case, was found where the values of fcntl could not be determined, therefore removing it from the restricted system call arguments. The increase did open up a path where the values for the system call arguments of writev could be found, which were added to the restricted syscall arguments. Overall we note that the increase of parameters does not increase the percentage of found arguments that much.

Table 4.1: Static analysis results for different parameters

No. of functions	Distance	No. of system calls	System call (arguments)	No. of arguments
5	350	4 (10%)	fcntl (cmd), futex (futex_op, val), openat (fd), ioctl (request)	5 (8%)
5	600	4 (10%)	futex (futex_op, val), openat (fd), ioctl (request), writev (fd, iovcnt)	6 (10%)
10	350	5 (12%)	fcntl (cmd), futex (futex_op, val), openat (fd), ioctl (request), poll(nfds)	6 (10%)

When the distance is kept equal at 350, and the number of functions increases from 5 to 10, the argument nfds of the system call poll is found for four API functions.

Not every system call argument in the table can be restricted for every API call in the interpreter. In Table 4.2 we show for each of the three applications which system call arguments we can restrict. Unlike phpMyAdmin and Joomla, the Drupal application does not restrict the request argument for system call ioctl. The API functions used by Drupal do not restrict this argument. Since not every API function restricts the system call arguments and not every API function is used by the applications, the percentage of restricted arguments drops to 3-4%.

Table 4.2: Restriction of arguments in applications for static analysis (5, 350)

Application	Restricted system call (arguments)	Percentage restricted arguments
PhpMyAdmin (4.8.1)	fcntl (cmd), futex (futex_op, val), ioctl(request)	4%
Drupal (7.58)	fcntl (cmd), futex (futex_op, val)	3%
Joomla (3.7.0)	fcntl (cmd), futex (futex_op, val), ioctl(request)	4%

4.2.2. Dynamic analysis

The dynamic analysis uses strace, which recovers every value for the arguments of the system calls used during execution. Hence one could say that 100 per cent of the arguments of the system calls used will be restricted. However, we have discussed in Section 3.2.1 that the file descriptor and pointer arguments are left out of the dynamic analysis. Hence we can compute the portion of non-file descriptor and non-pointer arguments out of all arguments. The system we used for testing has 334 system calls with 837 arguments. 354 arguments are left when excluding the file descriptor and pointer arguments. However, it is more interesting to see how many arguments the PHP interpreter uses and which portion of them are not file descriptor or pointer. We discuss possibly leaving out the size, length and offset arguments in Section 3.2.1 and thus compute which portion is left then as well. Table 4.3 shows both of these percentages and percentages computed for each of the applications.

Table 4.3: Portion of arguments that is inspected by dynamic analysis

Program	Ommited argument group(s)	Portion of argument to restricts
Interpreter (PHP 7.1)	file descriptor, pointer	88 out of 143 (62%)
Interpreter (PHP 7.1)	file descriptor, pointer, length/size/offset	74 out of 143 (52%)
PhpMyAdmin (4.8.1)	file descriptor, pointer, length/size/offset	34 out of 89 (38%)
Drupal (7.58)	file descriptor, pointer, length/size/offset	38 out of 94 (40%)
Joomla (3.7.0)	file descriptor, pointer, length/size/offset	47 out of 107 (44%)

Some arguments have only a limited number of options for their values. A good example of this is flock, which applies or removes an advisory lock on the open file specified by a file descriptor [21]. The second argument of flock is the operation argument and can only be one of the following three: LOCK_SH, LOCK_EX, LOCK_UN. The first two place a shared or exclusive lock, respectively, and the last removes the existing lock. Each of these can be ORed with a value LOCK_NB to make the request non-blocking. Hence, a total of 6 options are possible for this argument. Unfortunately, the interpreter requires each of these options. However,

this is when we look at all API functions together. In the case of flock, some API functions exist that only need one option. Only the flock API function requires all options to operate as intended according to our dynamic analysis. Checking the applications, it turns out the phpMyAdmin application does not require the flock API function and can thus be restricted, while Drupal and Joomla do require flock.

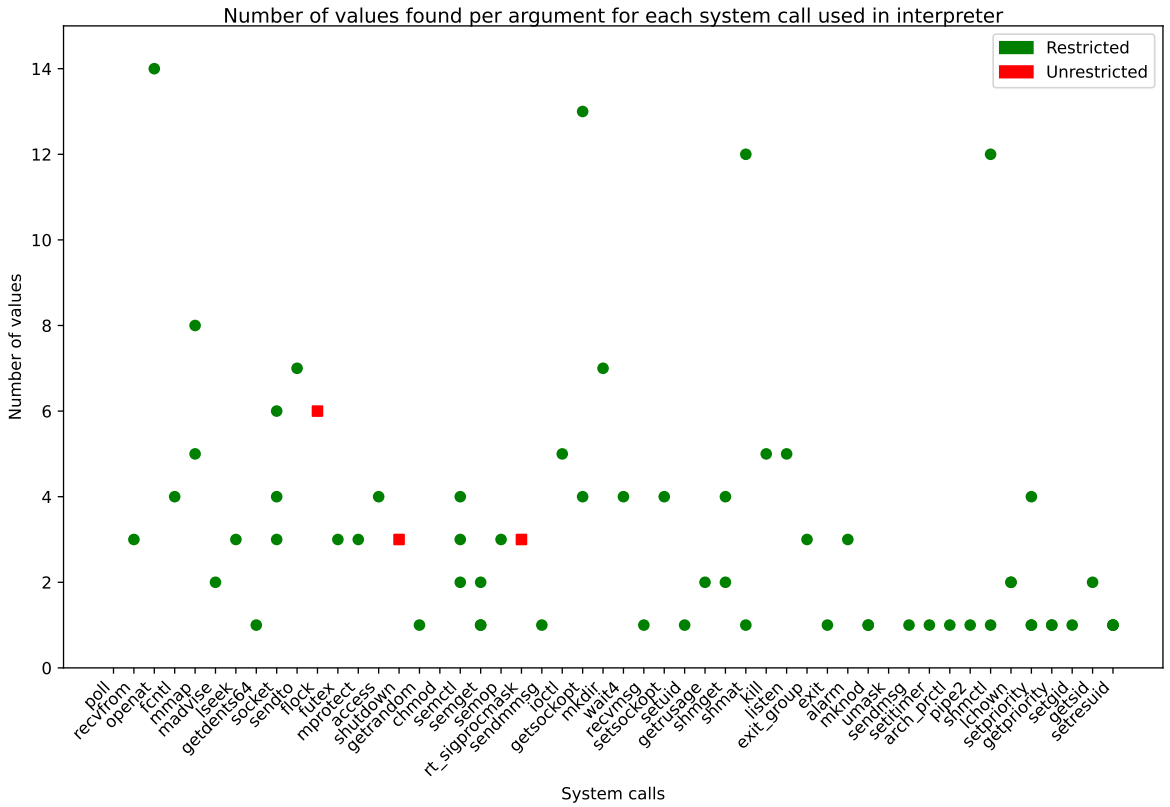


Figure 4.1: Indication of level of restriction for each system call argument in the interpreter.

Figure 4.1 shows for each system call argument whether it can be restricted or not. All system calls present in the interpreter binary are shown on the x-axis, and each dot represents a system call argument. The y-axis shows the number of values each argument can take according to the dynamic analysis. Eight of the system call arguments have more than 15 options and do not appear in the figure as this was too much to inspect manually. These points are given in Figure A.1 which can be found in Appendix A. For the evaluated arguments, 56 out of the 59 can be restricted.

4.3. Does filtering impact normal execution of applications?

To answer this question, we use the output of the analyses to create filters for the three applications, phpMyAdmin, Drupal and Joomla. Then, we test for each application if normal execution is possible with the filter in place and report any false positives for the filter. Recall that a false positive happens when the filter blocks a system call argument during normal, nonmalicious execution. The normal execution is evaluated using:

- the test suite packaged for Drupal;
- a test suite created through browsing traces with the browser extension tool called SeleniumIDE by the authors of Saphire for phpMyAdmin;
- and a similarly created test suite for Joomla.

4.3.1. Static analysis

We set the parameters for the distance equal to 350 and the number of functions equal to five and create filters for the applications with this static analysis. Table 4.4 shows the false positives created by these filters. The application phpMyAdmin stops execution when the test suite tries to log in. The system call argument cmd for system call fcntl is blocked. When adding the appropriate system call argument values to the allowed set, the table creation triggers a false positive for the system call argument futex_op of system call futex.

The filter for the Drupal application creates false positives for futex_op from futex and cmd from fcntl when logging in is attempted. However, after adding appropriate values to the filter, no more false positives are triggered during the test suite's execution.

Similar to phpMyAdmin, a false positive is triggered by fcntl when loading the login page of the Joomla application. After adding the required values to the filter, no more false positives come up during testing.

The extra system call, writev, found for parameters (5,600) for the static analysis, would not create other false positives as none of the applications uses it.

Table 4.4: False positives - static analysis with parameters distance=350 and no. of functions=5.

Application	Blocked system call (arguments)	Blocked action
PhpMyAdmin (4.8.1)	futex (futex_op), fcntl (cmd)	Load login page and create table
Drupal (7.58)	futex (futex_op), fcntl (cmd)	Load login page
Joomla (3.7.0)	fcntl (cmd)	Load login page

4.3.2. Dynamic analysis

The false positives are shown in Table 4.5. Each of the applications is blocked when loading the login page. The system call arguments that are blocked vary a bit per application, but they are all of the category size, length and offset. The new filters do not create any false positives on the tested applications when removing these arguments from the analysis.

Table 4.5: False positives - dynamic analysis with and without size, length and offset arguments

	Dynamic analysis includes size/length/offset arguments			Excludes
Application	False positives	System calls of blocked argument	Blocked action	False positives
PhpMyAdmin (4.8.1)	yes	mmap, munmap, write	Load login page	no
Drupal (7.58)	yes	munmap, connect, poll, sendto	Load login page	no
Joomla (3.7.0)	yes	munmap, sendto	Load login page	no

4.4. Are the filtered arguments important to security?

The answers to this question should give us insight into the importance of argument restriction. The paper [9] has composed a list of dangerous system calls and arguments. Most of these arguments are pointer arguments and fall outside of the scope of this project. Two groups of system calls and their argument are of interest to us, namely the setting of uid or gid (user id or group id) with system calls setuid, setresuid, setfsuid, setreuid and setgroups, setgid, setfsgid, setresgid, setregid. The dangerous parameters are uid and gid, respectively, set to zero. We will also investigate the security impact of other system call arguments through careful consideration of all values an argument can take and with the help of the man pages [21]. We examine the security aspects of several restricted arguments for the three applications using static analysis. Most of these are also restricted when using dynamic analysis. Since the dynamic analysis can restrict a much larger set of arguments, we do not discuss each of those. Instead, we will highlight the values these arguments can take in the applications and how those values could introduce security risks.

4.4.1. Static analysis

fcntl: cmd The cmd argument for the fcntl system call determines the operation performed on a file descriptor (the other argument to fcntl). A wide range of operations exists, such as duplication of the file descriptor or setting file status flags. As this argument allows a wide variety of operations, it improves security when we restrict this argument.

ioctl: request The request argument for the ioctl system call is a device-dependent code, which manipulates the underlying device parameters of special files such as the terminal. As this argument can be used for any operation, it would benefit security when we limit the allowed values for this argument.

openat: dirfd The dirfd argument is a directory file descriptor for the openat system call. It determines which directory is the basis for the pathname, another argument given to openat. The file descriptor has a special value for the current working directory. Restricting this argument only to allow the current working directory would help security in the case of file descriptor leaks. However, when other file descriptors should be allowed, the restriction would not provide much additional security.

futex: futex_op, val The futex_op argument for the futex system call determines which operation is performed on the futex, and the context of the val argument depends on futex_op. The operations describe various forms of wake and wait operations, as well as setting a private flag and a requeue operation. These do not seem to be of major importance to the security, but for the requeue operation, a vulnerability shows otherwise. The CVE-2014-3153 informs us of the risk of this argument in the Linux kernel through 3.14.5, as it “allows a local user to gain ring 0 control via the futex syscall. An unprivileged user could use this flaw to crash the kernel (resulting in a denial of service) or for privilege escalation."[3].

writev: fildes, iovcnt The writev system call writes iovcnt buffers of data to the file associated with the file descriptor fildes. Which data is written is determined by another argument. These two arguments are not likely to impact security.

poll: nfds The poll system call waits for one of a set of file descriptors to become ready to perform I/O. The nfds argument specifies the size of this set. This value may not exceed the maximum number of open file descriptors allowed for the process. A value nfds larger than the actual size of the given set will not allow the user to read into memory not owned by the process. We, therefore, consider this argument to be of low risk.

4.4.2. Dynamic analysis

The dynamic analysis can restrict a lot more arguments than the static analysis, and it is not feasible to discuss the security implications for each of them. However, some of the arguments discussed in the static analysis are also found by the dynamic analysis. For each of the values found in the dynamic analysis, we assess whether they impose a security risk.

As only part of the API functions provided by the interpreter is used by an application, it makes little sense to inspect all allowed argument values per system call in the interpreter. We could look at the argument values allowed per system call per API, but this would create almost 12.000 outputs to inspect as the PHP interpreter provides almost 4.000 API functions. So instead, we inspect the argument values of the system calls used per application.

For each of the arguments where the number of found values lies below 10, we manually inspected those values to establish whether any of them would pose a security risk. The result for application phpMyAdmin is shown in Figure 4.2. All system calls used by phpMyAdmin during static analysis are shown on the x-axis, and each dot represents a system call argument. The y-axis shows the number of values each argument can take according to the analysis. For two of the twenty-six arguments, we found values of interest. They are shown in Table 4.6. We will discuss the meaning and security risks of each.

The first argument is for the system call mkdir, which creates a directory. The mode argument specifies the mode, which means the access permissions, for the new directory. The mode argument is not solely responsible for the final mode of the new directory, the umask of the process also plays a role. The mode value 0777 allows read, write and execution rights. Therefore we consider this value a security risk. The umask somewhat tempers the risk as the final mode bits are determined by the AND bitwise operation on umask and mode.

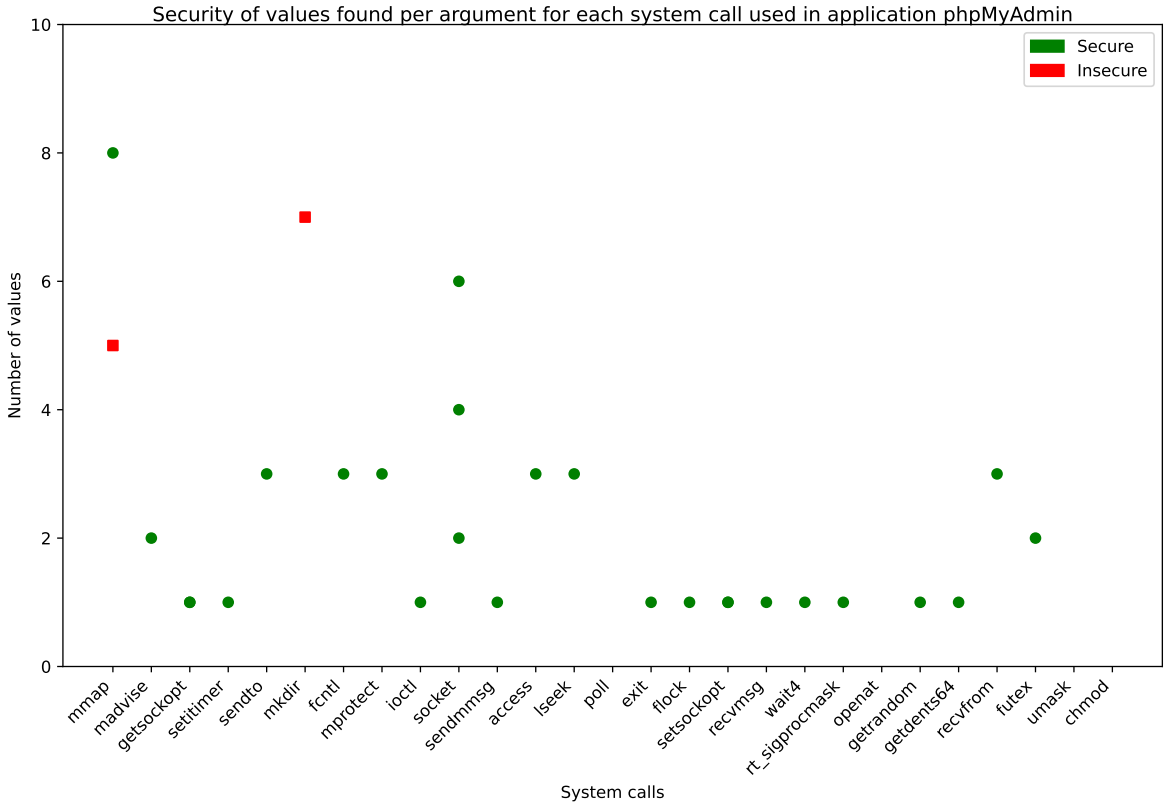


Figure 4.2: Inspection of security implications for system call arguments in phpMyAdmin

System call mmap, responsible for creating a mapping in the virtual address space of the calling process, has an argument which also receives a value of risk. Its third argument determines the memory protection and should equal a bitwise OR of one or more of the following: PROT_EXEC, PROT_READ, PROT_WRITE, PROT_NONE. Therefore seven options exist for this argument. Although two possible combinations are excluded, and the system call argument is restricted, the PROT_EXEC on its own or in combination with others is allowed. The PROT_EXEC allows pages to be executed and is required by almost 70 per cent of the files that make up the phpMyAdmin application.

Table 4.6: Dangerous argument values in the phpMyAdmin application

System call	Argument	Allowed dangerous values
mkdir	mode	0777
mmap	memory protection (prot)	PROT_EXEC

The API functions used in Drupal and phpMyAdmin are probably very similar, as the Drupal application has the same dangerous values as phpMyAdmin. However, the Joomla application has a few extra system calls, of which two have additional dangerous values for their arguments, reported in Table 4.7.

The first system call, setuid, which sets the effective user ID of the calling process, has one argument, namely uid, for which the zero value is allowed. The zero value for the uid argument means that the uid is set to root. This does not mean that a user with fewer privileges can use this to run a program as root. It does, however, have some safety concerns as the program would be allowed anything a root user is allowed. Similarly, the second system call, setgid, which sets the effective group ID of the calling process, allows the value zero for its gid argument. The system calls with dangerous argument values found in phpMyAdmin and Drupal are also found in Joomla and make up the last two rows of the table.

Table 4.7: Dangerous argument values in the Joomla application

System call	Argument	Allowed dangerous values
setuid	uid	0
setgid	gid	0
mkdir	mode	0777
mmap	memory protection (prot)	PROT_EXEC

Omitted argument groups The pointer, file descriptor, and length, size and offset arguments are left out of the dynamic analysis. To understand what this means for security, we investigate each of these groups. We start with the pointer arguments, which are the largest group. This group contains arguments that determine pathnames and filenames. Such arguments are listed as dangerous for the system calls chmod, chown, execve, mount, rename, open, link and create_module [9]. This group is thus important when it comes to security. The second group is the file descriptor arguments. File descriptors identify open files or directories. There exist three special file descriptors for standard input, standard output and standard error. Many system calls work with file descriptors; therefore, they can have a significant impact. However, the file descriptors are not just accessible to any process and therefore only form a security issue when they are exposed. The last group is the length, size and offset arguments, which have been excluded because of their dependence on the test suite. These arguments cannot easily be used for malicious activity as they are bound to maxima, and there exist other security measures. For instance, the read system call will not read past the end of the file.

5
Discussion

We discuss the results of the static and dynamic analysis separately and their interpretation for each of the research questions. We also compare the analyses to the Saphire framework, which our work extends. Thereafter, we discuss the limitations and future work.

5.1. Static analysis

The static analysis results indicate that only a small portion of the arguments can be restricted, around 4% for the applications. Increasing the parameter values further to minimise the number of paths left out of the analysis does not have much effect and is also very computationally expensive. Apart from the heuristics controlled by the parameters, the low percentage can be explained by a significant dependence on the API argument values. Thus, taking symbolic values for these means that we cannot determine the argument values. Hence, no matter what we try, the argument values cannot be determined without this information. As for the first reason, the heuristics, we have tried to cover as much as possible by varying the parameter values and allowing the symbolic execution to run for large amounts of time. For those arguments that are restricted, the values that were found result in false positives when testing the applications. We have two explanations for this:

- the path countering the restriction is not inspected due to one of the heuristics.
- the code in which the system calls with arguments are found is accessed through indirect calls. The static analysis does not solve these indirect calls; they are dealt with through dynamic analysis in the Saphire setup.

Overall the use of static analysis to do argument system call interposition is computation heavy and does not have the desired results. Furthermore, the usual advantage of static analysis not having any false positives falls away. Dynamic analysis should therefore be considered either on its own or in combination with static analysis when the computation cost for static analysis is permissible.

5.2. Dynamic analysis

The filters created with the results of the dynamic analysis yield false positives for each of the three applications. Studying the arguments that trigger these false positives, we find that they all fall within the category of size, length and offset arguments. Leaving these arguments out of the dynamic analysis creates filters that no longer result in false positives. These false positives indicate that this type of argument is often dependent on the values given to the API function arguments. Therefore, it is better to leave out such groups.

The analysis leaves out the pointer, file descriptor and size, length and offset argument groups, and we are therefore not restricting one hundred per cent of the system call arguments. The percentage of restricted arguments is around 40% for the distinct arguments in the applications. However, for most of the arguments that we do take into account for the dynamic analysis, the set of values found is a subset of all allowed values. Therefore, their values are restricted.

We categorise the arguments into two categories: secure and insecure. We label the arguments secure when we see no values from the analysis that indicate an increased risk. Fortunately, this holds for the majority of the found values. Some applications use API functions with more insecure argument values than

others. For example, Joomla requires the system calls setuid and setgid with argument value zero, described as dangerous by [9].

Overall, the use of dynamic analysis to create system call argument filtering looks promising. However, dynamic analysis will always come with the risk of false positives and should therefore be used in the appropriate settings where false positives have acceptable consequences.

Table 5.1: Comparison dynamic and static analysis

	Static analysis	**Dynamic analysis**
Restricted arguments	~4%	~40%
Fasle positives	multiple	none
Security impact	yes	yes

5.3. Comparison to Saphire

Having discussed the static and dynamic analysis, for which Table 5.1 shows an overview, we want to compare them to the Saphire framework to understand our work's additional costs and advantages. This will be a combination of qualitative and quantitative comparisons. We will discuss the aspects of time, false positives and negatives, effectiveness and, additional security.

5.3.1. Time

The Saphire framework prototype takes the most time on dynamic analysis as the interpreter test suite needs to be run, namely about an hour. Note that this part of the framework only needs to be run once, not for every application. Our static analysis does not require the test suite information. However, it needs to run symbolic execution, which takes a lot longer than the interpreter test suite, namely several hours, where the exact time depends on the parameter values. Our dynamic analysis works with the same traces retrieved by running the interpreter test suite as Saphire. As this takes up most of the time, the execution time of our dynamic analysis and Saphire are equivalent.

5.3.2. False positives and negatives

Saphire uses a combination of static and dynamic analysis. The dynamic analysis is required to analyse parts of the code accessed through indirect calls. Because this code is not analysed statically, it has the risk of leading to false positives. No false positives were found by the Saphire evaluation, however. The part of the code analysed by the static analysis has the opposite risk, false negatives. As the static analysis does not reason about the reachability of the system calls, it is likely there will be false negatives. Saphire does not investigate its false negatives. Our static analysis works with the system calls found by Saphire and, therefore, will yield false negatives for those argument values found for system calls that are false negatives in Saphire. Additionally, we investigate every path leading to a system call to find all possible options a value can take and therefore also create some false negatives. As we use symbolic execution, the false negatives could be limited by expressing a relation between the symbolic input and the found argument value and using this relation with the application source code to keep only a subset of the argument values.

Our dynamic analysis yields false positives for a particular argument group. However, after removing this group, the analysis yielded no more false positives but was tested on fewer applications than Saphire.

5.3.3. Effectiveness of analyses

The confidence that Saphire finds all system calls needed per API function is very high. The static analysis reads the interpreter binary line by line and accurately determines which system calls are used per function. This analysis, in combination with the reachability analysis, lists the system calls per API function accurately. The indirect calls it can not handle, but the dynamic analysis mends this. The dynamic analysis uses strace, ensuring that all executed system calls are found. The false positives, in combination with the line coverage reporting, are used to assert the effectiveness of the dynamic analysis.

We have already discussed that not all system call arguments per API function can be found with our analyses. This can be attributed to the dependence of these values on API arguments and to the large number of computations needed to come to the argument value.

5.3.4. Additional security

The security that Saphire and our extension provide is hard to compare as Saphire tests against known exploits while we do not. The exploits used by Saphire we cannot use for testing as Saphire would already protect against them. Finding other exploits that would circumvent Saphire is outside the scope of this thesis. Therefore, we assess the security impact of the system call arguments that can be restricted with static analysis and evaluate the risk of the values found using the dynamic analysis. Because most values found do not indicate a higher risk, we can confirm the usefulness of the analysis.

5.4. Limitations and future work

In this section, we discuss the limitations of the current solution and its evaluation, and ideas for future work.

The current solution implements an option for static and for dynamic analysis. The argument categories for the dynamic analysis are coarse-grained. Therefore, some more refined categorisation may be warranted. For example, split the “other" category into smaller sets. However, one must be careful not to overfit to limit the false positives only for the tested applications. A limitation to the static analysis is the two heuristics, which are used to reduce some workload. As discussed, these heuristics may impact the portion of arguments found during analysis. Finally, the tool angr used for symbolic executions has some limitations. There is no possibility of parallel execution, which means that the static analysis takes a long time. Furthermore, the technique called `MemoryWatcher`, which limits the allowed memory usage, does not always work appropriately, resulting in the analysis running out of memory.

Part of the evaluation of our solution depends on test suites for the three applications. The test suite for phpMyAdmin making use of Selenium was not working properly, and therefore, we had to use a test suite created from browsing traces which is less extensive. The Joomla application did not have any integration testing available and was also tested through browsing traces. The dynamic analysis reports zero false positives for the three tested applications after removing the arguments for length, size and offset. This does, however, not say that other applications will not suffer from false positives.

For future work, we discuss four ideas. Firstly, an improvement that could eliminate some of the false positives that occur for the static analysis. The analysis could be complemented by dynamic analysis, creating a hybrid version. The hybrid version should only restrict system call arguments that can be restricted by the static analysis, taking care of the blind spots (indirect calls) of static analysis while not creating new false positives by restricting system call arguments found only for the dynamic analysis. Secondly, the inclusion of pointer arguments in the analyses. Both the analyses and the filter system need to change to include this category. The seccomp filters cannot restrict pointer arguments, but there exist solutions that attempt to deal with pointer arguments. For the dynamic analysis, one would need to evaluate whether the pointer argument values do not depend too much on the test suite. Furthermore, appropriate parsing needs to be added. The static analysis will be more complex when including pointer arguments as the values for such arguments cannot be found by simply inspecting the registers. Thirdly, expressing relations between API arguments and system call arguments would create more complex policies that could restrict more arguments. However, a more advanced system call interposition system would be required to deal with such policies. The dynamic analysis may be able to create policies for the size, length and offset arguments. Fourthly, using the information from Saphire and our extension, developers could be informed about the API calls they are using in the applications, making them more aware of the security risks.

6

Related Work

This chapter discusses the literature on the topics of system call interposition, automation of policy generation and the necessary changes on both to include the arguments of system calls. The literature on system call interposition is older as the technique has been around for a while now. On the other hand, automation is a very active field where we can review state of the art. The last part on system call arguments is where we can compare our work to other literature.

6.1. System call interposition

The general definition of system call interposition is discussed in Section 2.3. In our work, we use seccomp for interposition. Here we discuss Janus, which is older than seccomp and still allowed the restriction of pointer arguments without taking care of any possible race condition problems. These race conditions and other security risks are discussed after Janus.

6.1.1. A secure environment for untrusted helper applications

The paper [19] proposes a user-based system call interposition system called Janus, which can be applied to helper applications. Helper applications may be handling untrustworthy data (for example, a document downloaded from the internet may be viewed in a document viewer, a helper application for the browser). Therefore, helper applications should not be trusted. The system's design is based on the belief that "An application can do little harm if its access to the underlying operating system is appropriately restricted." The idea is to let a process inspect the system calls of the untrusted process and let the inspecting process decide whether a system call is allowed or not. To implement this design, an operating system is required that will allow a "user-level process to watch the system calls executed by another process, and to control the second process in various ways (such as causing selected system calls to fail)". The writers found the operating system Solaris 2.4 most suitable. For each application that is to be traced, three steps are executed. First, a configuration file is read to create a dispatch table. The dispatch table stores all policies per system call number. Second, a child process is forked, and its state cleaned up. Third, this child process is used to run the application. The application runs and is only put to sleep when trying to perform a system call. The system call with its arguments is passed to the tracing process, which uses the dispatch table to decide whether to allow the system call. One of the design choices for this system was to keep it simple. The idea was that security through simplicity could be achieved.

Although simplicity is a good objective, it comes with limitations. Especially the interposition of system calls based on their arguments is tricky. There are several iterations of the Janus system, and the following paper discusses the problems and pitfalls encountered during this process. The paper uses a newer version of Janus, which has the same building blocks as this version.

6.1.2. Traps and pitfalls: practical problems in system call interposition based security tools

The paper [17] does not propose another system call interposition framework but is directed at designers of such frameworks and presents a set of practical problems. These problems are inspired by their framework called Janus, of which the previous section describes an earlier version. Five problems are described

with thoughts on possible solutions for some. The first warning is to prevent incorrectly replicating the OS. The state of the OS is needed for Janus to make policy decisions, presumably about system call arguments. The author's advice is to avoid replicating state and to minimise the amount of state required for policy decisions. A second problem is overlooking indirect paths to resources, which can cause Janus to grant access to resources that should be denied. Since paths are arguments given to system calls, this, again, has to do with (dis)allowing a system call based on its arguments. The third problem that is identified has to do with race conditions. Examples are symbolic link races, relative path races, argument races, file system information races and shared descriptor space races. Argument races, for instance, can occur when the argument inspected by the system call interposition system is not in kernel space. In this scenario, the argument can be changed in-between the checking by the system and the execution of the system call by the kernel. Thus argument races are only a problem for non-scalar arguments - which do not reside in kernel space - in a shared-memory environment. The authors discuss the advantages and disadvantages of several solutions. The first proposition is to copy sensitive arguments into the kernel. However, copying allows system calls to fetch their arguments from kernel memory which can have severe consequences when a mistake is made. Therefore, the authors would rather adopt less intrusive solutions such as protecting arguments in user space or checking that arguments do not reside in shared memory. The fourth problem for system call interposition identified is that creating the correct policies is complex. It may be hard to anticipate interactions between different system calls for an extensive application. The fifth problem is about what happens when a system call is denied execution by the system call interposition. The application that wants to perform the system call may not be able to handle such denial and can crash. This could leave the machine vulnerable to a DOS attack. Proposed solutions are emulation, redirection of calls or replication of resources.

This paper shows some of the problems a system call interposition system needs to tackle and why argument analysis is especially hard. However, most of these problems do not apply when considering only non-pointer arguments.

6.2. Automated policy generation

Multiple systems set out to automate policy generation. The setting for which they want to automate differs. For example, one can differentiate between binary or source code availability or the environment, such as a container, which is the focus of [18]. The systems can also be divided into three categories based on their analysis. The analysis of the program for policy generation can be static, dynamic or a combination of the two.

For the evaluation of the automated policies, multiple aspects can be considered. First, the number of false positives, meaning the number of system calls that are considered malicious by the policies but in reality are not, should be measured for normal program execution. Second, the percentage of attacks that can be averted with the generated policies and system call interposition could be published. The tested attacks are only a small portion of the possible attacks on the application in question. Third, automatically generated policies can be compared to policies manually crafted by professionals, which is done in [24].

6.2.1. Sysfilter: automated system call filtering for commodity software

Sysfilter, described in [13], is similar to Saphire as it does static analysis on binaries and uses seccomp to do the system call interposition. However, the framework only focuses on binaries and does not take the extra step to create automation for interpreted applications. Therefore, Sysfilter only consists of two parts, a system call set extraction and a system call set enforcement. The first part constructs "a safe over-approximation of the program's function-call graph (FCG), and performs a set of program analyses atop the FCG in order to extract the set of developer-intended syscalls". Interesting is that Sysfilter does not require debugging information for the binaries. Instead, the stack unwinding information is used to determine which part of a binary is used in the program.

Two downsides of Sysfilter are that it does not recover system call numbers when the value in the rax register is defined via instructions that involve memory operands, and neither does it take into account system call arguments. The last is to avoid pitfalls related to system call argument filtering that are described by [17].

The evaluation of Sysfilter reports on the system's correctness, performance, and effectiveness. The correctness is evaluated by testing ten applications comprised of 411 binaries which pass all tests. The effectiveness is evaluated by comparing the system calls allowed while using Sysfilter to system calls which can be used to exploit certain kernel vulnerabilities. Finally, they report on the percentage of binaries in their dataset that could still be affected.

6.2.2. Automating seccomp filter generation for Linux applications

A framework called Chestnut [12], consisting of two parts called Sourcalyzer and Binalyzer, has the same goal as Sysfilter, namely automating seccomp filter generation for applications. Although Sysfilter can only do this for compiled applications, Chestnut can do this for both compiled applications and source code. Depending on which is required, Binalyzer or Sourcalyzer is used respectively. Binalyzer scans the binaries for syscall instructions and uses the angr framework to do symbolic backward execution, which retrieves the system call numbers. A control-flow graph created with angr connects the system calls to built-in functions. Sourcalyzer requires the compilation and linking of the source code. It uses the LLVM compiler framework, which can detect syscalls in source code.

The correctness of the system call set determined by Chestnut is evaluated by executing test suites or by executing binaries with different configurations. Chestnut reports no crashes and therefore finds it reasonable to assume all system calls in the core functionality of the applications are found. For each application, they also report the line coverage obtained during testing. The security evaluation is extensive, reporting on how often dangerous system calls are blocked, the number of system calls in the allowed set which are not used by the application and the number of mitigated exploits. For small applications, Chestnut performs very well on mitigation of exploits (about 80 to 90% is mitigated) but has a harder time with larger applications (about 36-38%), which is to be expected as larger applications have more system calls and a more complex control-flow graph.

Comparing Chestnut to Sysfilter, it is apparent that Binalyzer and Sysfilter achieve the same goal. They do, however, differ in their approach. Binalyzer uses the angr framework to get the control-flow graph of the application. Sysfilter does not rely on a framework but on its own implementation to extract a function call graph from the binary. A function call graph contains fewer details than a control-flow graph and is, therefore, a more lightweight solution.

6.3. System calls arguments

Creating system call interposition systems that allow for filtering the arguments of system calls allows for more fine-grained filters. However, argument filtering is complicated. On top of the problems with the interposition comes the complexity of the automation for policies that include system calls arguments. This makes only a limited amount of research available on the topic. Most researchers refer to it as future work, and others do not even consider it. Nevertheless, the following four all believe that the filtering of system call arguments and the automation of generating appropriate policies are worthwhile.

6.3.1. Systrace

Paper [25] proposes a framework called Systrace that tackles both system call interposition and automatic policy generation. The system call interposition is a hybrid approach implemented in user space and at kernel level. Systrace has its own policy language to filter system calls. The policies are automatically generated during training sessions or interactively during program execution. The automatic policy generation is thus achieved through dynamic analysis. Before policies are checked, the Systrace kernel part evaluates the system call. It can permit or deny or inspect the policies by asking the Systrace part in user space. When inspecting the system call in user space, the arguments of the system call are translated (and normalised), and the policy is checked.

Systrace prevents “race conditions by replacing the system call arguments with the arguments that were resolved and evaluated by Systrace. The replaced arguments reside in kernel address space and are available to the monitored process via a read-only look-aside buffer. This ensures that the kernel executes only system calls that passed the policy check.".

As one of the analysis methods, we use dynamic analysis like Systrace. However, Systrace does not describe any exclusion of arguments that may depend too much on the training session. Therefore, the risk of false positives is much larger for Systrace than for our system.

6.3.2. Shredder

S. Mishra and M. Polychronakis focus on a Windows environment and observe that removing unused system API functions does not always provide us with the needed security. Often, system API functions are needed for the normal operation of a program. They notice that part of the API functions capabilities may not be needed and propose to do so-called API specialisation. Hence, the interface to the API functions is restricted. In practice, the arguments given to the API functions will be restricted. The framework they implement for

closed-source Windows applications is called Shredder [22]. It achieves value identification for over fifty per cent of the arguments of analysed Windows API functions.

Shredder consists of a static analysis part to generate policies and a protection part to enforce the policies. All call sites of API functions are detected during static analysis, and arguments are determined by backwards data flow analysis. This detection seems to be a more lightweight form than provided by the angr framework. The function is classified as known when argument values can be derived from the static data or constant values. When one or more call sites of a function do not return argument values, the function will be classified as unknown, and the API function will not be restricted. This approach is conservative and reduces the chances of false positives. The policies can denote the values an argument can take as well as the equality relation between arguments.

Although there is not a very detailed explanation of the policy generation, Shredder does report the following. It builds on the IDA Pro disassembler with IDAPython, scanning the executables for call sites utilising the import table. It uses IDA pro's stack variables window data structure which contains local variables and function arguments. Using a control flow graph, it performs backwards data flow tracking to identify values derived from a previous function. They found that a maximum recursion depth of three is optimal.

Considering if the approach of Shredder with the control flow graph would benefit our design, we must conclude that, unfortunately, it is not feasible for a program as extensive as the interpreter. The call graph our system uses contains less information but is better suited for the interpreter.

6.3.3. Automated policy synthesis of system call sandboxing: Abhaya

Pailoor et al. in [24] also conclude that system call arguments should be taken into account when automatically deriving policies for system call interposition. One of their observations is that “integer-valued syscall arguments often serve as important flags and crucially affect policy choice". Their technique, named Abhaya, can automate policy generation for C and C++ programs in two policy domain-specific languages, one of which is seccomp-bpf. To generate policies, first, the program is translated to a language of which the semantics are described in the paper. To reason about the program, Abhaya uses a set of derivation rules. With these rules and several algorithms, policies are generated in the preferred policy domain-specific language.

The generated policies seem very good as the majority of those tested against expert, handcrafted ones match. Since Abhaya is built upon LLVM, it is dependent on source code and cannot be used on binaries. Although the implementation is created for C/C++ applications, the framework should also work on other C-like languages.

Doing static analysis on source code is more straightforward than on binaries as there is more information for the reasoning. Abhaya is therefore working great on source code, but binaries may be too complex for such a process.

6.3.4. Authenticated system calls

System call interposition is a form of intrusion detection and prevention. Specifically, it is a form of host-based intrusion detection. Intrusion detection based on system call inspection faces the same problems as policy generation, namely deciding which system calls to allow. Intrusion prevention based on system calls also needs to deal with interposition. Strictly speaking, intrusion detection does not actively intervene, while intrusion prevention does. However, this distinction does not always hold nowadays as modern intrusion detection systems also stop detected threats [14]. One of such papers which also considers the arguments of system calls is [26]. The paper describes the use of authenticated system calls, which are system calls that have been transformed to include additional arguments to specify the policy, among other things. Therefore, a small kernel modification is necessary. Apart from the kernel modification, the program binary is modified to include these authenticated system calls. This modification follows after the static analysis of the binary to determine system call policies. Not needing a policy daemon makes this work a novel approach to system call interposition and allows them to steer clear of some of the pitfalls around system call arguments discussed earlier. Furthermore, with this strategy, the authors are able to detect between 30 to 40% of system call arguments on some real-world programs. The static analysis of the program works with PLTO, a link-time instrumentation and optimisation tool [28]. The analysis determines system call arguments by examining the values pushed onto the stack prior to a system call and “applying a standard reaching definitions analysis from PLTO".

Similar to our static analysis, this system makes use of reachability. However, the reachability for the interpreter needs to be determined per API function. Since the interpreter depends heavily on its input, we expect that utilising PLTO will be more complex than the current system in place for authenticated system

calls.

6.4. Considerations

We can conclude that caution is advised when dealing with system call interposition, especially for arguments of system calls. Additionally, we can note that automation of policy generation is recently more focussed on correctness, resulting in more research using static analysis as this creates an overestimation of the necessary system calls and prevents false positives. Some adaptations are necessary to use system call interposition and automation of policy generation on system call arguments. For system call interposition, one can think about the use of kernel space as is done in Systrace or the use of authenticated system calls described in [26]. The approach to automated policy generation varies depending on the setting. It is clearly easier to determine the arguments of system calls when source code is available. Also, it is not always possible to determine the value of all arguments since they can "depend on user input, generated as a result of a system call, or may be unknown because of pointer aliasing" [26]. For interpreted applications, we determine the system calls and their arguments used by the interpreter and will have additional unknown input from the interpreted application leading to even fewer arguments that can be determined.

We have drawn comparisons with every system call argument interposition system to our system and sum them up in Table 6.1.

	Properties		**Comparison**
	Applied to	**Analysis type**	
Systrace	binary	dynamic	no exclusion of argument groups during dynamic analysis
Shredder	binary	static	control flow graph not feasible for interpreter binary
Abhaya	source code	static	limited to availability of source code of compiled application
ASC	binary	static	execution path of interpreter depends heavily on input

Table 6.1: Comparison of system call argument policy generating systems

7

Conclusion

This thesis aimed to assess the effectiveness of system call argument interposition for interpreted applications. Based on the comparison of both a static and dynamic analysis solution, it can be concluded that proper policies can be generated to restrict system call argument usage for interpreted applications effectively. The dynamic analysis solution showed the most promise with an argument restriction rate of 40%.

The solution for the static analysis was much more extensive than for the dynamic analysis. Unfortunately, this design did not create more valuable results as the percentage of restricted arguments was below 10%. This low percentage could have been accepted due to the limited knowledge available if not for the false positives that appeared. Usually, one would not expect any false positives in a static analysis, but this can be accounted for by the fact that the symbolic analysis is not purely static and that part of the code was not inspected due to indirect calls. Although the dynamic analysis is prone to false positives as well, the restriction percentage was a lot higher. The set of arguments left out of the dynamic analysis contained: arguments that the filter solution seccomp could not deal with, namely pointer arguments; file descriptor arguments, because they are highly likely to depend on the output of other system calls; and arguments that appeared to have a high dependence on the test suite, the size, length and offset arguments. The portion of arguments used in the analysis was mainly restricted to harmless values, and thus these restrictions improved security. We also found that most arguments restricted by the static analysis were contributory to security.

As the dynamic analysis design was kept fairly simple, further research can improve this design to achieve a higher restriction rate and possibly reduce the false positives. For the static analysis, future research should focus on eliminating the false positives by creating a hybrid solution.

This research provides more insight into creating a more fine-grained interposition solution for interpreted programs which can further reduce the tools available to attackers to create a successful RCE attack. The two approaches evaluated showed that although the problem is more complex than for compiled applications, there certainly are steps that can be taken.

Bibliography

[1] Seccomp security profiles for docker. URL https://docs.docker.com/engine/security/seccomp/. (visited on April 19th 2022).

[2] Drupal. URL https://www.drupal.org/. (visited on May 25th 2022).

[3] Debian Security Advisory. URL https://www.debian.org/security/2014/dsa-2949. (visited on May 20th 2022).

[4] Joomla. URL https://www.joomla.org/. (visited on May 25th 2022).

[5] phpMyAdmin. URL https://www.phpmyadmin.net/. (visited on May 25th 2022).

[6] SQLite. URL www.sqlite.org. (visited on May 1st 2022).

[7] World wide web technology surveys. URL https://w3techs.com/. (visited on February 1st 2022).

[8] Nadav Avital and Gilad Yehudai. https://www.imperva.com/blog/crypto-mining-drives-almost-90-of-all-rce-attacks/, February 2018.

[9] Massimo Bernaschi, Emanuele Gabrielli, and Luigi V.Mancini. Enhancements to the linux kernel for blocking buffer overflow based attack. In *4th Annual Linux Showcase & Conference*, 2000.

[10] J. Glenn Brookshear and Dennis Brylow. *Computer Science: an overview*. Pearson, 12 edition, 2015.

[11] Alexander Bulekov, Rasoul Jahanshahi, and Manuel Egele. Saphire: Sandboxing PHP applications with tailored system call allowlists. In *USENIX Security Symposium*. USENIX Association, 2021.

[12] Claudio Canella, Mario Werner, Daniel Gruss, and Michael Schwarz. Automating seccomp filter generation for linux applications. In *Cloud Computing Security Workshop 2021*. ACM, 2021.

[13] 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)*, pages 459–474, 2020.

[14] Christian Doerr. *Network Security in Theory and Practice*. 2020.

[15] Jake Edge. https://lwn.net/Articles/656307/, September 2015. Linux Plumbers Conference.

[16] The Apache Software Foundation. Apache Log4j Security Vulnerabilities. https://logging.apache.org/log4j/2.x/security.html, December 2021.

[17] Tal Garfinkel. Traps and pitfalls: Practical problems in system call interposition based security tools. In *Network and Distributed Systems Security Symposium*, 2003.

[18] 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)*, pages 443–458, 2020.

[19] Ian Goldberg, David Wagner, Randi Thomas, and Eric A Brewer. A secure environment for untrusted helper applications: Confining the wily hacker. In *Proceedings of the 1996 USENIX Security Symposium*, volume 19. USENIX Association Berkeley, CA, 1996.

[20] Kapil Jain and R Sekar. User-level infrastructure for system call interposition: A platform for intrusion detection and confinement. In *Network and Distributed Security Symposium*. Citeseer, 2000.

[21] Michael Kerrisk. Linux manual pages. URL https://man7.org/linux/man-pages/. (visited between September 2021 and June 2022).

[22] Shachee Mishra and Michalis Polychronakis. Shredder: Breaking exploits through api specialization. In *Proceedings of the 34th Annual Computer Security Applications Conference*, pages 1–16. ACM, 2018.

[23] Mitre. CVE Drupalgeddon. URL `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-3704`. (visited on September 20th 2021).

[24] Shankara Pailoor, Xinyu Wang, Hovav Shacham, and Isil Dillig. Automated policy synthesis for system call sandboxing. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):1–26, 2020.

[25] Niels Provos. Improving host security with system call policies. In *USENIX Security Symposium*, pages 257–272, 2003.

[26] Mohan Rajagopalan, Matti A Hiltunen, Trevor Jim, and Richard D Schlichting. System call monitoring using authenticated system calls. *IEEE Transactions on Dependable and Secure Computing*, 3(3):216–229, 2006.

[27] Derick Rethans. Xdebug. URL `https://xdebug.org/`. (visited on April 20th 2022).

[28] Benjamin Schwarz, Saumya Debray, Gregory Andrews, and Matthew Legendre. PLTO: A link-time optimizer for the intel ia-32 architecture. In *Proc. 2001 Workshop on Binary Translation (WBT)*, 2001.

[29] Security Bulletin Citrix. `https://support.citrix.com/article/CTX267027`, October 2020.

[30] Yan Shoshitaishvili, Ruoyu Wang, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. Firmalice–automatic detection of authentication bypass vulnerabilities in binary firmware. In *NDSS*, volume 1, 2015.

[31] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, et al. Sok:(state of) the art of war: Offensive techniques in binary analysis. In *IEEE Symposium on Security and Privacy (SP)*, pages 138–157. IEEE, 2016.

[32] Nick Stephens, John Grosen, Christopher Salls, Audrey Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Driller: Augmenting fuzzing through selective symbolic execution. In *NDSS*, 2016.

[33] Claudio Viviani. Exploit Database Drupalgeddon, October 2014. URL `https://www.exploit-db.com/exploits/34992`. (visited on September 20th 2021).

A

Evaluation plot

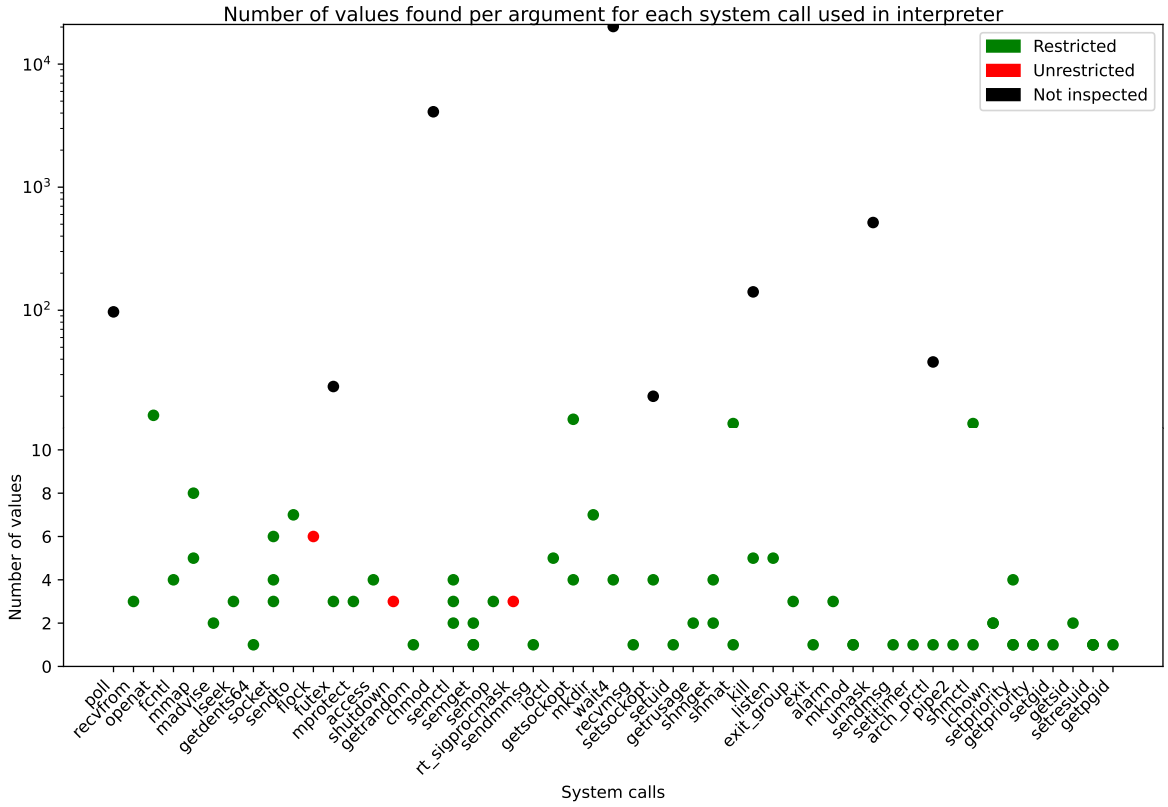


Figure A.1: The black dots indicate the arguments in the interpreter that are not evaluated